Formation R Initiation

Martin Chevalier (Insee)

Ce document est la version imprimable du support de la formation R initiation des 8 et 9 juin 2017.

Les supports de cette formation ont été conçus sous RStudio avec R Markdown et compilés le 11/06/2017. Certains éléments de mise en forme du site compagnon sont repris de l'ouvrage R packages de Hadley Wickham.

Un grand merci aux participants pour leurs nombreuses remarques et suggestions particulièrement constructives.

Module 1

Prise en main du logiciel

Un peu d'histoire et quelques grands principes

R est un langage utilisé pour le traitement de données statistiques créé au début des années 1990 par deux chercheurs de l'université d'Auckland, Ross Ihaka and Robert Gentleman. Il reprend de très nombreux éléments du langage S créé par le statisticien américain John Chambers à la fin des années 1970 au sein du laboratoire Bell.

La première version stable a été rendue publique en 2000 : d'abord principalement diffusé parmi les chercheurs et les statisticiens « académiques », R est aujourd'hui **de plus en plus utilisé au sein des Instituts nationaux de statistiques**.

R : un logiciel libre

À la différence d'autres logiciels de traitement statistique (SAS, SPSS ou Stata notamment), R est un **logiciel libre** : sa licence d'utilisation est gratuite et autorise chaque utilisateur à **accéder, modifier ou redistribuer son code source**. En pratique, il est maintenu par une équipe (la *R Core Team*) qui veille à la stabilité du langage et de ses implémentations logicielles.

Une des conséquences de cette philosophie « libre » présente dès les premières années du développement du langage est le rôle qu'y jouent les **modules complémentaires**, ou ***packages***. Au-delà des « briques » fondamentales de la *R Core Team*, **plusieurs milliers de *packages* sont disponibles et librement téléchargeables** *via* le *Comprehensive R Archive Network* (ou CRAN) ou encore par le biais de plate-formes de développement collaboratif comme GitHub. Ces *packages*, dont l'installation est particulièrement simple dans R, enrichissent considérablement les fonctionnalités du logiciel et sont une de ses principales forces.

Remarque importante Comme de nombreux logiciels libres, R est très influencé par le fonctionnement du système d'exploitation Linux. À ce titre, **certains éléments de sa syntaxe peuvent dérouter un utilisateur de Windows** :

- **R est sensible à la casse** : il distingue ainsi `matable` de `MATABLE` ou encore de `maTable`, même sous Windows (contrairement à SAS notamment) ;
- **dans R les chemins doivent utiliser des / et non des ** : ainsi, pour pointer vers le dossier `U:\monEnquete\donnees` il faut saisir dans R `U:/monEnquete/donnees`.

De manière plus générale, le fonctionnement de R est **plus proche de celui d'un langage de programmation « classique »** (Python, C, Java, etc.) **que de celui des autres logiciels de traitements statistiques**. Une manière d'introduire cet aspect fondamental du logiciel est de développer la **célèbre citation de John Chambers** :

> *To understand computations in R, two slogans are helpful :*
> — *Everything that exists is an object.*
> — *Everything that happens is a function call.*
> *John Chambers*

« Tout ce qui existe est un objet »

Tout ce qui existe et est manipulable dans R est un **objet** identifié par son nom et par son **environnement de référence**. Par défaut, tous les objets créés par l'utilisateur

apparaissent dans l'environnement dit « global » (`.GlobalEnv`) qui est implicite, de façon analogue à la bibliothèque `WORK` de SAS.

Pour créer un objet, la méthode la plus simple consiste à assigner une valeur à un nom avec l'opérateur `<-`. Par exemple :

```
a <- 4
```

assigne la valeur 4 à l'objet `a` (dans l'environnement global). Dès lors, il est possible d'afficher la valeur de `a` et de la **réutiliser dans des calculs** :

```
# Affichage de la valeur de a avec la fonction print() ...
print(a)
  ## [1] 4

# ... ou tout simplement en tapant son nom
a
  ## [1] 4

# Utilisation de a dans un calcul
2 * a
  ## [1] 8

# Définition et utilisation de b
b <- 6
a * b
  ## [1] 24
```

Il est bien sûr possible d'assigner à un nom **non pas une valeur numérique unique** (comme ici 4 à `a` et 6 à `b`) **mais des données provenant d'une table externe**.

Exemple Le code suivant associe à l'objet `reg` les caractéristiques des régions dans le Code officiel géographique (COG) au 1er janvier 2017.

```
# Lecture du fichier du COG contenant le nom des régions
# et stockage dans l'objet dont le nom est `reg`
reg <- read.delim("reg2017.txt")
```

```
# Affichage de l'objet reg
reg
```

```
##    REGION CHEFLIEU TNCC                        NCC
## 1       1    97105    3                 GUADELOUPE
## 2       2    97209    3                 MARTINIQUE
## 3       3    97302    3                     GUYANE
## 4       4    97411    0                 LA REUNION
## 5       6    97608    0                    MAYOTTE
## 6      11    75056    1              ILE-DE-FRANCE
## 7      24    45234    2       CENTRE-VAL DE LOIRE
```

```
## 8      27    21231    0    BOURGOGNE-FRANCHE-COMTE
## 9      28    76540    0                  NORMANDIE
## 10     32    59350    4          HAUTS-DE-FRANCE
## 11     44    67482    2                 GRAND EST
## 12     52    44109    4          PAYS DE LA LOIRE
## 13     53    35238    0                   BRETAGNE
## 14     75    33063    3        NOUVELLE-AQUITAINE
## 15     76    31555    1                 OCCITANIE
## 16     84    69123    1      AUVERGNE-RHONE-ALPES
## 17     93    13055    0 PROVENCE-ALPES-COTE D'AZUR
## 18     94    2A004    0                     CORSE
```

Dans tous les cas, les objets créés sont **stockés dans la mémoire vive de l'ordinateur** (comme dans Stata), ce qui présente des avantages et des inconvénients :

- (+) on ne modifie jamais les fichiers originaux, uniquement les objets chargés en mémoire ;
- (+) les opérations sur les objets chargés peuvent être **extrêmement rapides**, car elles ne nécessitent pas de lire des données sur le disque ;
- (-) à chaque lancement de R il faut **recharger les données nécessaires en mémoire** ;
- (-) la **taille totale des données chargées ne peut pas excéder celle de la mémoire vive installée** (80 Go partagés sur un serveur AUS actuellement).

« Tout ce qui se produit est un appel de fonction »

Une fois les objets sur lesquels on souhaite travailler créés (*i.e.* les tables importées), R dispose d'un grand nombre de **fonctions** pour transformer ces données et mener à bien des traitements statistiques. **Dans R une fonction est un type d'objet particulier** : une fonction est identifiée par son nom (dans un environnement de référence) suivi de parenthèses.

Exemple La fonction `ls()` (sans argument) permet d'afficher les objets chargés en mémoire.

```
# Affichage des objets chargés en mémoire avec ls()
ls()
  ## [1] "a"   "b"   "reg"
```

Il y a pour l'instant trois objets en mémoire : `a`, `b` et `reg`.

Progresser dans la maîtrise de R signifie essentiellement étendre son « vocabulaire » de fonctions connues. Avec le temps, il est fréquent que l'on revienne sur d'anciens codes pour les simplifier en utilisant des fonctions découvertes entre temps (ou parfois en exploitant mieux les mêmes fonctions !).

Il est également extrêmement facile et courant dans R de créer ses propres fonctions.

Exemple La fonction `monCalcul()` renvoie le résultat de `param1 * 10 + param2`, où `param1` et `param2` sont deux paramètres.

```
# Définition de la fonction monCalcul()
monCalcul <- function(param1, param2){
  resultat <- param1 * 10 + param2
  return(resultat)
}

# Test de la fonction monCalcul() avec les valeurs 1 et 3
monCalcul(1, 3)
  ## [1] 13

# Test de la fonction monCalcul() avec les valeurs a et 2
a
  ## [1] 4
monCalcul(a, 2)
  ## [1] 42
```

Quand on saisit uniquement le **nom de la fonction (sans parenthèse)**, R affiche son code :

```
# Affichage du code de la fonction monCalcul()
monCalcul
  ## function(param1, param2){
  ##   resultat <- param1 * 10 + param2
  ##   return(resultat)
  ## }
  ## <environment: 0x7f8f238>
```

À noter que **rien ne distingue les fonctions pré-chargées dans le logiciel** (comme `read.delim()` ou `ls()` utilisées précédemment) **des fonctions créées par l'utilisateur**. Il est ainsi tout à fait possible d'afficher le code de ces fonctions.

```
# Affichage du code de la fonction read.delim()
read.delim
  ## function (file, header = TRUE, sep = "\t", quote = "\"", dec = ".",
  ##     fill = TRUE, comment.char = "", ...)
  ## read.table(file = file, header = header, sep = sep, quote = quote,
  ##     dec = dec, fill = fill, comment.char = comment.char, ...)
  ## <bytecode: 0x6a87990>
  ## <environment: namespace:utils>
```

C'est une **conséquence du caractère « libre » du logiciel** : non seulement le code des fonctions pré-chargées est consultable, mais il est également modifiable.

Exemple Il est tout à fait possible dans R (même si cela n'a *a priori* pas grand intérêt…) de modifier la signification des signes arithmétiques (qui comme toutes les autres opérations dans R correspondent à des fonctions).

```
# On décide d'associer au signe + l'opération effectuée habituellement
# par le signe - :
`+` <- `-`

# Le signe + est désormais associé à la soustraction :
2 + 2
  ## [1] 0
```

Cet exemple illustre la **très grande souplesse de R comme langage** : tous ses aspects sont modifiables, si bien qu'il est possible de **développer facilement des programmes R parfaitement adaptés aux besoins les plus spécifiques**.

Découverte de l'interface

En tant que tel, R est un *langage* susceptible d'être implémenté dans de nombreuses interfaces. Le choix est fait ici de présenter d'abord son **implémentation minimale** (en mode « console ») puis une **implémentation beaucoup plus complète** par le biais du programme RStudio. Dans tous les cas, la plate-forme utilisée est Windows.

Effectuer des manipulations de base dans la console

Par défaut sous Windows, R est fourni avec une interface graphique minimiale (`Rgui.exe`), dont la fenêtre principale est une **console**, c'est-à-dire un **terminal** dans lequel taper des instructions (comparable à l'invite de commandes Windows). Les instructions sont à taper après le signe `>` en rouge.

Toutes les commandes peuvent être passées au logiciel par le biais de la console, même si **en pratique les commandes les plus longues sont stockées et soumises depuis un fichier de script** (*cf.* la sous-partie suivante). En particulier, il est fréquent d'effectuer dans la console :

- **des assignations et des rappels de valeur** : le signe `<-` permet d'assigner des valeurs à des noms pour être réutilisées ultérieurement. Quand une valeur est assignée à un nom, il suffit de taper le nom dans la console pour afficher la valeur.
- **des opérations sur les objets en mémoire** :
  - la fonction `ls()` affiche tous les objets en mémoire ;

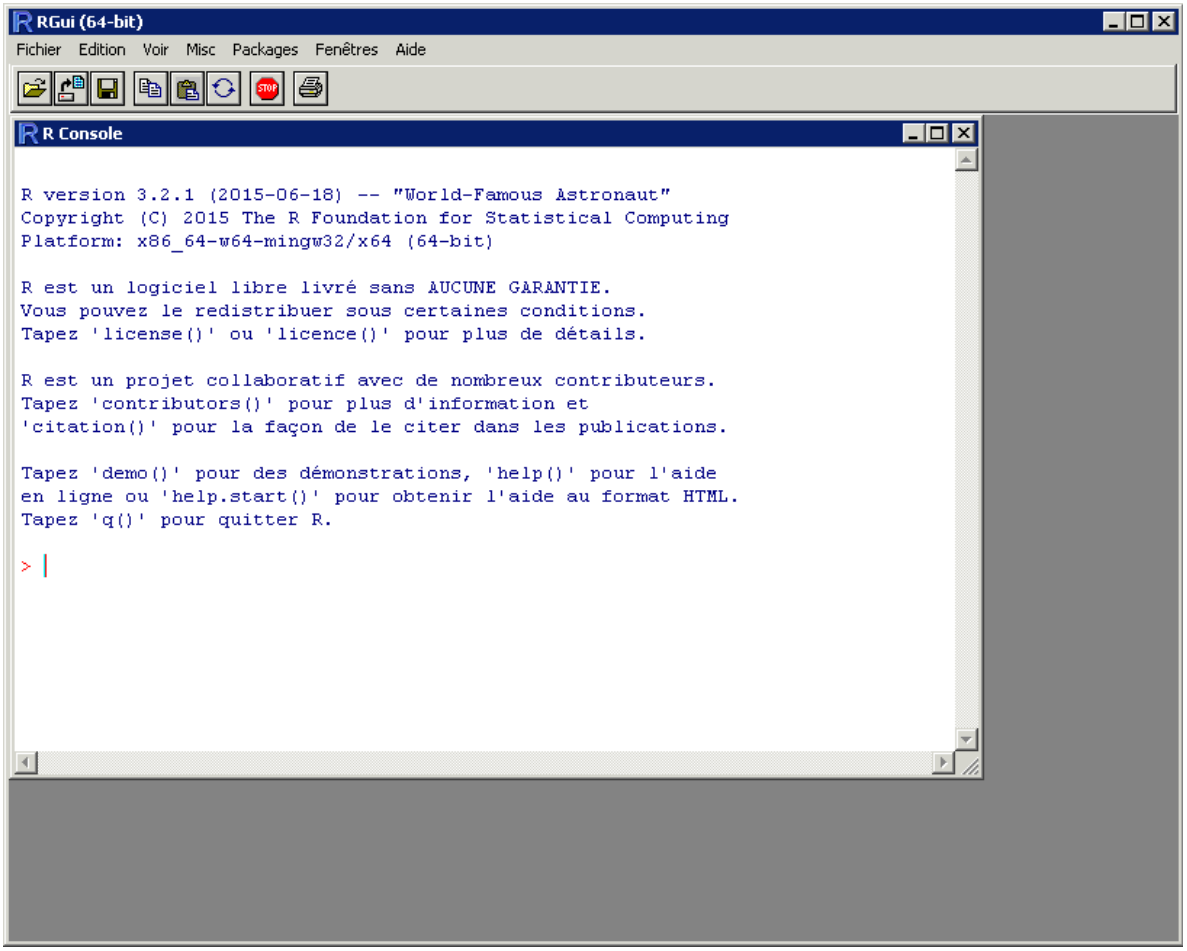


FIGURE 1.1 – Interface fenêtrée de R sous Windows

  - la fonction `str(a)` affiche les caractéristiques (ou encore la *structure*) de l'objet `a` (son type, sa longueur, etc.) ;
  - la fonction `rm(a)` supprime l'objet `a`.
- **des requêtes pour l'aide** : pour afficher l'aide sur une fonction dont le nom est `maFonction`, il suffit d'utiliser `help(maFonction)` ou plus simplement `?maFonction`.
- **des opérations simples** : le tableau ci-dessous présente quelques opérations arithmétiques et les symboles correspondant en R.

Code R	Résultat
`a + b`	Somme de a et b
`a - b`	Soustraction de b à a
`a * b`	Produit de a et b
`a / b`	Division de a par b
`a ^ b`	a puissance b
`a %/% b`	Quotient de la division euclidienne de a par b
`a %% b`	Reste de la division euclidienne de a par b
`sqrt(a)`	Racine carrée de a

Cas pratique 1.1 Convertir une durée de secondes en minutes-secondes

Il est souvent très utile de mesurer et d'afficher la durée d'un traitement un peu long (script exécuté régulièrement par exemple). La fonction `system.time()` de R n'affiche néanmoins que le temps écoulé en *secondes*, ce qui n'est guère lisible. **L'objectif de ce cas pratique est de convertir une durée de secondes en minutes-secondes**.

a. Ouvrez une session AUS (en utilisant votre idep et votre mot de passe) et lancez le programme R (pas Rstudio).

b. Dans la console, associez la valeur 2456 à l'objet `duree`. C'est sur cette durée (en secondes) que vont porter tous les calculs. Une fois assignée, rappelez la valeur de `duree` dans la console.

```
# Note : Pour copier-coller les éléments de solution,
# vous pouvez utiliser les raccourcis clavier Ctrl + C
# (copier) et Ctrl + V (coller).

# Pour assignez une valeur à un objet, on utilise l'opérateur <-
duree <- 2456
# Pour soumettre une opération en mode console, il suffit d'appuyer
# sur ENTREE.
```

```
# Dès lors qu'une valeur est assignée à un objet, il suffit de taper
# le nom de l'objet dans la console pour rappeler sa valeur :
duree
  ## [1] 2456
```

---

c. Calculez le nombre de minutes correspondant à la valeur de `duree`. Comment obtenir un nombre entier (*cf.* le tableau des opérations arithmétiques) ? Associez cette valeur à l'objet `min`.

```
# Pour obtenir le nombre de minutes dans duree, il suffit de
# diviser par 60 :
duree / 60
  ## [1] 40.93333
# Néanmoins, pour obtenir un nombre entier de minutes, il faut utiliser
# la division euclidienne (cf. tableau)
duree %/% 60
  ## [1] 40
# On stocke le nombre de minutes dans l'objet min avec <-
min <- duree %/% 60
min
  ## [1] 40
```

---

d. Calculez le nombre de secondes restantes une fois le nombre de minutes déterminé. Vous pouvez utiliser la flèche ↑ du clavier pour rappeler et modifier le code que vous venez de soumettre. Associez cette valeur à l'objet `sec`.

```
# Calculer le nombre de secondes restantes revient à calculer
# le reste de la division euclidienne de duree par 60 : c'est ce
# que fait l'opérateur %% (cf. tableau).
duree %% 60
  ## [1] 56
# A nouveau, on rappelle le code saisi précédemment avec la flèche
# HAUT du clavier et on associe la valeur à l'objet sec avec l'opérateur <-.
sec <- duree %% 60
sec
  ## [1] 56
```

---

e. Utilisez la fonction `help()` (ou de façon équivalente `?`) pour recherchez de l'aide sur la fonction `paste()`. Que se passe-t-il quand vous soumettez le code

```
paste("La durée est de", duree, "secondes.")
```

En utilisant tous ces éléments, afficher dans la console le texte :

```
## [1] "Le traitement a duré `min` minutes et `sec` secondes."
```

```
# Pour consulter l'aide de R, il suffit d'utiliser la fonction help()
# avec entre parenthèses le nom de la fonction sur laquelle on souhaite
# obtenir de l'aide.
help(paste)
# On peut aussi plus simplement utiliser ? :
? paste
# Un navigateur s'ouvre alors pour afficher la page d'aide de la fonction.
```

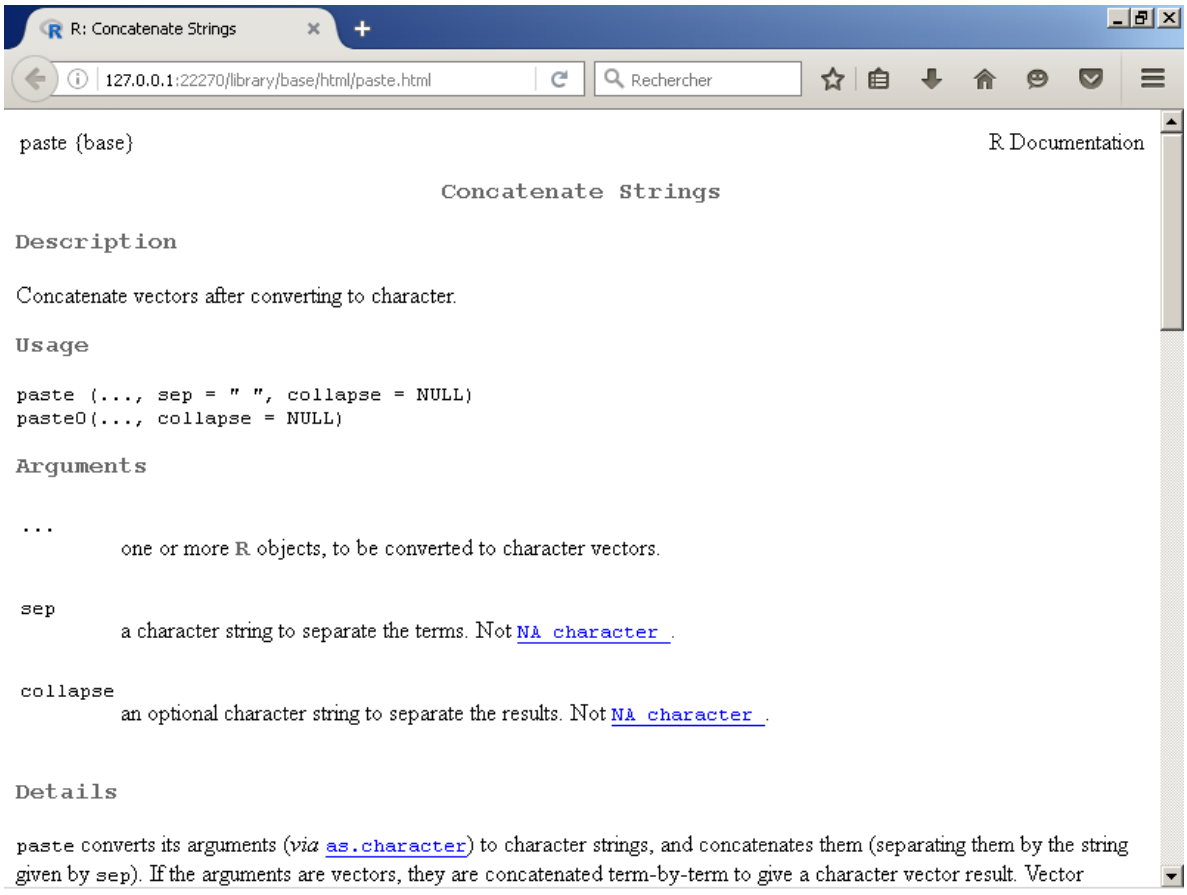


FIGURE 1.2 – Aide de la fonction paste()

```
# On y apprend que cette fonction sert à concaténer des "vecteurs"
# (cf. module 2 de la formation). En soumettant l'exemple proposé,
# on obtient :
```

```
paste("La durée est de", duree, "secondes.")
  ## [1] "La durée est de 2456 secondes."

# Ce code produit une chaîne de caractère : les chaînes de texte
# (entre "") sont reproduites telles quelles mais l'objet `duree`
# est remplacé par sa valeur. Les différents arguments de la fonction
# paste() sont séparés par des virgules.

# Pour répondre à la question, il suffit donc d'utiliser les objets
# `min` et `sec` définis précédemment comme arguments de la fonction paste() :
paste("Le traitement a duré", min, "minutes et", sec, "secondes.")
  ## [1] "Le traitement a duré 40 minutes et 56 secondes."
```

Cas pratique 1.2 Manipuler des objets en mémoire

Par défaut en mode console, l'utilisateur ne dispose d'aucune information sur les objets stockés en mémoire. **L'objectif de ce cas pratique est de vous familiariser avec les principales fonctions de manipulation des objets en mémoire.**

a. Utilisez la fonction `ls()` (sans argument) pour afficher les objets actuellement stockés en mémoire. Affectez la valeur 567 à l'objet `Duree` (avec un `D` majuscule) et relancez la fonction `ls()`. Pourquoi R distingue-t-il les objets `duree` et `Duree` ?

```
# Utilisée sans arguments, la fonction ls() liste tous les objets
# stockés dans l'environnement de référence de R
ls()
  ## [1] "duree" "min"   "sec"
Duree <- 567
ls()
  ## [1] "duree" "Duree" "min"   "sec"
# R distingue les objets duree et Duree car il est sensible à la casse.
```

---

b. Associez à l'objet `monTexte` la chaîne de caractère `"Hello world!"`. En utilisant la fonction `str()`, comparez les caractéristiques des objets `duree` et `monTexte`. À quel type chacun de ces deux objets appartient-il ?

```
# Comme dans la fonction paste(), la manipulation de chaînes
# de caractère nécessite l'utilisation de guillemets ""
monTexte <- "Hello world!"
ls()
  ## [1] "duree"    "Duree"    "min"      "monTexte" "sec"

# La fonction str(a) permet d'afficher les caractéristiques
# de l'objet a.
str(duree)
  ##  num 2456
str(monTexte)
  ##  chr "Hello world!"
# duree est un objet de type numérique alors que
# monTexte est un objet de type caractère.
```

---

c. Utilisez la fonction `rm()` pour supprimer les objets `Duree` et `monTexte`. Vérifiez que la suppression est effective (et n'a pas affecté `duree`) en relançant la fonction `ls()`.

```
# Objets présents au début de la question
ls()
  ## [1] "duree"    "Duree"    "min"      "monTexte" "sec"

# La fonction rm() (de l'anglais remove) permet de supprimer
# un ou plusieurs objets.
rm(Duree, monTexte)
ls()
  ## [1] "duree" "min"   "sec"
# Les objets Duree et monTexte ont bien été supprimés.
```

---

d. Recherchez de l'aide sur la fonction `rm()`, et plus spécifiquement sur son argument `list`. En utilisant cet argument combiné avec la fonction `ls()`, écrivez une instruction qui supprime tous les objets dans l'environnement de référence de R.

```
# Pour recherchez de l'aide sur une fonction, on utilise tout simplement ?
? rm
```

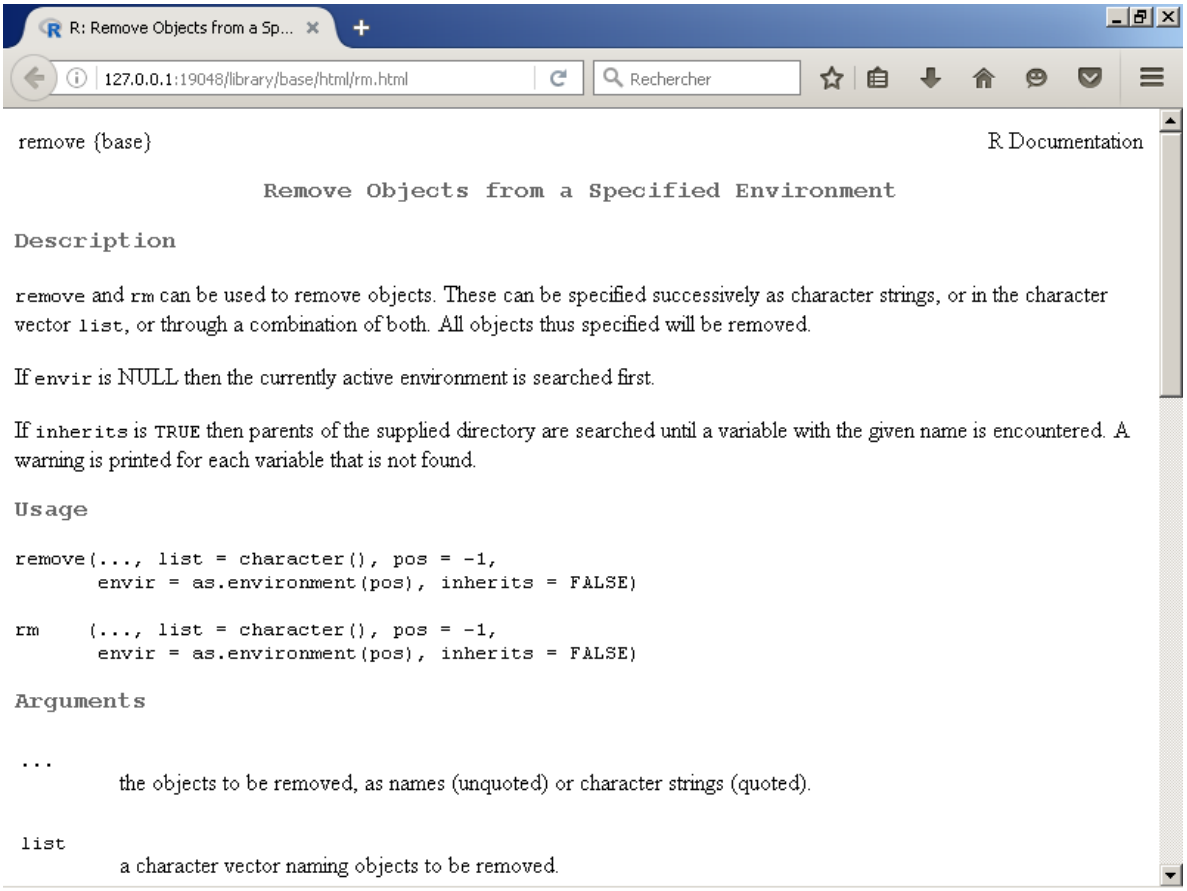


FIGURE 1.3 – Aide de la fonction rm()

```r
# L'argument list permet de spécifier les objets à supprimer
# sous la forme d'une vecteur de type caractère. Or c'est précisément
# ce que produit la fonction ls() :
ls()
  ## [1] "duree" "min"   "sec"

# Pour supprimer tous les éléments en une seule commande, il suffit
# de spécifier le résultat de la commande ls() à l'argument list de la
# fonction rm() :
rm(list = ls())

# On vérifie alors qu'il n'y a plus aucun objet
# dans l'environnement de référence :
ls()
  ## character(0)
```

Utiliser des scripts dans RStudio

Quoique toutes les fonctionnalités de R soient accessibles en mode console, ce type d'interface présente l'inconvénient majeur de **ne pas permettre de garder facilement une trace du code saisi** (sinon par le biais de l'historique des commandes accessible par ↑). Pour combler ce manque, les différentes interfaces graphiques de R permettent d'utiliser des **scripts** au format `.R`, à l'image des éditeurs de SAS (fichiers `.sas`) ou des *do-file* de Stata (fichiers `.do`).

En particulier, l'environnement de développement **RStudio** propose de nombreuses fonctionnalités qui **rendent l'utilisation de R beaucoup plus simple et intuitive** : explorateur d'environnements, colorisation et auto-complétion du code, afficheur de fenêtres d'aide et de résultats, etc.

À l'ouverture de **RStudio**, en règle générale trois panneaux sont visibles :

- La **console** (à gauche par défaut) : la principale différence avec précédemment tient à la couleur du texte, noire pour les messages et bleue pour le signe `>`. Pour vider l'intégralité de la console, taper `Ctrl + L`.
- L'**explorateur d'environnements et l'historique** (en haut à droite par défaut) : l'explorateur d'environnements permet d'afficher les objets présents (comme la fonction `ls()`) dans l'environnement de référence ; l'historique rappelle toutes les commandes saisies à la manière de la touche ↑ dans la console.
- La **fenêtre de visualisation** (en bas à droite par défaut) : ce panneau intègre à la fenêtre du logiciel l'aide ou encore les graphiques produits.

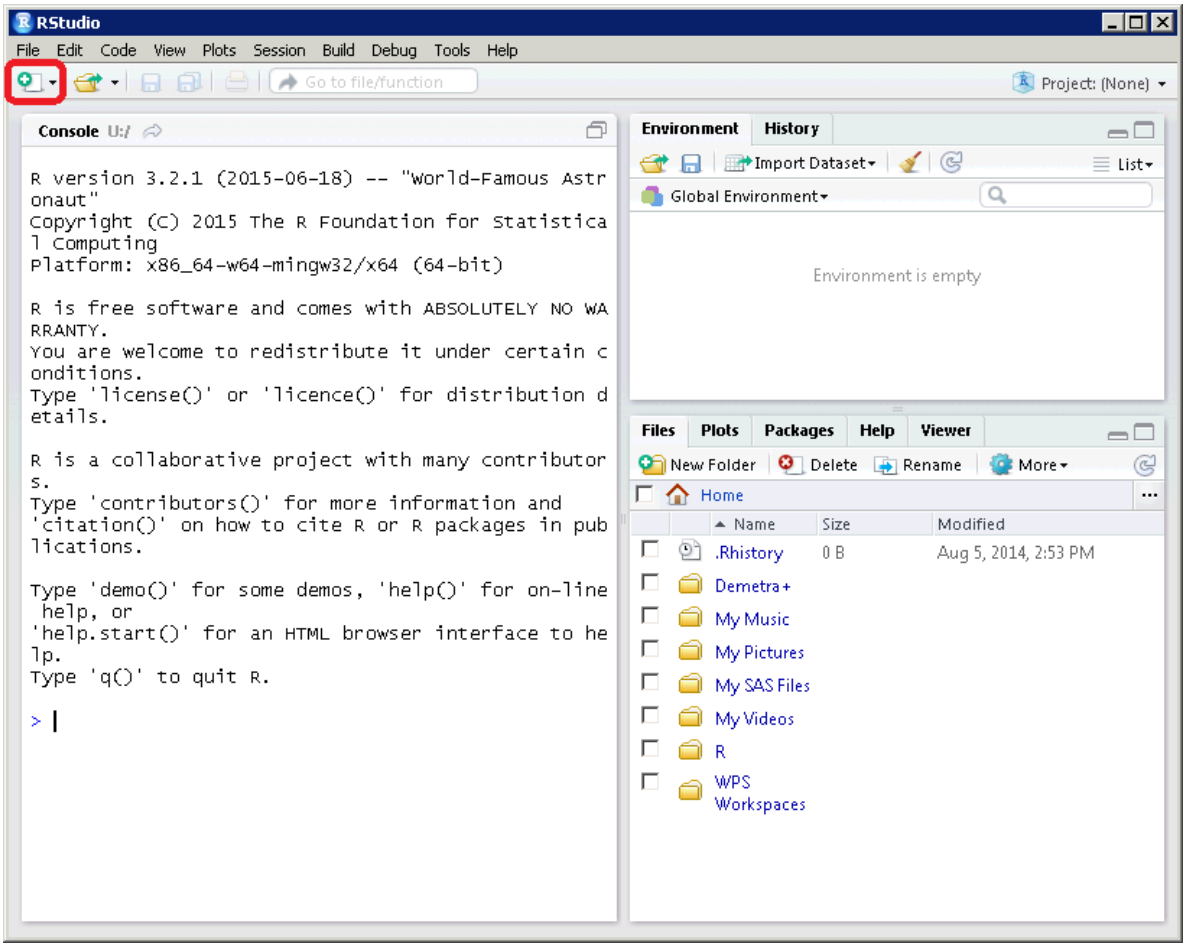


FIGURE 1.4 – Interface fenêtrée de **RStudio** sous Windows

En appuyant sur « Nouveau » > « Script R » (bouton entouré en rouge dans la figure précédente), les fenêtres se réorganisent pour faire apparaître une zone de texte : l'**éditeur de script**.

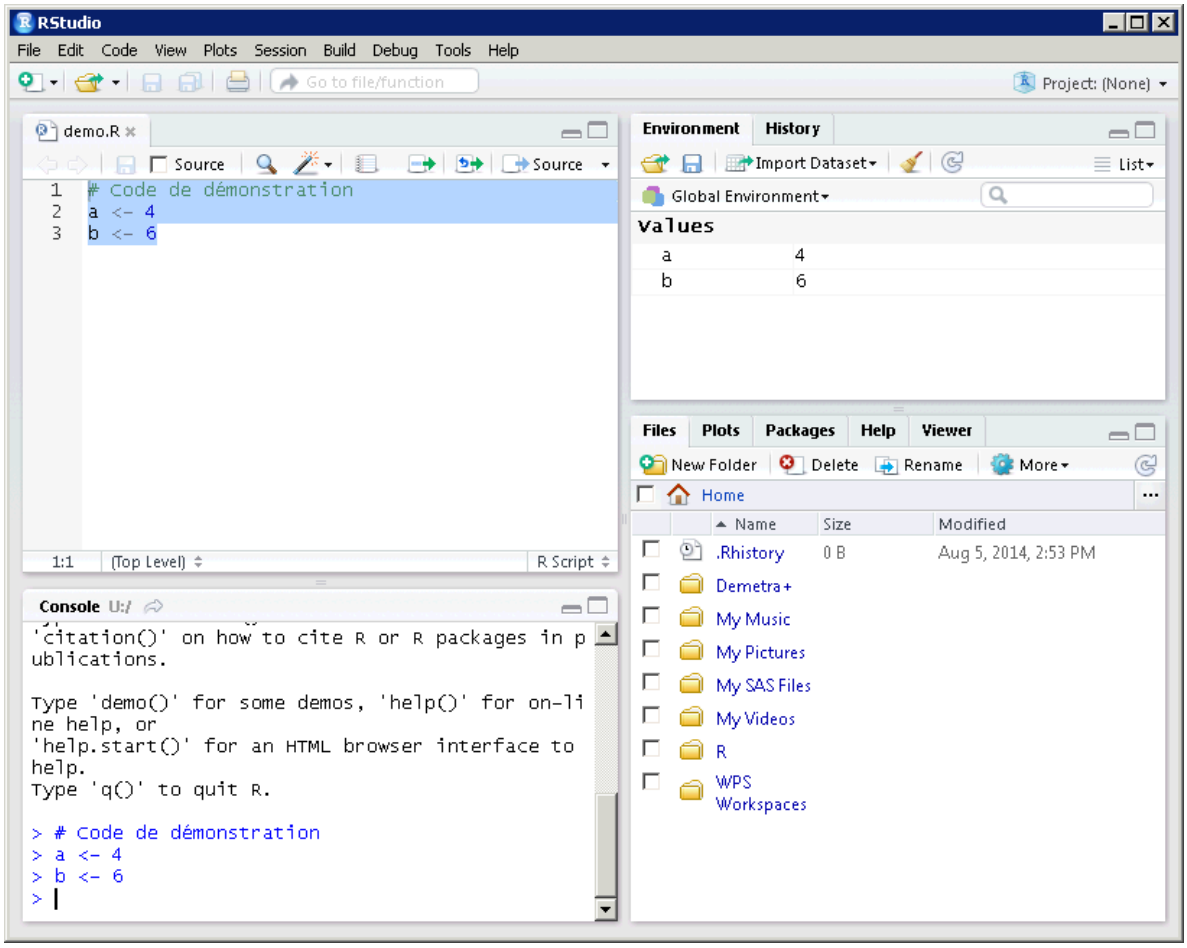


FIGURE 1.5 – Interface fenêtrée de **RStudio** sous Windows – Avec éditeur de scripts ouvert

L'utilisation de l'éditeur de scripts sous **RStudio** est analogue à celle de l'éditeur sous SAS ou du *do-file editor* de Stata :

- il est possible d'**enregistrer** et d'**ouvrir un script** avec les boutons de la barre d'outils correspondants. Le format d'enregistrement par défaut est `.R`, mais le fichier est directement lisible par n'importe quel éditeur de texte (bloc-note ou Notepad++ sous Windows par exemple) ;
- pour **soumettre une ou plusieurs lignes de code**, il suffit de les sélectionner et de saisir au clavier **Ctrl + R** ou **Ctrl + Entrée** ;
- les **éléments de syntaxe apparaissent en couleur** : les commentaires (précédés de `#` à chaque ligne) en vert clair, les objets en noir, les nombres en bleu et les chaînes de caractère (entre `""` ou `''`) en vert foncé. Pour **commenter plusieurs lignes de code simultanément**, il suffit d'utiliser le raccourci **Ctrl + Maj + C** ;

— des **suggestions apparaissent au cours de la frappe** : quand **RStudio** détecte le début du nom d'un objet déjà défini (par exemple une fonction), il fournit des **propositions d'auto-complétion**. Le logiciel double également automatiquement les guillemets et les parenthèses.

Cas pratique 1.3 Construire une fonction de conversion de secondes en minutes-secondes

Ce cas pratique reprend les éléments du cas pratique 1.1. Son objectif est de construire une fonction `conversion()` qui, à partir d'un paramètre `duree` exprimé en secondes, crée une chaîne de caractère du type

```
## [1] "Le traitement a duré `min` minutes et `sec` secondes."
```

a. Toujours sur AUS, ouvrez le programme **RStudio**. Créez un nouveau script et sauvegardez-le sous votre répertoire personnel `U:\` (par exemple sous `U:\R_initiation\module1.R`).

b. En vous inspirant de l'exemple de la fonction `monCalcul()` (*cf. supra*), écrivez dans le script une première version de la fonction `conversion()` qui, à partir du paramètre `duree`, affiche le temps en secondes correspondant (sans le modifier dans un premier temps).

```
# La structure de base d'une définition de fonction est simple :
# l'opérateur d'assignation est utilisé pour associer à un nom
# le code de la fonction
conversion <- function(duree){
  return(duree)
}
# Dans cette première version, on ne fait que renvoyer la valeur
# de duree à l'identique.
conversion(2456)
  ## [1] 2456
```

---

c. Intégrez à la fonction `conversion()` les éléments définis aux différentes étapes du cas pratique 1.1 (définition de `min`, de `sec`, concaténation avec la fonction `paste()`) pour atteindre le résultat désiré. Testez votre fonction avec les valeurs 2456 et 7564.

```
# On reprend les éléments du cas pratique 1.1 pour rendre la fonction
# véritablement opérante :
conversion <- function(duree){
  min <- duree %/% 60
  sec <- duree %% 60
  resultat <- paste(
    "Le traitement a duré", min, "minutes et", sec, "secondes."
  )
  return(resultat)
}
conversion(2456)
  ## [1] "Le traitement a duré 40 minutes et 56 secondes."
conversion(7564)
  ## [1] "Le traitement a duré 126 minutes et 4 secondes."
```

---

d. Observez comment l'éditeur colorise votre code, mais aussi les différents objets créés dans l'explorateur d'environnements. Saisissez dans l'éditeur ou la console les lettres `conver` et utilisez l'auto-complétion pour sélectionner votre fonction. Ajoutez des commentaires (précédés par `#`), manuellement ou en utilisant le raccourci clavier Ctrl + Maj + C.

Charger et explorer des données

Explorer des données statistiques avec R est relativement intuitif, en particulier grâce aux fonctionnalités de RStudio : affichage des objets chargés en mémoire, explorateur d'objets, auto-complétion. **Manipuler des données exige en revanche une plus grande maîtrise des briques élémentaires du langage** qui sont présentées en détails dans le module 2 de la formation.

R travaille sur des **objets stockés en mémoire** : pour explorer des données, la première étape consiste donc à les **charger en mémoire depuis leur emplacement sur le disque dur de l'ordinateur**. On utilise en général pour ce faire la **fonction `load()`** :

```
# Chargement des données du fichier module1.RData
load("U:/R_initiation/donnees/module1.RData")
# NOTE : DANS R LES CHEMINS SONT INDIQUES AVEC DES / ET NON DES \
```

La fonction `load()` charge dans l'environnement de référence les objets contenus dans le fichier `module1.RData` en les décompressant au passage (par défaut les

fichiers saugevardés par R sont compressés). L'environnement de référence comporte désormais deux nouveaux objets :

```
# Fichiers présents dans l'environnement de référence
ls()
  ## [1] "bpe"        "conversion" "rp"

# Caractéristiques de l'objet bpe
str(bpe)
  ## 'data.frame':  358 obs. of  8 variables:
  ##  $ ancreg  : chr  "11" "11" "11" "11" ...
  ##  $ reg     : chr  "11" "11" "11" "11" ...
  ##  $ dep     : chr  "92" "92" "92" "92" ...
  ##  $ depcom  : chr  "92046" "92046" "92046" "92046" ...
  ##  $ dciris  : chr  "92046_0000" "92046_0000" "92046_0000" "92046_0000" ...
  ##  $ an      : chr  "2015" "2015" "2015" "2015" ...
  ##  $ typequ  : chr  "D104" "D109" "F102" "F107" ...
  ##  $ nb_equip: num  1 1 2 1 2 1 2 1 1 1 ...
```

L'objet `bpe` correspond à une extraction de la Base permanente des équipements 2015 restreinte aux équipements de la ville de Malakoff (code Insee 92046). La nomenclature des équipements est présentée sur cette page.

Pour parcourir ce fichier dans **RStudio**, il suffit de **cliquer sur son nom dans l'explorateur d'environnements**. Plusieurs manipulations peuvent par ailleurs être effectuées de façon relativement intuitive :

— **afficher les premières lignes** avec la fonction `head()`, les **dernières lignes** avec la fonction `tail()` :

```
# Affichage des premières et dernières lignes de l'objet bpe
head(bpe)
  ##   ancreg reg dep depcom     dciris   an typequ nb_equip
  ## 1     11  11  92  92046 92046_0000 2015   D104        1
  ## 2     11  11  92  92046 92046_0000 2015   D109        1
  ## 3     11  11  92  92046 92046_0000 2015   F102        2
  ## 4     11  11  92  92046 92046_0000 2015   F107        1
  ## 5     11  11  92  92046 92046_0000 2015   F111        2
  ## 6     11  11  92  92046 92046_0000 2015   F112        1
tail(bpe)
  ##     ancreg reg dep depcom     dciris   an typequ nb_equip
  ## 353     11  11  92  92046 92046_0111 2015   D233        1
  ## 354     11  11  92  92046 92046_0111 2015   D235        1
  ## 355     11  11  92  92046 92046_0111 2015   D301        1
  ## 356     11  11  92  92046 92046_0111 2015   D501        2
  ## 357     11  11  92  92046 92046_0111 2015   E101        5
```

```
## 358     11  11  92  92046 92046_0111 2015   F121        1
```

— **accéder au contenu d'une variable avec l'opérateur $** (ici uniquement les 20 premières valeurs pour des raisons de présentation) :

```
# Affichage des premières de la variable codant le type d'équipement
bpe$typequ
```

```
##  [1] "D104" "D109" "F102" "F107" "F111" "F112" "F113" "F120"
##  [9] "F121" "F303" "A301" "A401" "A402" "A403" "A404" "A504"
## [17] "A507" "B101" "B304" "B306"
```

— **calculer le total et des statistiques descriptives** sur une variable de nature quantitative avec les fonctions `sum()` et `summary()` :

```
# Total et distribution de la variable dénombrant le nombre d'équipements
# par iris et par type
sum(bpe$nb_equip)
  ## [1] 867
summary(bpe$nb_equip)
  ##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  ##   1.000   1.000   1.000   2.422   3.000  33.000
```

— **déterminer les modalités distinctes et tabuler** une variable de nature qualitative avec les fonctions `unique()` et `table()` :

```
# Liste des iris associés à la commune de Malakoff
unique(bpe$dciris)
  ##  [1] "92046_0000" "92046_0101" "92046_0102" "92046_0103"
  ##  [5] "92046_0104" "92046_0105" "92046_0106" "92046_0107"
  ##  [9] "92046_0108" "92046_0109" "92046_0110" "92046_0111"

# Nombre de types d'équipements distincts par iris à Malakoff
table(bpe$dciris)
  ##
  ## 92046_0000 92046_0101 92046_0102 92046_0103 92046_0104
  ##         10         20         46         35         27
  ## 92046_0105 92046_0106 92046_0107 92046_0108 92046_0109
  ##         61         35         37         29          9
  ## 92046_0110 92046_0111
  ##         29         20
```

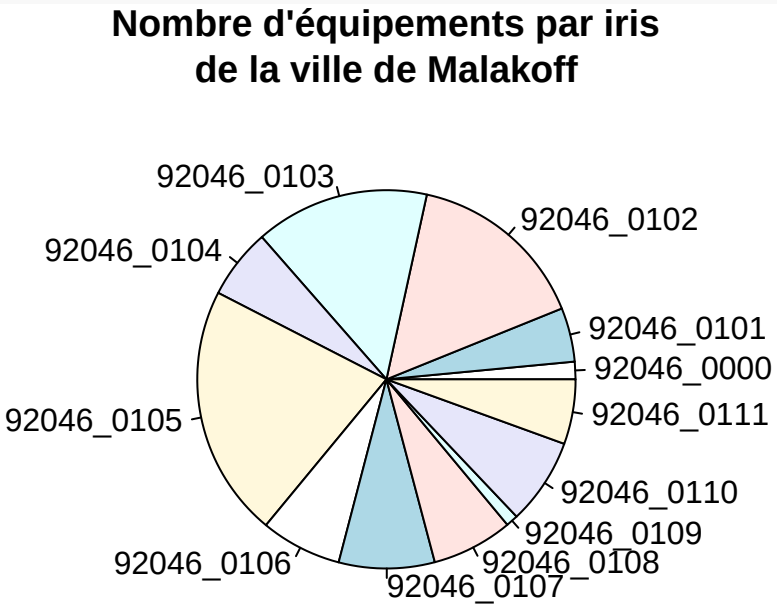
— **appliquer une fonction selon les modalités d'une autre variable** avec la fonction `by()` :

```
# Nombre total d'équipements par iris
by(bpe$nb_equip, bpe$dciris, sum)
  ## bpe$dciris: 92046_0000
  ## [1] 13
  ## ------------------------------------------------
  ## bpe$dciris: 92046_0101
  ## [1] 40
  ## ------------------------------------------------
  ## bpe$dciris: 92046_0102
  ## [1] 134
  ## ------------------------------------------------
  ## bpe$dciris: 92046_0103
  ## [1] 129
  ## ------------------------------------------------
  ## bpe$dciris: 92046_0104
  ## [1] 52
  ## ------------------------------------------------
  ## bpe$dciris: 92046_0105
  ## [1] 187
  ## ------------------------------------------------
  ## bpe$dciris: 92046_0106
  ## [1] 60
  ## ------------------------------------------------
  ## bpe$dciris: 92046_0107
  ## [1] 71
  ## ------------------------------------------------
  ## bpe$dciris: 92046_0108
  ## [1] 60
  ## ------------------------------------------------
  ## bpe$dciris: 92046_0109
  ## [1] 9
  ## ------------------------------------------------
  ## bpe$dciris: 92046_0110
  ## [1] 64
  ## ------------------------------------------------
  ## bpe$dciris: 92046_0111
  ## [1] 48
```

— **faire des représentation graphiques simples** avec les fonctions `pie()`, `barplot()` et `plot()` :

```
# Représentation du nombre total d'équipements par iris
pie(
  by(bpe$nb_equip,bpe$dciris,sum)
```

```
  , main = "Nombre d'équipements par iris\nde la ville de Malakoff"
)
```



Remarque D'un point de vue technique, **l'objet `bpe` est de type `data.frame`**, qui correspond au format le plus fréquent pour les tableaux de données dans R. Ce type d'objet est **relativement complexe et est présenté en détails dans le module 3 de la formation**.

Cas pratique 1.4 Charger et explorer des données : Le recensement de la population 2013 dans les Hauts-de-Seine

Ce cas pratique vise à charger et à effectuer **quelques manipulations simples sur une extraction du fichier du recensement de la population (RP) 2013 dans les Hauts-de-Seine** (accessible sur le site de l'Insee). Les données ont été préalablement téléchargées et converties (*cf.* sous-partie suivante) et sont contenues dans le fichier `module1.RData`.

a. L'ensemble des données de la formation sont contenues dans le fichier donnees.zip. Téléchargez ce fichier, copiez-collez puis décompressez-le sous AUS dans le répertoire `U:\R_initiation\donnees`.

b. Utilisez la fonction `load()` pour charger les données contenues dans le fichier `U:\R_initiation\donnees\module1.RData`. **Pensez à bien utiliser des / et non des \ dans le chemin du fichier** (sans quoi le chargement ne fonctionnera

pas). Affichez les caractéristiques de l'objet `rp` : combien ce fichier comporte-t-il d'observations et de variables ? Affichez ses premières lignes.

```
# Chargement du fichier exemples.RData
load("U:/R_initiation/donnees/module1.RData")
# NOTE : DANS R LES CHEMINS SONT INDIQUES AVEC DES / ET NON DES \
```

```
# Objets présents dans l'environnement de travail
ls()
  ## [1] "bpe"        "conversion" "rp"

# Caractéristiques de l'objet rp
str(rp)
  ## 'data.frame':  609446 obs. of  6 variables:
  ##  $ DEPT  : int  92 92 92 92 92 92 92 92 92 92 ...
  ##  $ IPONDI: num  3.49 3.49 3.49 1.1 1.1 ...
  ##  $ ANAI  : int  1960 1938 1936 2008 2004 1972 1951 1978 1976 1993 ...
  ##  $ SEXE  : int  1 2 1 2 2 2 1 2 1 2 ...
  ##  $ STOCD : chr  "10" "10" "10" "10" ...
  ##  $ CS1   : int  8 7 7 8 8 8 3 3 3 8 ...
# Le fichier rp comporte 609 446 observations et 6 variables

# Pour afficher les premières lignes d'une table, on utilise
# la fonction head()
head(rp)
  ##   DEPT   IPONDI ANAI SEXE STOCD CS1
  ## 1   92 3.492757 1960    1    10   8
  ## 2   92 3.492757 1938    2    10   7
  ## 3   92 3.492757 1936    1    10   7
  ## 4   92 1.101669 2008    2    10   8
  ## 5   92 1.101669 2004    2    10   8
  ## 6   92 1.101669 1972    2    10   8
```

---

c. Utilisez l'opérateur `$` pour afficher les valeurs de la variable de pondération `IPONDI`. Pensez à bien respecter la casse du nom de la variable. Appliquez la fonction `sum()` à la variable `IPONDI` pour déterminer la population totale des Hauts-de-Seine au sens du RP 2013 (*i.e.* calculer la somme de la variable `IPONDI`).

```
# Affichage du contenu de la variable IPONDI
rp$IPONDI
# Note : pour des raisons de présentation, seules les 20 premières valeurs
# sont affichées ici.
```

```
##  [1] 3.4927574 3.4927574 3.4927574 1.1016688 1.1016688 1.1016688
##  [7] 1.1016688 3.6388523 3.6388523 0.8573350 0.8573350 3.7741295
## [13] 3.7741295 3.5340019 1.0162282 1.0343014 1.0343014 0.8821233
## [19] 0.8821233 3.3128186
```

```
sum(rp$IPONDI)
  ## [1] 1591365
# La population des Hauts-de-Seine au sens du RP 2013 est de
# 1 591 365 habitants.
```

---

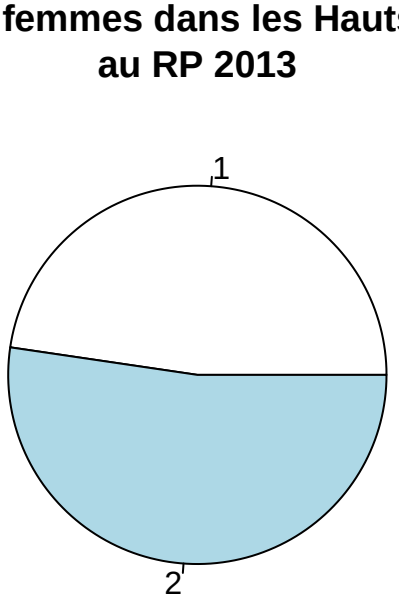
d. Affichez les modalités distinctes de la variable `SEXE`. Appliquez la fonction `table()` à cette variable pour déterminer la répartition par sexe des individus recensés. Combinez les fonctions `by()` et `sum()` pour calculer la somme de la variable `IPONDI` selon les modalités de la variable `SEXE`.

```
# Pour afficher les modalités distinctes d'une variable, on utilise
# la fonction unique()
unique(rp$SEXE)
  ## [1] 1 2
# Comme souvent, le sexe est codé par un chiffre, "1" pour
# les hommes et "2" pour les femmes.
table(rp$SEXE)
  ##
  ##      1      2
  ## 290444 319002
# La fonction table() permet d'effectuer une tabulation
# simple (et non pondérée) d'une variable

# Pour obtenir la somme des poids par sexe, on peut recourir
# à la fonction by() combinée à la fonction sum()
by(rp$IPONDI, rp$SEXE, sum)
  ## rp$SEXE: 1
  ## [1] 758656.9
  ## ------------------------------------------------
  ## rp$SEXE: 2
  ## [1] 832707.8
# Au RP 2013, le département des Hauts-de-Seine comptait 758 657 hommes
# et 832 708 femmes.
```

---

e. (Optionnel) Utilisez les résultats des deux questions précédentes pour calculer le pourcentage d'hommes et de femmes dans les Hauts-de-Seine au sens du RP 2013. Représentez ces données avec un diagramme circulaire.

```
# Plusieurs méthodes existent pour calculer un pourcentage
# à partir d'un objet de table(). La plus simple est
# de diviser par la taille totale de la population
# calculée à la question b.
by(rp$IPONDI, rp$SEXE, sum)/sum(rp$IPONDI)
  ## rp$SEXE: 1
  ## [1] 0.4767335
  ## ------------------------------------------------
  ## rp$SEXE: 2
  ## [1] 0.5232665

# Pour représenter ces données avec un diagramme
# circulaire, on utilise la fonction pie()
pie(
  by(rp$IPONDI, rp$SEXE, sum)/sum(rp$IPONDI)
  , main = "Hommes et femmes dans les Hauts-de-Seine\nau RP 2013"
)
```



Importer des données à l'aide de *packages*

En règle générale, les fichiers de données sur lesquels on souhaite travailler ne sont pas en format R natif : il convient donc de les **importer**. **R dispose de très nombreuses fonctions pour importer des données provenant d'autres logiciels** : SAS, Stata, Excel, etc. Toutes ne sont cependant pas chargées par défaut au démarrage du logiciel, mais sont facilement accessibles par le biais de *packages*.

Le « fil rouge » de cette sous-partie est l'**importation d'autres données de la Base permanente des équipements** (relatives à Montrouge, code Insee 92049) **et stockées dans différents formats** (`bpe2.csv`, `bpe2.dbf`, `bpe2.sas7bdat`). **L'utilisation des *packages*, leur chargement et leur installation sont présentés en parallèle**.

Pour faciliter l'import de fichiers différents, on modifie le répertoire de travail (*working directory*) **de R** : il s'agit du répertoire dans lequel le logiciel **recherche par défaut les fichiers à importer**. Une fois le répertoire de travail convenablement défini (avec la fonction `setwd()`), il suffit de saisir le nom du fichier à importer pour que R le trouve automatiquement :

```
# Définition du répertoire de la formation comme répertoire de travail
setwd("U:/R_initiation/donnees")

# Utilisation de la fonction load() sans avoir à indiquer un chemin
load("module1.RData")
```

Importer des fichiers plats avec `read.table()`

R dispose nativement d'un fonction capable de lire les fichiers dits « plats » (`.txt`, `.csv` ou `.dlm` le plus souvent) : la **fonction `read.table()`** (taper `? read.table` pour afficher sa page d'aide). Cette fonction comporte un grand nombre de paramètres susceptibles d'être ajustés au format du fichier en entrée : délimiteur, séparateur de décimales, etc.

Afin de faciliter l'utilisation de cette fonction, des fonctions « alias » sont également disponibles qui correspondent à des **versions pré-paramétrées de `read.table()`**. En particulier :

- `read.csv()` importe des fichiers dont les colonnes sont **séparées par des virgules** ;
- `read.delim()` importe des fichiers dont les colonnes sont **séparées par des tabulations**.

Les colonnes du fichier `bpe2.csv` utilisé dans cet exemple sont **séparées par des virgules** (comme ceux des fichiers produits par défaut par LibreOffice Calc) : c'est donc la fonction `read.csv()` qu'il convient d'utiliser :

```
# Chargement du fichier bpe2.csv
bpe2_csv <- read.csv("bpe2.csv")
```

```
# Premières lignes de bpe2_csv
head(bpe2_csv)
  ##   ancreg reg dep depcom     dciris   an typequ nb_equip
  ## 1     11  11  92  92049 92049_0000 2015   B301        1
  ## 2     11  11  92  92049 92049_0000 2015   F101        1
  ## 3     11  11  92  92049 92049_0000 2015   F112        2
  ## 4     11  11  92  92049 92049_0000 2015   F114        1
  ## 5     11  11  92  92049 92049_0000 2015   F120        1
  ## 6     11  11  92  92049 92049_0000 2015   F121        4
```

Importer des fichiers `.dbf` ou `.dta` avec le *package* `foreign`

Au-delà des fonctions natives de R, plusieurs *packages* permettent d'importer facilement des données dans R, dont le *package* `foreign`. Dans **RStudio**, la sous-fenêtre *Packages* de la fenêtre de visualisation (en bas à droite par défaut) permet d'afficher l'ensemble des *packages* installés avec une description succincte.

Le *package* `foreign` a la particularité d'être **pré-installé** : pour utiliser ses fonctions, il suffit de le charger avec la fonction `library()` (une fois par session suffit).

```
# Chargement du package foreign
library("foreign")
```

Dans **RStudio**, cocher la case devant le nom du *package* génère automatiquement une ligne de code équivalente.

Dès lors qu'il est chargé, les fonctions d'import de données du *package* `foreign` sont accessibles, par exemple depuis un fichier au format `.dbf` (la plupart des fichiers « détails » mis en ligne sur le site de l'Insee sont des `.dbf` zippés) :

```
# Chargement du fichier bpe2.dbf
bpe2_dbf <- read.dbf("bpe2.dbf")

# Premières lignes de bpe2_dbf
head(bpe2_dbf)
  ##   ancreg reg dep depcom     dciris   an typequ nb_equip
  ## 1     11  11  92  92049 92049_0000 2015   B301        1
  ## 2     11  11  92  92049 92049_0000 2015   F101        1
  ## 3     11  11  92  92049 92049_0000 2015   F112        2
  ## 4     11  11  92  92049 92049_0000 2015   F114        1
  ## 5     11  11  92  92049 92049_0000 2015   F120        1
  ## 6     11  11  92  92049 92049_0000 2015   F121        4
```

Le package `foreign` permet également d'importer des fichiers `.dta` (fichiers de données Stata, version 5-12), mais aussi d'exporter des fichiers `.dbf` et `.dta` avec les fonctions

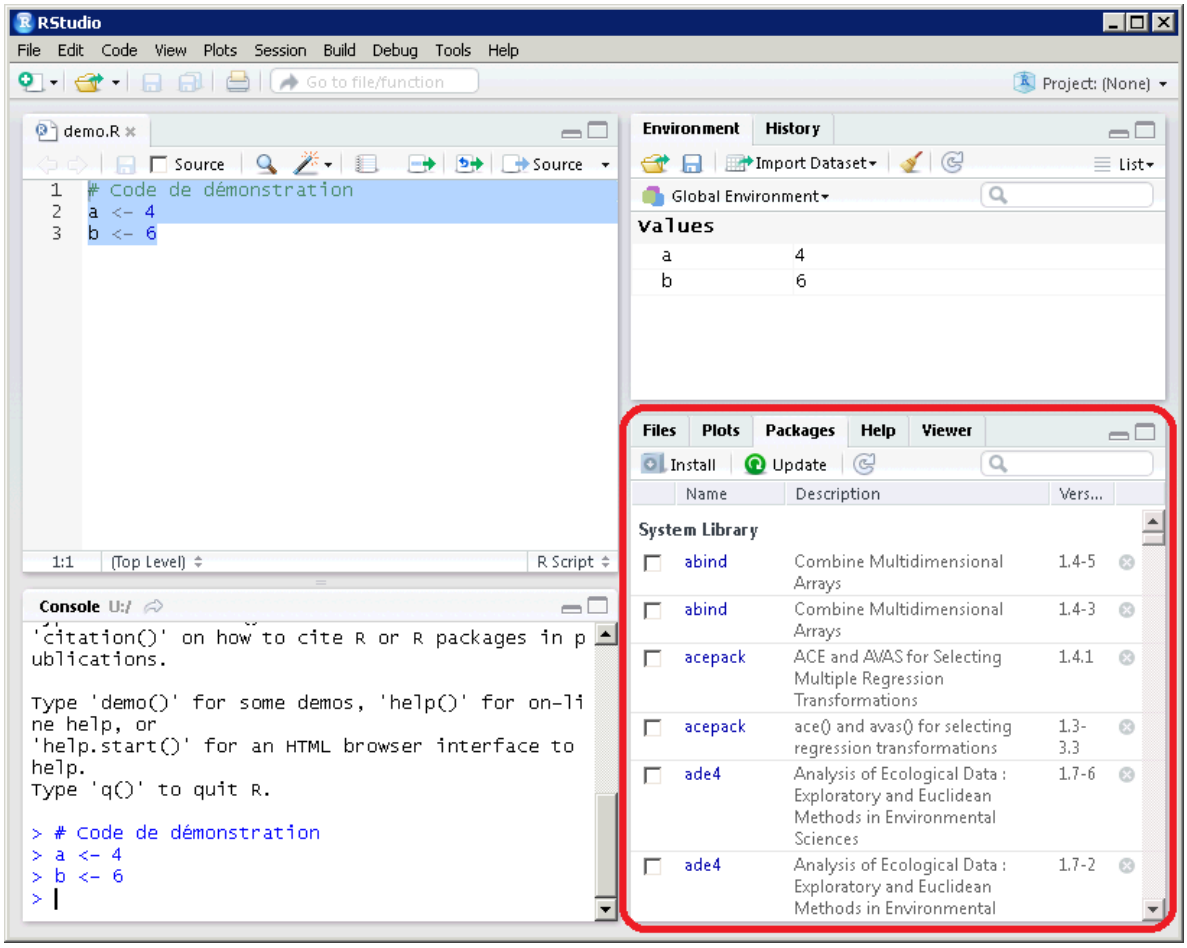


FIGURE 1.6 – Interface fenêtrée de **RStudio** sous Windows – Liste des *packages* installés

`write.dbf()` et `write.dta()` respectivement :

```
# Export du fichier bpe2_dbf en .dta
write.dta(bpe2_dbf, file = "bpe2.dta")
# Note : par défaut les fichiers produits par un code R sont
# créés dans le répertoire de travail, ici U:\R_initiation\donnees.
```

Importer des fichiers `.sas7bdat` avec le *package* `haven`

Aucune fonction native ou package pré-installé de R ne permet d'**importer des données au format SAS `.sas7bdat`**. Pour ce faire, **la meilleure solution consiste à installer et à utiliser le *package* `haven`**.

Dans R l'installation de *packages* est effectuée *via* la fonction `install.packages()` :

```
# Installation du package haven
install.packages("haven")
```

En règle générale **une fenêtre apparaît pour demander de choisir un « miroir » pour le téléchargement des fichiers**. Comme pour la plupart des logiciels libres, les éléments constitutifs de R ne sont pas disponibles sur un seul serveur mais sur une multitude de serveurs identiques (d'où le nom « miroir »), en général maintenus par des universités ou des institutions de recherche. N'importe quel « miroir » peut donc faire l'affaire, mais il est courant de privilégier le serveur le plus proche géographiquement (plusieurs miroirs sont situés à Paris).

Si nécessaire, le programme télécharge et installe également, en plus du *package* demandé, l'ensemble des **dépendances** indispensables à son fonctionnement. **Il est en effet fréquent qu'un *package* s'appuie sur des fonctionnalités proposées par d'autres *packages* non pré-installés par défaut**. Pour connaître la liste des dépendances d'un *package*, il suffit de consulter les rubriques *Depends* et *Imports* de sa page de référence sur le *Comprehensive R Archive Network* (CRAN). Par exemple pour le *package* `haven` : https://CRAN.R-project.org/package=haven

---

Note Afin de pouvoir facilement installer de nouveaux packages **sur AUS** (sur lequel les utilisateurs n'ont pas accès à internet), **un dépôt local de *packages* a été mis en place et est sélectionné par défaut**. Très spécifiquement sur AUS, **le *package* `haven` figure dans le lot de packages pré-installés** : il n'a donc pas à être installé par chaque utilisateur.

Une fois le *package* `haven` installé, il suffit de le charger avec la fonction `library()` puis d'utiliser la fonction `read_sas()` pour importer des données au format `.sas7bdat`.

```
# Chargement du package haven
library("haven")

# Chargement du fichier bpe2.sas7bdat
bpe2_sas <- read_sas("bpe2.sas7bdat")

# Premières lignes de bpe2_sas
head(bpe2_sas)
  ## # A tibble: 6 x 8
  ##   ancreg   reg   dep depcom     dciris    an typequ nb_equip
  ##    <chr> <chr> <chr>  <chr>      <chr> <chr>  <chr>    <dbl>
  ## 1     11    11    92  92049 92049_0000  2015   B301        1
  ## 2     11    11    92  92049 92049_0000  2015   F101        1
  ## 3     11    11    92  92049 92049_0000  2015   F112        2
  ## 4     11    11    92  92049 92049_0000  2015   F114        1
  ## 5     11    11    92  92049 92049_0000  2015   F120        1
  ## 6     11    11    92  92049 92049_0000  2015   F121        4
```

Cas pratique 1.5 Importer des données

Les **données du Code officiel géographique** (COG) sont diffusées sur le site de l'Insee en plusieurs formats (`.txt` ou `.dbf` zippés). Ce cas pratique vise à importer ces données dans R, et le cas pratique suivant à les sauvegarder en format R natif.

a. On cherche à importer le fichier `depts2017.txt`. Il s'agit d'un fichier dont les colonnes sont séparées par des tabulations `\t` : quelle fonction semble adaptée selon vous pour importer ce fichier ? Utilisez-la pour lire ce fichier dans l'objet `dep`. Affichez-en les caractéristiques ainsi que les premières lignes.

```
# Les colonnes du fichier étant séparées par des tabulations,
# c'est la fonction read.delim() qui est la plus adaptée.
dep <- read.delim("depts2017.txt")
str(dep)
  ## 'data.frame':  101 obs. of  6 variables:
  ##  $ REGION  : int  84 32 84 93 93 93 84 44 76 44 ...
  ##  $ DEP     : chr  "01" "02" "03" "04" ...
  ##  $ CHEFLIEU: chr  "01053" "02408" "03190" "04070" ...
  ##  $ TNCC    : int  5 5 5 4 4 4 5 4 5 5 ...
  ##  $ NCC     : chr  "AIN" "AISNE" "ALLIER" "ALPES-DE-HAUTE-PROVENCE" ...
  ##  $ NCCENR  : chr  "Ain" "Aisne" "Allier" "Alpes-de-Haute-Provence" ...
head(dep)
```

```
##   REGION DEP CHEFLIEU TNCC                     NCC
## 1     84  01    01053    5                     AIN
## 2     32  02    02408    5                   AISNE
## 3     84  03    03190    5                  ALLIER
## 4     93  04    04070    4 ALPES-DE-HAUTE-PROVENCE
## 5     93  05    05061    4            HAUTES-ALPES
## 6     93  06    06088    4         ALPES-MARITIMES
##                    NCCENR
## 1                     Ain
## 2                   Aisne
## 3                  Allier
## 4 Alpes-de-Haute-Provence
## 5             Hautes-Alpes
## 6          Alpes-Maritimes
```

---

b. Les fichiers du COG sont également disponibles sous forme de fichiers `.dbf` zippés. Le fichier `comsimp2017.dbf` correspond ainsi à la liste des communes à géographie 2017. Chargez le *package* `foreign` et utilisez la fonction `read.dbf()` pour importer ce fichier dans l'objet `com`. Affichez-en les caractéristiques et les premières lignes.

```
# Chargement du package foreign
library(foreign)

# Utilisation de la fonction read.dbf()
com <- read.dbf("comsimp2017.dbf")
str(com)
  ## 'data.frame':  35416 obs. of  12 variables:
  ##  $ CDC     : Factor w/ 2 levels "0","2": 1 1 1 1 1 1 1 1 1 1 ...
  ##  $ CHEFLIEU: Factor w/ 5 levels "0","1","2","3",..: 1 1 2 1 1 1 1 1 1 1 .
  ##  $ REG     : Factor w/ 18 levels "01","02","03",..: 16 16 16 16 16 16 16
  ##  $ DEP     : Factor w/ 101 levels "01","02","03",..: 1 1 1 1 1 1 1 1 1 1
  ##  $ COM     : Factor w/ 950 levels "001","002","003",..: 1 2 4 5 6 7 8 9 1
  ##  $ AR      : Factor w/ 9 levels "1","2","3","4",..: 2 1 1 2 1 1 1 1 1 4 .
  ##  $ CT      : Factor w/ 55 levels "01","02","03",..: 8 1 1 22 4 1 1 4 10 1
  ##  $ TNCC    : Factor w/ 8 levels "0","1","2","3",..: 6 6 2 2 2 2 2 2 2 2 .
  ##  $ ARTMAJ  : Factor w/ 6 levels "(AUX)","(L')",..: 2 2 NA NA NA NA NA NA
  ##  $ NCC     : Factor w/ 32825 levels "AAST","ABAINVILLE",..: 19 21 456 458
  ##  $ ARTMIN  : Factor w/ 6 levels "(Aux)","(L')",..: 2 2 NA NA NA NA NA NA
  ##  $ NCCENR  : Factor w/ 32895 levels "Aast","Abainville",..: 19 21 486 488
  ##  - attr(*, "data_types")= chr  "C" "C" "C" "C" ...
```

```
head(com)
  ##   CDC CHEFLIEU REG DEP COM AR CT TNCC ARTMAJ
  ## 1   0        0  84  01 001  2 08    5   (L')
  ## 2   0        0  84  01 002  1 01    5   (L')
  ## 3   0        1  84  01 004  1 01    1   <NA>
  ## 4   0        0  84  01 005  2 22    1   <NA>
  ## 5   0        0  84  01 006  1 04    1   <NA>
  ## 6   0        0  84  01 007  1 01    1   <NA>
  ##                     NCC ARTMIN                    NCCENR
  ## 1 ABERGEMENT-CLEMENCIAT   (L') Abergement-Cl\xe9menciat
  ## 2   ABERGEMENT-DE-VAREY   (L')      Abergement-de-Varey
  ## 3     AMBERIEU-EN-BUGEY   <NA>     Amb\xe9rieu-en-Bugey
  ## 4   AMBERIEUX-EN-DOMBES   <NA>   Amb\xe9rieux-en-Dombes
  ## 5               AMBLEON   <NA>               Ambl\xe9on
  ## 6              AMBRONAY   <NA>                 Ambronay
```

---

c. Le fichier `arrond2017.sas7bdat` correspond à la table des arrondissements convertie au format `.sas7bdat`. Utilisez le *package* `haven` pour importer ce fichier dans l'objet `arrond`. Affichez-en les caractéristiques et les premières lignes.

```
# Chargement du package haven (pré-installé sur AUS)
library(haven)

# Remarque : si haven n'avait pas été pré-installé, il aurait
# fallu l'installer avec
# install.packages("haven")

# Utilisation de la fonction read_sas()
arrond <- read_sas("arrond2017.sas7bdat")
str(arrond)
  ## Classes 'tbl_df', 'tbl' and 'data.frame':  333 obs. of  9 variables:
  ##  $ REGION  : atomic  84 84 84 84 ...
  ##   ..- attr(*, "label")= chr "REGION"
  ##   ..- attr(*, "format.sas")= chr "$"
  ##  $ DEP     : atomic  01 01 01 01 ...
  ##   ..- attr(*, "label")= chr "DEP"
  ##   ..- attr(*, "format.sas")= chr "$"
  ##  $ AR      : atomic  1 2 3 4 ...
  ##   ..- attr(*, "label")= chr "AR"
  ##   ..- attr(*, "format.sas")= chr "$"
  ##  $ CHEFLIEU: atomic  01034 01053 01173 01269 ...
```

```
##   ..- attr(*, "label")= chr "CHEFLIEU"
##   ..- attr(*, "format.sas")= chr "$"
##  $ TNCC    : atomic  0 0 0 0 ...
##   ..- attr(*, "label")= chr "TNCC"
##   ..- attr(*, "format.sas")= chr "$"
##  $ ARTMAJ  : atomic      ...
##   ..- attr(*, "label")= chr "ARTMAJ"
##   ..- attr(*, "format.sas")= chr "$"
##  $ NCC     : atomic  BELLEY BOURG-EN-BRESSE GEX NANTUA ...
##   ..- attr(*, "label")= chr "NCC"
##   ..- attr(*, "format.sas")= chr "$"
##  $ ARTMIN  : atomic      ...
##   ..- attr(*, "label")= chr "ARTMIN"
##   ..- attr(*, "format.sas")= chr "$"
##  $ NCCENR  : atomic  Belley Bourg-en-Bresse Gex Nantua ...
##   ..- attr(*, "label")= chr "NCCENR"
##   ..- attr(*, "format.sas")= chr "$"
##  - attr(*, "label")= chr "ARROND2017"
head(arrond)
## # A tibble: 6 x 9
##   REGION   DEP    AR CHEFLIEU  TNCC ARTMAJ             NCC
##    <chr> <chr> <chr>    <chr> <chr>  <chr>           <chr>
## 1     84    01     1    01034     0                 BELLEY
## 2     84    01     2    01053     0        BOURG-EN-BRESSE
## 3     84    01     3    01173     0                    GEX
## 4     84    01     4    01269     0                 NANTUA
## 5     32    02     1    02168     0        CHATEAU-THIERRY
## 6     32    02     2    02408     0                   LAON
## # ... with 2 more variables: ARTMIN <chr>, NCCENR <chr>
```

Sauvegarder des données en format R natif

Une fois des données importées, il est souvent utile de les **sauvegarder sur le disque dur dans un format susceptible d'être lu rapidement par R**. Deux fonctions sont particulièrement utiles dans ce contexte :

— `save()` : la fonction `save()` est **le pendant de la fonction `load()` utilisée dans la sous-partie précédente**. Elle permet de **sauvegarder un ou plusieurs fichiers** que la fonction `load()` recharge tels quels (en particulier avec le même nom) dans l'environnement de référence :

```
# Sauvegarde de tous les fichiers importés dans le fichier bpe2.RData
save(bpe2_csv, bpe2_dbf, bpe2_sas, file = "bpe2.RData")

# Suppression des fichiers bpe2_csv, bpe2_dbf et bpe2_sas
rm(bpe2_csv, bpe2_dbf, bpe2_sas)
ls()
  ## [1] "arrond"     "bpe"        "com"        "conversion"
  ## [5] "dep"        "rp"

# Chargement du fichier bpe2.RData
load("bpe2.RData")
ls()
  ## [1] "arrond"     "bpe"        "bpe2_csv"   "bpe2_dbf"
  ## [5] "bpe2_sas"   "com"        "conversion" "dep"
  ## [9] "rp"
```

En particulier, **quand un objet qui est déjà présent dans l'environnement de référence a le même nom qu'un objet rechargé avec `load()`, il est écrasé**.

```
# Redéfinition de l'objet bpe2_csv
bpe2_csv <- "Mon nouvel objet bpe2_csv"
str(bpe2_csv)
  ##  chr "Mon nouvel objet bpe2_csv"

# Chargement du fichier bpe2.RData
load("bpe2.RData")
str(bpe2_csv)
  ## 'data.frame':  544 obs. of  8 variables:
  ##  $ ancreg  : int  11 11 11 11 11 11 11 11 11 11 ...
  ##  $ reg     : int  11 11 11 11 11 11 11 11 11 11 ...
  ##  $ dep     : int  92 92 92 92 92 92 92 92 92 92 ...
  ##  $ depcom  : int  92049 92049 92049 92049 92049 92049 92049 92049 92049 92049
  ##  $ dciris  : chr  "92049_0000" "92049_0000" "92049_0000" "92049_0000" ...
  ##  $ an      : int  2015 2015 2015 2015 2015 2015 2015 2015 2015 2015 ...
  ##  $ typequ  : chr  "B301" "F101" "F112" "F114" ...
  ##  $ nb_equip: int  1 1 2 1 1 4 1 3 1 1 ...
```

— `saveRDS()` : **la fonction `saveRDS()` permet de créer des fichiers `.rds` stockant chacun un seul et unique objet en format R natif**. La **fonction `readRDS()`** permet de les recharger et d'affecter leur valeur à un objet de son choix :

```
# Sauvegarde de l'objet bpe2_csv en .rds
saveRDS(bpe2_csv, file = "bpe2_csv.rds")
```

```
# Chargement du fichier bpe2_csv.rds dans l'objet bpe3
bpe3 <- readRDS("bpe2_csv.rds")
str(bpe3)
  ## 'data.frame':  544 obs. of  8 variables:
  ##  $ ancreg  : int  11 11 11 11 11 11 11 11 11 11 ...
  ##  $ reg     : int  11 11 11 11 11 11 11 11 11 11 ...
  ##  $ dep     : int  92 92 92 92 92 92 92 92 92 92 ...
  ##  $ depcom  : int  92049 92049 92049 92049 92049 92049 92049 92049 92049 9
  ##  $ dciris  : chr  "92049_0000" "92049_0000" "92049_0000" "92049_0000" ...
  ##  $ an      : int  2015 2015 2015 2015 2015 2015 2015 2015 2015 2015 ...
  ##  $ typequ  : chr  "B301" "F101" "F112" "F114" ...
  ##  $ nb_equip: int  1 1 2 1 1 4 1 3 1 1 ...

# Comparaison de bpe2_csv et de bpe3
identical(bpe2_csv,bpe3)
  ## [1] TRUE
```

Remarque Quoique moins connues, on recommande souvent (par exemple ici) de **privilégier les fonctions `saveRDS()`/`readRDS()` à `save()`/`load()`**, ne serait-ce que pour éviter les **conflits de noms** et les écrasements inintentionnels de données qui en résultent.

Cas pratique 1.6 Sauvegarder des données

a. Sauvegardez les objets `dep`, `com` et `arrond` créés dans le cas pratique précédent dans le fichier `cog.RData` à l'aide de la fonction `save()`. Vérifiez que le fichier est bien créé dans le répertoire de travail que vous avez indiqué.

```
# Sauvegarde des objets du COG dans cog.RData
save(dep, com, arrond, file = "cog.RData")
```

---

b. Supprimez l'ensemble des objets de l'environnement de référence puis rechargez les fichiers du COG en utilisant la fonction `load()` sur `cog.RData`. Vérifiez que les objets concernés sont bien de nouveau présent dans l'environnement de travail.

```r
# Suppression de tous les fichiers de l'environnement de référence
rm(list = ls())
ls()
  ## character(0)

# Chargement des fichiers du COG
load("cog.RData")
ls()
  ## [1] "arrond" "com"    "dep"
```

---

c. Utilisez la fonction `saveRDS()` pour sauvegarder l'objet `dep` dans le fichier `dep.rds`. Utilisez alors la fonction `readRDS()` pour charger le fichier `dep.rds` dans l'objet `dep_rds`. Utilisez la fonction `identical()` pour vérifier que les objets `dep` et `dep_rds` sont bien identiques.

```r
# Sauvegarde de l'objet dep dans le fichier dep.rds
saveRDS(dep, "dep.rds")

# Chargement du fichier dep.rds dans l'objet dep_rds
dep_rds <- readRDS("dep.rds")

# Remarque : la différence essentielle avec la fonction load()
# est qu'il est impératif d'envoyer le résultat de readRDS()
# dans un objet déterminé (alors que load() conserve le nom
# des objets). On évite ce faisant avec readRDS() les conflits
# de noms (quand un objet est écrasé silencieusement par load()).

# Vérification que dep et dep_rds sont identiques
identical(dep, dep_rds)
  ## [1] TRUE
```

Module 2

Manipuler les éléments fondamentaux du langage

La philosophie de ce deuxième module diffère sensiblement de celle des modules 1 et 3. Son objectif est de vous amener à **manipuler les briques élémentaires du langage de R : vecteurs, matrices et listes**. À ce titre, il s'agit d'un détour indispensable avant d'aborder les opérations plus complexes portant sur les tables de données (sélection d'observations et de variables, tri, fusion, etc.).

Plus encore que les autres modules, il est pensé pour **articuler étroitement apprentissage d'un « vocabulaire » de fonctions et mise en oeuvre autour de cas pratiques**.

Manipuler les vecteurs

Les vecteurs constituent un des types d'objets les plus simples et les plus courants dans R. **Ils interviennent dans la manipulation de la plupart des autres types d'objets** et méritent à ce titre une attention particulière.

Exemples Les variables d'une table sont des vecteurs, tout comme la plupart des paramètres passés à une fonction.

Créer des vecteurs et connaître leurs caractéristiques

La **fonction `c()`** permet de créer des vecteurs :

```
# Création de vecteurs
c(8, 5)
  ## [1] 8 5
c("z","B","e")
  ## [1] "z" "B" "e"
c(TRUE, FALSE, FALSE, TRUE)
  ## [1]  TRUE FALSE FALSE  TRUE
```

Pour associer un vecteur à un nom d'objet, il suffit d'utiliser l'**opérateur d'assignation `<-`** :

```
# Assignation de vecteurs à des noms
a1 <- c(8, 5)
a2 <- c("z", "B", "e")
a3 <- c(TRUE, FALSE, FALSE, TRUE)

# Rappel de la valeur des vecteurs définis
a1
  ## [1] 8 5
a2
  ## [1] "z" "B" "e"
a3
  ## [1]  TRUE FALSE FALSE  TRUE
```

Un vecteur possède plusieurs caractéristiques essentielles (que l'on qualifie d'**attributs**) :

- son **type** : les types les plus courants sont numérique, caractère et logique ;
- sa **longueur** : le nombre d'éléments qui le composent.

Il est possible d'afficher ces attributs avec les **fonctions `str()`, `typeof()` et `length()`**.

```
# Attributs de a1
str(a1)
  ##  num [1:2] 8 5
```

```
typeof(a1)
  ## [1] "double"
length(a1)
  ## [1] 2
# Note : Les vecteurs de type numérique peuvent
# être enregistrés de plusieurs façons différentes
# (double, integer, etc.).

# Attributs de a2
str(a2)
  ##  chr [1:3] "z" "B" "e"
typeof(a2)
  ## [1] "character"
length(a2)
  ## [1] 3

# Attributs de a3
str(a3)
  ##  logi [1:4] TRUE FALSE FALSE TRUE
typeof(a3)
  ## [1] "logical"
length(a3)
  ## [1] 4
```

Les fonctions **`is.numeric()`**, **`is.character()`** et **`is.logical()`** permettent de tester si un vecteur est de type numérique, caractère ou logique respectivement.

```
# Utilisation de is.numeric()
is.numeric(a1)
  ## [1] TRUE
is.numeric(a2)
  ## [1] FALSE
is.numeric(a3)
  ## [1] FALSE

# Utilisation de is.character()
is.character(a1)
  ## [1] FALSE
is.character(a2)
  ## [1] TRUE
is.character(a3)
  ## [1] FALSE

# Utilisation de is.logical()
is.logical(a1)
```

```
  ## [1] FALSE
is.logical(a2)
  ## [1] FALSE
is.logical(a3)
  ## [1] TRUE
```

Remarques :

— Quand on souhaite créer un vecteur de longueur 1, la fonction `c()` est inutile. C'est ce qui a été fait pendant tout le module 1 de la formation.

```
# Création d'un vecteur de longueur 1
a4 <- 2
a5 <- c(2)
identical(a4, a5)
  ## [1] TRUE
```

— Les vecteurs de type logique ne peuvent comporter que **deux valeurs** (en plus des valeurs manquantes `NA`, *cf. infra*) : vrai (`TRUE`) et faux (`FALSE`). **`TRUE` et `FALSE` sont des mots-clés spécifiques qui doivent être écrits intégralement en majuscules** :

```
# Création d'un vecteur logique
a6 <- c(TRUE, FALSE, TRUE, TRUE)
a6
  ## [1]  TRUE FALSE  TRUE  TRUE
is.logical(a6)
  ## [1] TRUE

# Quand TRUE et FALSE ne sont pas écrits intégralement
# en majuscules, des erreurs surviennent
a7 <- c(True, fALSE, true,false)
  ## Error in eval(expr, envir, enclos): objet 'True' introuvable
# R recherche un objet dont le nom est `True` mais n'en
# trouve aucun.
```

— Quand nombres, caractères ou valeurs logiques coexistent dans la définition d'un vecteur, des **conversions automatiques** sont opérées :

```
# Création d'un vecteur mélangeant nombres, caractères
# et valeurs logiques
a8 <- c("a", 2, "b", TRUE)
a8
  ## [1] "a"    "2"    "b"    "TRUE"
```

```
# Des guillemets apparaissent autour des valeurs numériques
# ou logiques : le vecteur est de type caractère
is.character(a8)
  ## [1] TRUE
```

La fonction `c()` permet également de créer un vecteur à partir de plusieurs autres.

```
# Création des vecteurs de type caractère a9 et a10
a9 <- c("a", "b", "c", "d")
a10 <- c("mais", "ou", "et", "donc", "or", "ni", "car")

# Concaténation avec la fonction c()
c(a9, a10)
  ##  [1] "a"    "b"    "c"    "d"    "mais" "ou"   "et"   "donc"
  ##  [9] "or"   "ni"   "car"
c(a10, a9)
  ##  [1] "mais" "ou"   "et"   "donc" "or"   "ni"   "car"  "a"
  ##  [9] "b"    "c"    "d"
```

La fonction `rep()` permet enfin de créer des vecteurs en répétant une ou plusieurs valeurs un certain nombre de fois.

```
# Création d'un vecteur avec la fonction rep()
rep(1, times = 5)
  ## [1] 1 1 1 1 1

# Quand le premier argument de rep() est un vecteur,
# il est répété en entier
rep(c(1, 2), times = 5)
  ##  [1] 1 2 1 2 1 2 1 2 1 2

# Utilisé à la place de times = , l'argument each =
# permet de répéter chaque élément et non le vecteur
# en entier
rep(c(1, 2), each = 5)
  ##  [1] 1 1 1 1 1 2 2 2 2 2
```

Cas pratique 2.1 Créer des vecteurs et connaître leurs caractéristiques

a. Devinez le type et la longueur des vecteurs définis par le code suivant, puis vérifiez-les en créant ces vecteurs et en utilisant les fonctions `str()`, `length()` et `typeof()`.

```
b1 <- c(1, 2, 3)
b2 <- rep(c("aaa","bbb"), times = 2)
b3 <- c(TRUE, FALSE, TRUE)
b4 <- c("TRUE", "FALSE", "FALSE")
b5 <- c(b2, b4)
b6 <- c(b1, b3)
```

```
b1 <- c(1, 2, 3)
# b1 est de type numérique et de longueur 3
str(b1)
  ##  num [1:3] 1 2 3
typeof(b1)
  ## [1] "double"
length(b1)
  ## [1] 3

b2 <- rep(c("aaa","bbb"), times = 2)
# b2 est de type caractère et de longueur 4
str(b2)
  ##  chr [1:4] "aaa" "bbb" "aaa" "bbb"

b3 <- c(TRUE, FALSE, TRUE)
# b3 est de type logique et de longueur 3
str(b3)
  ##  logi [1:3] TRUE FALSE TRUE

b4 <- c("TRUE", "FALSE", "FALSE")
# b4 est de type caractère et de longueur 3
str(b4)
  ##  chr [1:3] "TRUE" "FALSE" "FALSE"
# Note : les mots-clés TRUE et FALSE sont entre
# guillemets, ils sont donc reconnus comme des
# caractères.

b5 <- c(b2, b4)
# b5 est de type caractère et de longueur 7
str(b5)
  ##  chr [1:7] "aaa" "bbb" "aaa" "bbb" "TRUE" "FALSE" "FALSE"
```

```
# Les deux vecteurs  b2 et b4 sont de type caractère
# et respectivement de longueur 4 et 3

b6 <- c(b1, b3)
# b6 est de type numérique et de longueur 6
str(b6)
  ##  num [1:6] 1 2 3 1 0 1
# b1 est de type numérique, b3 de type logique :
# b3 est converti en numérique (TRUE en 1 et
# FALSE en 0) avant la concaténation.
```

---

b. Utilisez la fonction `rep()` pour créer la séquence 1, 2, 1, 2. Utilisez de nouveau `rep()` pour créer la séquence 1, 1, 1, 2, 2, 2. Combinez ces éléments pour créer la séquence 1, 1, 1, 2, 2, 2, 1, 1, 1, 2, 2, 2.

```
# Utilisation de l'argument times = de rep()
rep(c(1, 2), times = 2)
  ## [1] 1 2 1 2

# Utilisation de l'argument each = de rep()
rep(c(1, 2), each = 3)
  ## [1] 1 1 1 2 2 2

# Deux possibilités ici :
# - enchasser le second appel de rep() dans le premier
rep(rep(c(1, 2), each = 3), times = 2)
  ##  [1] 1 1 1 2 2 2 1 1 1 2 2 2
# - utiliser directement rep() avec each = et times =
rep(c(1, 2), each = 3, times = 2)
  ##  [1] 1 1 1 2 2 2 1 1 1 2 2 2
```

---

c. Créez la fonction `maSequence(x, y)` telle que `maSequence(c("a", "b"), c("c", "d"))` génère automatiquement la séquence :

```
##  [1] "a" "a" "b" "b" "c" "c" "c" "d" "d" "d" "a" "a" "b" "b" "c"
## [16] "c" "c" "d" "d" "d"
```

```r
# Dans l'exemple fourni, chaque élément du vecteur c("a", "b")
# est répété deux fois et chaque élément du vecteur c("c", "d")
# est répété trois fois.
rep(c("a", "b"), each = 2)
  ## [1] "a" "a" "b" "b"
rep(c("c", "d"), each = 3)
  ## [1] "c" "c" "c" "d" "d" "d"

# Puis l'ensemble de la séquence est répétée deux fois.
# On commence donc par concaténer les deux séquences
# élémentaires.
c(rep(c("a", "b"), each = 2), rep(c("c", "d"), each = 3))
  ##  [1] "a" "a" "b" "b" "c" "c" "c" "d" "d" "d"

# Puis on utilise de nouveau rep(), avec l'argument times
rep(
  c(rep(c("a", "b"), each = 2), rep(c("c", "d"), each = 3))
  , times = 2
)
  ##  [1] "a" "a" "b" "b" "c" "c" "c" "d" "d" "d" "a" "a" "b" "b" "c"
  ## [16] "c" "c" "d" "d" "d"
# C'est bien le résultat attendu

# Pour créer une fonction, il suffit d'utiliser
# l'opérateur d'assignation <- avec le mot-clé function()
maSequence <- function(x, y){

}

# Pour l'heure la fonction est vide et ne renvoie aucun
# résultat (NULL)
maSequence(c("a", "b"), c("c", "d"))
  ## NULL

# Il suffit d'adapter le code développé avec c("a", "b")
# et c("c", "d") à l'intérieur de la fonction
maSequence <- function(x, y){
  resultat <- rep(c(rep(x, each = 2), rep(y, each = 3)), times = 2)
  return(resultat)
}

# On vérifie que cela fonctionne
maSequence(c("a", "b"), c("c", "d"))
  ##  [1] "a" "a" "b" "b" "c" "c" "c" "d" "d" "d" "a" "a" "b" "b" "c"
```

```
  ## [16] "c" "c" "d" "d" "d"

# On essaie avec d'autres arguments
maSequence(c(1, 2, 3), 4)
  ##  [1] 1 1 2 2 3 3 4 4 4 1 1 2 2 3 3 4 4 4
```

Extraire les éléments d'un vecteur

L'opérateur d'extraction `[` permet de sélectionner des éléments en utilisant leur **position dans le vecteur** :

```
# Définition du vecteur c1
c1 <- c("a","b","c","d","e","f","g","h","i","j")
c1
  ##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"

# Sélection de l'élément en position 2
c1[2]
  ## [1] "b"

# Sélection de l'élément en position 5
c1[5]
  ## [1] "e"
```

Pour extraire plus d'une valeur à la fois, il suffit d'utiliser l'opérateur `[` avec le **vecteur des positions souhaitées** :

```
# Sélection des éléments en position 3 et 6
c1[c(3, 6)]
  ## [1] "c" "f"
```

Pour sélectionner **toutes les valeurs sauf certaines**, il suffit de d'indiquer leur **position précédée de `-`** :

```
# Sélection de tous les éléments SAUF celui en position 3
c1[-3]
  ## [1] "a" "b" "d" "e" "f" "g" "h" "i" "j"

# Sélection de tous les éléments SAUF ceux en position 2 et 7
c1[-c(2,7)]
  ## [1] "a" "c" "d" "e" "f" "h" "i" "j"
```

Il est également possible de **définir des vecteurs dont chaque élément est nommé** :

```
# Création du vecteur numérique c2 nommé
c2 <- c("pierre" = 1, "feuille" = 2, "ciseaux" = 3)
c2
  ##  pierre feuille ciseaux
  ##       1       2       3
```

Il est alors possible d'**utiliser les noms pour sélectionner un ou plusieurs éléments** :

```
# Sélection de l'élément associé au nom "pierre"
c2["pierre"]
  ## pierre
  ##      1

# Sélection des éléments associés aux noms "ciseaux" et "feuille"
c2[c("ciseaux","feuille")]
  ## ciseaux feuille
  ##       3       2
```

Il est possible d'**afficher et de modifier les noms** associés à un vecteur en utilisant la **fonction `names()`** :

```
# Affichage des noms associés au vecteur c2
names(c2)
  ## [1] "pierre"  "feuille" "ciseaux"

# Modification des noms associés au vecteur c2
names(c2) <- c("rouge", "jaune", "bleu")
c2
  ## rouge jaune  bleu
  ##     1     2     3
```

Remarque importante Les éléments d'un vecteur sont extraits **dans l'ordre dans lequel sont renseignés les positions ou les noms**.

```
# On compare le résultat de c1[c(3, 6)] et de c1[c(6, 3)]
c1
  ##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"

c1[c(3, 6)]
  ## [1] "c" "f"

c1[c(6, 3)]
```

```
  ## [1] "f" "c"

# Cela est vrai également quand l'extraction est opérée par les noms
c2
  ## rouge jaune  bleu
  ##     1     2     3

c2[c("rouge", "jaune")]
  ## rouge jaune
  ##     1     2

c2[c("jaune", "rouge")]
  ## jaune rouge
  ##     2     1
```

En cas de répétition d'une position ou d'un nom, l'élément du vecteur correspondant est répété dans le résultat :

```
c1[c(3, 3, 6, 6)]
  ## [1] "c" "c" "f" "f"

c2[c("rouge", "jaune", "jaune", "rouge")]
  ## rouge jaune jaune rouge
  ##     1     2     2     1
```

Cette propriété est extrêmement importante, dans la mesure où c'est sur elle que repose les **opérations de tri de tables de données** *via* la fonction `order()`(*cf. infra* et module 3).

Cas pratique 2.2 Extraire les valeurs d'un vecteur

a. On définit le vecteur numérique `d1` par `d1 <- c(2, 7, 5, 8)`. Sélectionnez l'élément en troisième position, puis les éléments en quatrième et deuxième positions (dans cet ordre). Sélectionnez enfin tous les éléments sauf celui en première position.

```
d1 <- c(2, 7, 5, 8)
d1
  ## [1] 2 7 5 8
```

```
# Sélection de l'élément en troisième position
d1[3]
  ## [1] 5

# Sélection des éléments en quatrième et en deuxième
# position (dans cet ordre)
d1[c(4, 2)]
  ## [1] 8 7

# Sélection de tous les éléments sauf celui en
# première position
d1[-1]
  ## [1] 7 5 8
```

---

b. On définit le vecteur logique `d2` nommé par `d2 <- c("a" = TRUE, "b" = FALSE, "c" = FALSE, "d" = TRUE, "e" = TRUE)`. Que signifient les lettres `"a"`, `"b"`, `"c"`, `"d"` et `"e"` dans la définition du vecteur ? Proposez deux méthodes pour sélectionner les éléments de `d2` situé en troisième et première position (dans cet ordre).

```
d2 <- c("a" = TRUE, "b" = FALSE, "c" = FALSE, "d" = TRUE, "e" = TRUE)
d2
  ##     a     b     c     d     e
  ##  TRUE FALSE FALSE  TRUE  TRUE
# Les lettres "a", "b", "c", "d" et "e" correspondent à des noms
# associés aux éléments de d2

# Sélection par les positions
d2[c(3, 1)]
  ##     c     a
  ## FALSE  TRUE
# Remarque : on obtient bien un résultat différent de d2[c(1, 3)]
d2[c(1, 3)]
  ##     a     c
  ##  TRUE FALSE

# Sélection par les noms
d2[c("c", "a")]
  ##     c     a
  ## FALSE  TRUE
```

---

c. Affichez le vecteur de noms associé au vecteur `d2` avec la fonction `names()`. Quel est le type de ce vecteur ? Modifiez le vecteur de noms associé au vecteur `d2` et remplacez le par `c(2011, 2012, 2013, 2014, 2015)`.

```
# Affichage du vecteur de noms et de ses caractéristiques
# grâce à la fonction names()
names(d2)
  ## [1] "a" "b" "c" "d" "e"
str(names(d2))
  ##  chr [1:5] "a" "b" "c" "d" "e"
# Comme attendu, le vecteur de noms est de type caractère

# Modification du vecteur de noms associés au vecteur
# d2
names(d2) <- c(2011, 2012, 2013, 2014, 2015)
d2
  ##  2011  2012  2013  2014  2015
  ##  TRUE FALSE FALSE  TRUE  TRUE
```

---

d. Que se passe-t-il quand vous saisissez `d2[c(2012, 2015)]`. Comment le comprenez-vous ? Quel code proposeriez-vous pour sélectionner les éléments dont les noms sont `"2012"` et `"2015"` ?

Indication Quel est le type du vecteur de noms associé à `d2` ?

```
d2[c(2012, 2015)]
  ## <NA> <NA>
  ##   NA   NA

# On obtient un vecteur comprenant deux valeurs NA (cf. infra)
# et non le vecteur c(FALSE, TRUE) attendu.

# Comme le vecteur c(2012, 2015) est un vecteur numérique,
# le logiciel essaie d'extraire les éléments et 2012ème et
# 2015ème position respectivement, qui n'existent pas.

# Le vecteur de noms associé à d2 est de type caractère
str(names(d2))
  ##  chr [1:5] "2011" "2012" "2013" "2014" "2015"
```

```
# Le vecteur numérique c(2011, 2012, 2013, 2014, 2015) a été
# converti en vecteur caractère au moment de son assignation
# comme vecteur de noms à d2.

# Pour utiliser des noms pour extraire des éléments de d2,
# il suffit de les saisir comme des chaînes de caractères.
d2[c("2012", "2015")]
  ##  2012  2015
  ## FALSE  TRUE
```

Manipuler des vecteurs logiques

Les vecteurs logiques sont particulièrement importants dans la mesure où ils interviennent dans l'évaluation et l'utilisation d'**expresions logiques**. Comme la plupart des langages, R dispose d'opérateurs logiques lui permettant d'évaluer certaines expressions (*cf.* tableau). **Ces opérateurs ne sont rien d'autres que des fonctions dont le résultat est un vecteur logique**.

Code R	Résultat
`a == 1`	Renvoie `TRUE` si `a` vaut 1
`a != 1`	Renvoie `TRUE` si `a` est différent de 1
`a < 1`	Renvoie `TRUE` si `a` est strictement inférieur à 1
`a <= 1`	Renvoie `TRUE` si `a` est inférieur ou égal à 1
`a > 1`	Renvoie `TRUE` si `a` est strictement supérieur à 1
`a >= 1`	Renvoie `TRUE` si `a` est supérieur ou égal à 1
`a & b`	Renvoie `TRUE` si `a` est `TRUE` **et** `b` est `TRUE`
`a | b`	Renvoie `TRUE` si `a` est `TRUE` **ou** `b` est `TRUE`
`!a`	Renvoie `TRUE` si `a` est `FALSE`, `FALSE` si `a` est `TRUE`
`a %in% c(1,2)`	Renvoie `TRUE` si `a` vaut 1 ou 2

```
# Définition du vecteur e1
e1 <- c(11, 12, 13, 14, 15)
e1
  ## [1] 11 12 13 14 15

# Evaluation d'expressions logiques
e1 == 13
  ## [1] FALSE FALSE  TRUE FALSE FALSE
```

```
e1 != 13
  ## [1]  TRUE  TRUE FALSE  TRUE  TRUE

e1 < 13
  ## [1]  TRUE  TRUE FALSE FALSE FALSE

e1 <= 13
  ## [1]  TRUE  TRUE  TRUE FALSE FALSE

!(e1 <= 13)
  ## [1] FALSE FALSE FALSE  TRUE  TRUE

e1 >= 11 & e1 < 14
  ## [1]  TRUE  TRUE  TRUE FALSE FALSE

e1 < 12 | e1 > 14
  ## [1]  TRUE FALSE FALSE FALSE  TRUE

e1 %in% c(11, 13)
  ## [1]  TRUE FALSE  TRUE FALSE FALSE
```

Les vecteurs logiques peuvent ainsi être utilisés dans de nombreuses situations :

— **combinés avec la fonction `sum()`**, pour déterminer le nombre d'éléments d'un vecteur qui respectent une certaine condition :

```
e1
  ## [1] 11 12 13 14 15
# Nombre d'éléments de e1 strictement inférieurs à 13
sum(e1 < 13)
  ## [1] 2
```

— **combinés avec la fonction `which()`**, pour récupérer la position des éléments d'un vecteur respectant une certaine condition :

```
e1
  ## [1] 11 12 13 14 15
# Position des éléments de e1 strictement supérieurs à 12
which(e1 > 12)
  ## [1] 3 4 5
```

— **combinés avec l'opérateur d'extraction `[`**, pour sélectionner ou remplacer les éléments respectant une certaine condition :

```
e1
  ## [1] 11 12 13 14 15
```

```
# Sélection des éléments de e1 dont la valeur
# est strictement inférieure à 13
e1[e1 < 13]
  ## [1] 11 12

# Remplacement des éléments de e1 dont la valeur
# est strictement inférieure à 13 par 0
e1[e1 < 13] <- 0
e1
  ## [1]  0  0 13 14 15
```

À retenir Il existe ainsi **trois méthodes pour extraire les éléments d'un vecteur *via* l'opérateur `[`** :

- utiliser un **vecteur de positions** ;
- utiliser un **vecteur de noms** (quand des noms sont définis) ;
- utiliser un **vecteur logique de même longueur**.

```
e2 <- c("a" = 1, "b" = 2, "c" = 3, "d" = 4, "e" = 5)
e2
  ## a b c d e
  ## 1 2 3 4 5
# Objectif : extraire les éléments en 2ème et 5ème position de e2

# Méthode 1 : par les positions
e2[c(2, 5)]
  ## b e
  ## 2 5

# Méthode 2 : par les noms
e2[c("b", "e")]
  ## b e
  ## 2 5

# Méthode 3 : avec un vecteur logique de longueur 5
# (car e2 est de longueur 5)
e2[c(FALSE, TRUE, FALSE, FALSE, TRUE)]
  ## b e
  ## 2 5
```

Les deux premières méthodes permettent de modifier l'ordre des éléments ou de les répéter, mais pas la troisième :

```
e2
  ## a b c d e
  ## 1 2 3 4 5

e2[c(2, 1, 2, 3, 1)]
  ## b a b c a
  ## 2 1 2 3 1

e2[c("b", "a", "b", "c", "a")]
  ## b a b c a
  ## 2 1 2 3 1

e2[c(TRUE, TRUE, TRUE, FALSE, FALSE)]
  ## a b c
  ## 1 2 3
# Note : il est impossible de changer l'ordre dans lequel apparaissent
# les éléments extraits (ni de les répéter) quand on utilise un vecteur
# logique pour mener l'extraction.
```

L'utilisation de vecteurs logique pour extraire des valeurs est particulièrement importante, dans la mesure où elle intervient dans la plupart des opérations de **sélection d'observations ou de variables** dans une table de données (*cf. infra* et module 3).

Cas pratique 2.3 Manipuler des vecteurs logiques

a. On définit le vecteur `f1 <- c(5, 2, -4, 8)`. Devinez la valeur que renvoient les expressions logiques suivantes, puis vérifiez-les en créant `f1` et en les évaluant.

```
f1 == 2
f1 != 7
f1 < 6
f1 != 2
!(f1  == 2)
f1 > 3 & f1 != 5
(f1 < 1 | f1 > 3) & f1 != 8
f1 %in% c(-4, 7)
```

```
# Création du vecteur f1
f1 <- c(5, 2, -4, 8)
```

```r
# Evaluation des expressions logiques
f1 == 2
  ## [1] FALSE  TRUE FALSE FALSE
f1 != 7
  ## [1] TRUE TRUE TRUE TRUE
f1 < 6
  ## [1]  TRUE  TRUE  TRUE FALSE
f1 != 2
  ## [1]  TRUE FALSE  TRUE  TRUE
!(f1  == 2)
  ## [1]  TRUE FALSE  TRUE  TRUE
f1 > 3 & f1 != 5
  ## [1] FALSE FALSE FALSE  TRUE
(f1 < 1 | f1 > 3) & f1 != 8
  ## [1]  TRUE FALSE  TRUE FALSE
f1 %in% c(-4, 7)
  ## [1] FALSE FALSE  TRUE FALSE
```

---

b. On définit le vecteur `f2 <- rep(c("a","b","a"), times = 10)`. Déterminez automatiquement le nombre d'éléments de `f2` égaux à `"a"` ainsi que leur position. Sélectionnez les éléments égaux à `"b"` et remplacez leur valeur par `"c"`.

```r
# Création du vecteur f2
f2 <- rep(c("a","b","a"), times = 10)
f2
  ##  [1] "a" "b" "a" "a" "b" "a" "a" "b" "a" "a" "b" "a" "a" "b" "a"
  ## [16] "a" "b" "a" "a" "b" "a" "a" "b" "a" "a" "b" "a" "a" "b" "a"

# Nombre d'éléments égaux à "a"
sum(f2 == "a")
  ## [1] 20

# Position des éléments égaux à "a"
which(f2 == "a")
  ##  [1]  1  3  4  6  7  9 10 12 13 15 16 18 19 21 22 24 25 27 28 30

# Sélection des éléments égaux à "b"
f2[f2 == "b"]
  ##  [1] "b" "b" "b" "b" "b" "b" "b" "b" "b" "b"

# Remplacement des élements égaux à "b" par "c"
```

```
f2[f2 == "b"] <- "c"
f2
  ##  [1] "a" "c" "a" "a" "c" "a" "a" "c" "a" "a" "c" "a" "a" "c" "a"
  ## [16] "a" "c" "a" "a" "c" "a" "a" "c" "a" "a" "c" "a" "a" "c" "a"
```

Manipuler des vecteurs numériques

Plusieurs fonctions sont spécifiquement utilisées pour générer des vecteurs de type numérique :

— `seq()` : **`seq()` produit des séquences de nombres**. Dans les cas courants, elle peut être remplacée par `:` :

```
# Création d'un vecteur avec la fonction seq()
seq(1, 20)
  ##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20

# Remplacement par `:`
1:20
  ##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20

# Un cas particulier où seq() ne peut pas directement être remplacé
# par `:`
seq(1, 20, by = 2)
  ##  [1]  1  3  5  7  9 11 13 15 17 19
```

— les **fonctions `rXXXX` de tirage dans une variable (pseudo-)aléatoire** : R dispose d'un large famille de fonctions tirant de façon pseudo-aléatoire selon une certaine loi (spécifiée par les lettres `XXXX`). **Les plus fréquemment utilisées sont `runif()` (loi uniforme sur [0 ;1]) et `rnorm()` (loi normale centrée réduite)** :

```
# Création d'un vecteur de taille 20 avec la fonction runif()
runif(20)
  ##  [1] 0.26550866 0.37212390 0.57285336 0.90820779 0.20168193
  ##  [6] 0.89838968 0.94467527 0.66079779 0.62911404 0.06178627
  ## [11] 0.20597457 0.17655675 0.68702285 0.38410372 0.76984142
  ## [16] 0.49769924 0.71761851 0.99190609 0.38003518 0.77744522

# Création d'un vecteur de taille 20 avec la fonction rnorm()
rnorm(20)
  ##  [1]  1.51178117  0.38984324 -0.62124058 -2.21469989  1.12493092
```

```
  ##  [6] -0.04493361 -0.01619026  0.94383621  0.82122120  0.59390132
  ## [11]  0.91897737  0.78213630  0.07456498 -1.98935170  0.61982575
  ## [16] -0.05612874 -0.15579551 -1.47075238 -0.47815006  0.41794156
```

Les **opérations arithmétiques** sont appliquées termes à termes sur des vecteurs :

```
# Génération de deux vecteurs numériques
g1 <- rep(2, times = 10)
g1
  ##  [1] 2 2 2 2 2 2 2 2 2 2
g2 <- 1:10
g2
  ##  [1]  1  2  3  4  5  6  7  8  9 10

# Application d'opérateurs arithmériques
g1 + g2
  ##  [1]  3  4  5  6  7  8  9 10 11 12
g1 - g2
  ##  [1]  1  0 -1 -2 -3 -4 -5 -6 -7 -8
g1 * g2
  ##  [1]  2  4  6  8 10 12 14 16 18 20
g1 / g2
  ##  [1] 2.0000000 1.0000000 0.6666667 0.5000000 0.4000000 0.3333333
  ##  [7] 0.2857143 0.2500000 0.2222222 0.2000000
```

Quand les vecteurs ne sont pas de même longueur, les éléments du plus petit des deux sont automatiquement répétés. Un avertissement apparaît quand la longueur du plus grand vecteur n'est pas un multiple de la longueur du plus petit.

```
g1
  ##  [1] 2 2 2 2 2 2 2 2 2 2

# Répétition automatique des éléments du vecteur g3
g3 <- 1:5
g3
  ## [1] 1 2 3 4 5
g1 + g3
  ##  [1] 3 4 5 6 7 3 4 5 6 7

# Répétition automatique des éléments du vecteur g4
g4 <- 1:3
g4
  ## [1] 1 2 3
g1 + g4
```

```
  ## Warning in g1 + g4: la taille d'un objet plus long n'est pas
  ## multiple de la taille d'un objet plus court
  ##  [1] 3 4 5 3 4 5 3 4 5 3
```

Cette réutilisation des éléments du vecteur permet de très simplement effectuer des **opérations entre un vecteur de taille quelconque et un scalaire** (*i.e.* un vecteur de taille 1).

```
# Opération entre un vecteur et un scalaire
g2 * 3
  ##  [1]  3  6  9 12 15 18 21 24 27 30
# La valeur unique du vecteur c(3) est réutilisée
# pour atteindre la longueur de g2 (10).
```

Enfin, de nombreuses fonctions peuvent être appliquées à l'**ensemble d'un vecteur de type numérique** (*cf.* tableau).

Code R	**Résultat**
`sum(v)`	Somme du vecteur `v`
`cumsum(v)`	Somme cumulée du vecteur `v`
`mean(v)`	Moyenne du vecteur `v`
`quantile(v)`	Quantiles du vecteur `v`
`summary(v)`	Moyenne et quantiles du vecteur `v`
`max(v)`	Valeur maximum du vecteur `v`
`min(v)`	Valeur minimum du vecteur `v`
`which.min(v)`	Position du minimum du vecteur `v`
`which.max(v)`	Position du maximum du vecteur `v`
`round(v, 2)`	Arrondi du vecteur `v` à deux décimales

Cas pratique 2.4 Manipuler des vecteurs numériques

a. Utilisez la fonction `seq()` pour construire la série de nombres de 0 à 10 de 0.5 en 0.5. Comment pourriez-vous y parvenir en utilisant uniquement l'opérateur `:` ?

```
# Méthode directe : utilisation de l'argument by = de seq()
seq(0, 10, by = 0.5)
  ##  [1]  0.0  0.5  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5
  ## [13]  6.0  6.5  7.0  7.5  8.0  8.5  9.0  9.5 10.0
```

```
# Méthode "manuelle" : utilisation de `:` et division
(0:20) / 2
  ##  [1]  0.0  0.5  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5
  ## [13]  6.0  6.5  7.0  7.5  8.0  8.5  9.0  9.5 10.0
```

---

b. Générez un vecteur `h1` de longueur 20 tiré dans une loi uniforme sur [0 ;1]. Sélectionnez les éléments de `h1` dont la position est paire selon deux méthodes, l'une utilisant la fonction `seq()` et l'autre la fonction `rep()`.

```
# Création du vecteur h1 avec runif()
h1 <- runif(20)
h1
  ##  [1] 0.91287592 0.29360337 0.45906573 0.33239467 0.65087047
  ##  [6] 0.25801678 0.47854525 0.76631067 0.08424691 0.87532133
  ## [11] 0.33907294 0.83944035 0.34668349 0.33377493 0.47635125
  ## [16] 0.89219834 0.86433947 0.38998954 0.77732070 0.96061800
# Note : la génération de h1 étant aléatoire, il est
# normal que vous n'obteniez pas exactement les mêmes
# valeurs.

# Méthode avec seq() :
# 1) construction du vecteur des positions
seq(2, 20, by = 2)
  ##  [1]  2  4  6  8 10 12 14 16 18 20
# 2) utilisation avec l'opérateur `[`
h1[seq(2, 20, by = 2)]
  ##  [1] 0.2936034 0.3323947 0.2580168 0.7663107 0.8753213 0.8394404
  ##  [7] 0.3337749 0.8921983 0.3899895 0.9606180

# Méthode avec rep()
# 1) construction d'un vecteur logique
rep(c(FALSE, TRUE), times = 10)
  ##  [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
  ## [11] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
# 1) utilisation avec l'opérateur `[`
h1[rep(c(FALSE, TRUE), times = 10)]
  ##  [1] 0.2936034 0.3323947 0.2580168 0.7663107 0.8753213 0.8394404
  ##  [7] 0.3337749 0.8921983 0.3899895 0.9606180
```

---

c. En vous inspirant de la méthode utilisant la fonction `seq()` de la question précédente, construisez la fonction `elementsPairs(x)` qui retourne automatiquement les éléments du vecteur `x` dont la position est paire.

Indication Généralisez la réponse à la question précédente en utilisant notamment la fonction `length()`.

```
# Création d'un vecteur de test
test <- rnorm(20)
test
  ##  [1] -0.1645236 -0.2533617  0.6969634  0.5566632 -0.6887557
  ##  [6] -0.7074952  0.3645820  0.7685329 -0.1123462  0.8811077
  ## [11]  0.3981059 -0.6120264  0.3411197 -1.1293631  1.4330237
  ## [16]  1.9803999 -0.3672215 -1.0441346  0.5697196 -0.1350546

# Reprise de la méthode avec seq()
test[seq(2, 20, by = 2)]
  ##  [1] -0.2533617  0.5566632 -0.7074952  0.7685329  0.8811077
  ##  [6] -0.6120264 -1.1293631  1.9803999 -1.0441346 -0.1350546

# Difficulté : il faut que la fonction puisse porter
# sur un vecteur de taille quelconque
# La taille du vecteur de positions à générer doit
# donc dépendre de la longueur du vecteur sur lequel
# porte la fonction. Pour ce faire, on utilise la fonction length().

# Définition de la fonction
elementsPairs <- function(x){
  resultat <- x[seq(2, length(x), by = 2)]
  return(resultat)
}

# Appel de la fonction elementsPairs()
elementsPairs(test)
  ##  [1] -0.2533617  0.5566632 -0.7074952  0.7685329  0.8811077
  ##  [6] -0.6120264 -1.1293631  1.9803999 -1.0441346 -0.1350546
elementsPairs(1:10)
  ## [1]  2  4  6  8 10
elementsPairs(1:9)
  ## [1] 2 4 6 8

# Note : Ne fonctionne pas avec des vecteurs de longueur 1
elementsPairs(17)
  ## Error in seq.default(2, length(x), by = 2): signe incorrect de l'argument
```

---

d. Créez un vecteur `h2` de longueur 15 et tiré dans une loi normale centrée réduite. Déterminez sa valeur maximale. En utilisant notamment l'opérateur d'extraction `[`, déterminez alors la deuxième valeur maximale de `h2`.

```
# Génération du vecteur h2 avec la fonction rnorm()
h2 <- rnorm(15)
h2
  ##  [1]  2.40161776 -0.03924000  0.68973936  0.02800216 -0.74327321
  ##  [6]  0.18879230 -1.80495863  1.46555486  0.15325334  2.17261167
  ## [11]  0.47550953 -0.70994643  0.61072635 -0.93409763 -1.25363340
# Note : la génération de h2 étant aléatoire, il est
# normal que vous n'obteniez pas exactement les mêmes
# valeurs.

# Utilisation de la fonctions max()
max(h2)
  ## [1] 2.401618

# Pour déterminez la deuxième valeur maximale de h2,
# il suffit d'appliquer la fonction max au vecteur
# h2 privé de sa valeur maximale

# Deux stratégies :

# 1) utiliser la fonction which.max() pour renvoyer
# la position de la valeur maximale de h2 et l'exclure avec -
which.max(h2)
  ## [1] 1
h2[-which.max(h2)]
  ##  [1] -0.03924000  0.68973936  0.02800216 -0.74327321  0.18879230
  ##  [6] -1.80495863  1.46555486  0.15325334  2.17261167  0.47550953
  ## [11] -0.70994643  0.61072635 -0.93409763 -1.25363340
max(h2[-which.max(h2)])
  ## [1] 2.172612

# 2) utiliser un vecteur logique pour ne sélectionner
# que les valeurs strictement inférieures à la valeur
# maximale
h2 < max(h2)
  ##  [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
  ## [11]  TRUE  TRUE  TRUE  TRUE  TRUE
```

```
h2[h2 < max(h2)]
  ##  [1] -0.03924000  0.68973936  0.02800216 -0.74327321  0.18879230
  ##  [6] -1.80495863  1.46555486  0.15325334  2.17261167  0.47550953
  ## [11] -0.70994643  0.61072635 -0.93409763 -1.25363340
max(h2[h2 < max(h2)])
  ## [1] 2.172612
```

Manipuler des vecteurs caractères

Comme pour les vecteurs de type numérique, il existe dans R des fonctions spécifiquement adaptées pour créer et manipuler des vecteurs de type caractère :

— `nchar()`, `toupper()`, `tolower()` : **`nchar()` renvoie le nombre de caractères** que représente chaque élément d'un vecteur de type caractère, les fonctions **`tolower()` et `toupper()` convertissent un vecteur caractère en minuscules et majuscules** respectivement.

```
# Création du vecteur i1
i1 <- c("aa", "B", "cccc", "DDD")
i1
  ## [1] "aa"   "B"    "cccc" "DDD"

# Détermination du nombre de caractères avec nchar()
nchar(i1)
  ## [1] 2 1 4 3

# Passage en minuscules ou en majuscules
tolower(i1)
  ## [1] "aa"   "b"    "cccc" "ddd"
toupper(i1)
  ## [1] "AA"   "B"    "CCCC" "DDD"
```

— `paste()` : **`paste()` et sa variante `paste0()` permettent d'agglutiner un ou plusieurs vecteurs caractères**.

```
# Création des vecteurs i2 et i3
i2 <- c("a", "b")
i3 <- c("c", "d")

# Fonctionnement de paste() et paste0()
paste(i2, i3)
  ## [1] "a c" "b d"
```

```
paste(i2, i3, sep = "_")
  ## [1] "a_c" "b_d"
paste0(i2, i3)
  ## [1] "ac" "bd"

# Argument collapse =
paste(i2, collapse = "*")
  ## [1] "a*b"
paste(i2, i3, sep = "_", collapse = "*")
  ## [1] "a_c*b_d"
```

— `formatC` : **`formatC()` convertit un vecteur numérique en vecteur caractère en spécifiant un format**.

```
# Utilisation de formatC() pour ajouter des zéros
# devant des chiffres
formatC(c(1, 2, 56, 789), flag = "0", width = 4)
  ## [1] "0001" "0002" "0056" "0789"
```

— `letters` et `LETTERS` : `letters` et `LETTERS` sont des objets qui contiennent les 26 lettres de l'alphabet, en minuscules et en majuscules respectivement.

```
letters
  ##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o"
  ## [16] "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
LETTERS
  ##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O"
  ## [16] "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

Cas pratique 2.5 Manipuler des vecteurs caractères : Reconstituer un identifiant de fiche-adresse

L'objectif de ce cas pratique est de **reconstituer un identifiant de fiche-adresse** (utilisé dans les enquêtes auprès des ménages de l'Insee) à partir de **trois informations** :

— le **numéro de la région de gestion** (`rges`) codé sur **deux positions** ;
— le **numéro de la fiche-adresse** (`numfa`) codé sur **six positions** (avec des 0 devant si nécessaire) ;
— le **numéro de sous-échantillon** (`ssech`) codé sur **deux positions** (avec un 0 devant si nécessaire).

```
rges <- c(11, 11, 21, 21, 22, 31, 74, 81, 81, 94)
numfa <- c(1, 102, 32, 1219, 98, 3, 678, 21, 89, 45)
ssech <- c(1, 11, 1, 1, 1, 2, 2, 2, 12, 11)
```

L'identifiant de fiche-adresse est défini par la concaténation de `rges`, `numfa` et `ssech` : `rges||numfa||ssech`. Par exemple, si `rges = 11`, `numfa = 1` et `ssech = 1`, l'identifiant de fiche-adresse est `1100000101` (après ajout de `0` intercalaires).

a. Utilisez la fonction `paste()` pour agglutiner les vecteurs `rges`, `numfa` et `ssech`. Utilisez l'argument `sep =` pour supprimer le séparateur. Cela produit-il le résultat souhaité ?

```
# Utilisation de la fonction paste()
paste(rges, numfa, ssech)
  ##  [1] "11 1 1"    "11 102 11" "21 32 1"   "21 1219 1" "22 98 1"
  ##  [6] "31 3 2"    "74 678 2"  "81 21 2"   "81 89 12"  "94 45 11"

# Suppression du séparateur
paste(rges, numfa, ssech, sep = "")
  ##  [1] "1111"    "1110211" "21321"   "2112191" "22981"   "3132"
  ##  [7] "746782"  "81212"   "818912"  "944511"
# Cela ne correspond pas car il manque les 0 devant le numéro
# de fiche-adresse et le numéro de sous-échantillon
```

---

b. Utilisez la fonction `formatC()` pour reformater correctement le vecteur `numfa`. Combinez les fonctions `formatC()` et `paste()` (ou `paste0()`) et appliquez-les à `numfa` et à `ssech` pour obtenir le résultat souhaité.

```
# Utilisation de la fonction formatC()
formatC(numfa, flag = "0", width = 6)
  ##  [1] "000001" "000102" "000032" "001219" "000098" "000003"
  ##  [7] "000678" "000021" "000089" "000045"

# Reformatage complet avec paste0()
paste0(
  formatC(rges, flag = "0", width = 2)
  , formatC(numfa, flag = "0", width = 6)
  , formatC(ssech, flag = "0", width = 2)
)
  ##  [1] "1100000101" "1100010211" "2100003201" "2100121901"
```

```
  ##  [5] "2200009801" "3100000302" "7400067802" "8100002102"
  ##  [9] "8100008912" "9400004511"
# Cette fois-ci on obtient bien le résultat souhaité.
```

---

c. Créez la fonction `creerIdentFA(rges, numfa, ssech)` qui produise automatiquement l'identifiant de fiche-adresse.

```
# Création de la fonction creerIdentFA()
creerIdentFA <- function(rges, numfa, ssech){
  paste0(
    formatC(rges, flag = "0", width = 2)
    , formatC(numfa, flag = "0", width = 6)
    , formatC(ssech, flag = "0", width = 2)
  )
}
creerIdentFA(rges, numfa, ssech)
  ##  [1] "1100000101" "1100010211" "2100003201" "2100121901"
  ##  [5] "2200009801" "3100000302" "7400067802" "8100002102"
  ##  [9] "8100008912" "9400004511"
```

Modifier la structure d'un vecteur

Plusieurs fonctions sont susceptibles d'être appliquées à un vecteur pour modifier ses caractéristiques :

— les **opérations ensemblistes** : fonctions `intersect()` et `setdiff()`

```
# Création des vecteurs k1 et k2
k1 <- letters[1:4]
k2 <- letters[3:6]
k1
  ## [1] "a" "b" "c" "d"
k2
  ## [1] "c" "d" "e" "f"

# Intersection de k1 et k2
intersect(k1, k2)
  ## [1] "c" "d"
```

```r
# Elements présents dans k1 mais pas dans k2
setdiff(k1, k2)
  ## [1] "a" "b"

# Elements présents dans k2 mais pas dans k1
setdiff(k2, k1)
  ## [1] "e" "f"
```

— les fonctions de **traitement des doublons** : la fonction `duplicated(x)` indique si un élément est le doublon d'un élément dont la position est inférieure dans le vecteur `x` (autrement dit qui apparaît précédemment dans le vecteur), la fonction `unique(x)` renvoie le vecteur `x` sans doublons.

```r
# Création du vecteur k3
k3 <- c(1, 2, 1, 4, 2, 3)
k3
  ## [1] 1 2 1 4 2 3

# Détection des éléments qui sont des doublons
duplicated(k3)
  ## [1] FALSE FALSE  TRUE FALSE  TRUE FALSE

# Suppression des doublons
unique(k3)
  ## [1] 1 2 4 3
```

— les fonctions de **changement d'ordre** : `rev(x)` inverse l'ordre du vecteur `x`, `sort(x)` renvoie le vecteur `x` trié et `order(x)` renvoie la permutation des positions du vecteur `x` nécessaire pour que `x` soit trié

```r
# Création du vecteur k4
k4 <- c("a", "d", "b", "c")
k4
  ## [1] "a" "d" "b" "c"

# Inversion de k4
rev(k4)
  ## [1] "c" "b" "d" "a"

# Tri de k4 avec sort()
sort(k4)
  ## [1] "a" "b" "c" "d"

# Tri de k4 avec order()
```

```
order(k4)
  ## [1] 1 3 4 2
k4[order(k4)]
  ## [1] "a" "b" "c" "d"
```

Remarque Le tri d'un vecteur avec `order()` est **beaucoup moins intuitif qu'avec `sort()`** et ne présente pas grand intérêt en lui-même. Néanmoins, **seule la méthode avec `order()` est disponible pour trier un tableau de données** (*cf.* module 3), aussi autant se familiariser au plus tôt avec sa logique de fonctionnement !

Cas pratique 2.6 Modifier la structure d'un vecteur : Travailler avec des identifiants

L'objectif de ce cas pratique est d'utiliser les fonctions présentées dans cette sous-partie pour travailler efficacement avec des identifiants dans R. On définit les deux vecteurs suivants :

```
# Départements d'Ile-de-France présents dans une enquête
enq <- c("91", "75", "75", "94", "93", "94", "78", "77", "77")

# Liste des départements de la petite couronne
pc <- c("75","92","93","94")
```

a. À l'aide d'opérations ensemblistes, déterminez :

   i. les départements de la petite couronne présents dans l'enquête ;
   ii. les départements de la petite couronne absents de l'enquête ;
   iii. les départements de l'enquête qui ne sont pas dans la petite couronne.

```
enq
  ## [1] "91" "75" "75" "94" "93" "94" "78" "77" "77"
pc
  ## [1] "75" "92" "93" "94"

# i.
intersect(enq, pc)
  ## [1] "75" "94" "93"
# Les départements de la petite couronne présents dans
# l'enquête sont le 75, le 94 et le 93
```

```
# ii.
setdiff(pc, enq)
  ## [1] "92"
# Le 92 est le seul département de la petite couronne absent de l'enquête

# iii.
setdiff(enq, pc)
  ## [1] "91" "78" "77"
# Les départements 91, 78 et 77 sont présents dans
# l'enquête mais ne sont pas de la petite couronne
```

---

b. Le vecteur `enq` comporte-t-il des valeurs en double ? Répondez en utilisant la fonction `duplicated()`. Supprimez les valeurs en double dans `enq` avec `duplicated()` ou `unique()`.

```
enq
  ## [1] "91" "75" "75" "94" "93" "94" "78" "77" "77"

# Que renvoie duplicated(enq) ?
duplicated(enq)
  ## [1] FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE
# duplicated(enq) renvoie TRUE si la valeur de l'élément
# est déjà apparue dans le vecteur

# Pour déterminer si enq comporte ou non des valeurs
# en double, il suffit de compter le nombre de valeurs
# vraies de duplicated(enq)
sum(duplicated(enq))
  ## [1] 3

# Deux stratégies pour supprimer les doublons :
# 1) prendre les valeurs fausses de duplicated(enq)
enq[!duplicated(enq)]
  ## [1] "91" "75" "94" "93" "78" "77"
# 2) utiliser la fonction unique()
unique(enq)
  ## [1] "91" "75" "94" "93" "78" "77"
```

---

c. Proposez deux méthodes pour trier le vecteur `enq`, une qui utilise `sort()` et une qui utilise `order()`.

```
enq
  ## [1] "91" "75" "75" "94" "93" "94" "78" "77" "77"

# Méthode directe : Tri avec sort()
sort(enq)
  ## [1] "75" "75" "77" "77" "78" "91" "93" "94" "94"

# Méthode indirecte : Obtention de la permutation avec
# order() puis redéfinition du vecteur
order(enq)
  ## [1] 2 3 8 9 7 1 5 4 6
# Ces nombres indiquent la position des éléments de dep
# à utiliser pour obtenir un vecteur trié.

# Il n'y a donc qu'à utiliser order(enq) avec l'opérateur
# `[` pour obtenir le résultat souhaité
enq[order(enq)]
  ## [1] "75" "75" "77" "77" "78" "91" "93" "94" "94"
```

Savoir traiter les valeurs spéciales

R dispose de plusieurs **valeurs spéciales** qui interviennent dans des situations très différentes :

— `NA` (pour *Not Available*) correspond à des valeurs manquantes. Il est très fréquent en pratique de rencontrer des valeurs `NA` dans des tableaux de données. À noter que les valeurs manquantes sont toujours indiquées par `NA`, quel que soit le type du vecteur.

```
# Exemple de vecteurs présentant des valeurs NA
l1 <- c(1, 2, 3, NA, 5)
l1
  ## [1]  1  2  3 NA  5
l2 <- c("a", "b", NA, "d")
l2
  ## [1] "a" "b" NA  "d"
l3 <- c(NA, NA, TRUE, FALSE)
l3
  ## [1]    NA    NA  TRUE FALSE
```

— `Inf` et `-Inf` correspondent à l'infini en positif et en négatif respectivement.

```
# Exemple de situation dans laquelle survient un Inf
5/0
  ## [1] Inf
```

— `NaN` (pour `Not a Number`) correspond aux cas dans lesquels un calcul mathématique ne conduit à aucun résultat sensé.

```
# Exemple de situation dans laquelle survient un NaN
0/0
  ## [1] NaN
```

Pour identifier (voire supprimer ou remplacer) ces valeurs spéciales, des fonctions spécifiques existent : `is.na()`, `is.infinite()`, `is.nan()`.

```
# Création du vecteur l4
l4 <- c(1, NA, 3, NaN, 5, Inf)
l4
  ## [1]   1  NA   3 NaN   5 Inf

is.na(l4)
  ## [1] FALSE  TRUE FALSE  TRUE FALSE FALSE
# Remarque : la fonction is.na() identifie à la fois les éléments NA
# et les éléments NaN.

is.nan(l4)
  ## [1] FALSE FALSE FALSE  TRUE FALSE FALSE
# Remarque : la fonction is.nan() n'identifie que les éléments NaN
# (pas les éléments NA)

# Pour identifier les éléments NA uniquement (et pas les NaN), il suffit
# de combiner logiquement is.na() et is.nan()
is.na(l4) & !is.nan(l4)
  ## [1] FALSE  TRUE FALSE FALSE FALSE FALSE

is.infinite(l4)
  ## [1] FALSE FALSE FALSE FALSE FALSE  TRUE
```

Ces valeurs **changent le comportement de la plupart des fonctions, notamment les fonctions `sum()`, `table()`**.

```
l1
  ## [1]  1  2  3 NA  5
```

```
# En présence d'une ou plusieurs valeurs NA, la fonction sum()
# renvoie systématiquement NA
sum(l1)
  ## [1] NA

# Pour modifier ce comportement, il suffit d'utiliser l'argument
# na.rm = TRUE de la fonction sum() (taper ? sum pour plus
# d'informations).
sum(l1, na.rm = TRUE)
  ## [1] 11

# Par défaut, la fonction table() n'affiche pas les valeurs manquantes
l5 <- c("Femme", NA, "Homme", "Femme", NA, "Femme")
table(l5)
  ## l5
  ## Femme Homme
  ##     3     1

# Utiliser l'argument useNA = "always" permet d'afficher
# toujours le nombre de valeurs NA (y compris quand il n'y
# en a 0).
table(l5, useNA = "always")
  ## l5
  ## Femme Homme  <NA>
  ##     3     1     2
```

Remarque importante En présence d'une valeur `NA`, **l'opérateur == renvoie `NA`**. Ce comportement ne correspond pas à celui d'autres logiciels statistiques et peut s'avérer **source d'erreur dans le recodage de variables**. Pour cette raison, on peut lui **préférer systématiquement l'opérateur `%in%`**.

```
l5
  ## [1] "Femme" NA      "Homme" "Femme" NA      "Femme"

# En présence de valeurs NA, == renvoie NA
l5 == "Homme"
  ## [1] FALSE    NA  TRUE FALSE    NA FALSE

# En présence de valeurs NA, %in% renvoie FALSE
l5 %in% "Homme"
  ## [1] FALSE FALSE  TRUE FALSE FALSE FALSE
```

Cas pratique 2.7 (Optionnel) Savoir traiter les valeurs spéciales

a. On définit le vecteur `m1 <- c(1, 2, NA, NaN, 5, 6, Inf, 8, 9, NA, NA, -Inf, NaN, 14)`. Comptez le nombre de valeurs `NA` ou `NaN` d'une part, le nombre de valeurs infinies d'autre part.

```
# Création du vecteur m1
m1 <- c(1, 2, NA, NaN, 5, 6, Inf, 8, 9, NA, NA, -Inf, NaN, 14)
m1
  ##  [1]    1    2   NA  NaN    5    6  Inf    8    9   NA   NA -Inf
  ## [13]  NaN   14

# L'idée est la suivante : construire un vecteur
# logique à l'aide des fonctions is.na(), etc.
# puis utiliser la fonction sum()
is.na(m1)
  ##  [1] FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE
  ## [11]  TRUE FALSE  TRUE FALSE
sum(is.na(m1))
  ## [1] 5

is.infinite(m1)
  ##  [1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
  ## [11] FALSE  TRUE FALSE FALSE
sum(is.infinite(m1))
  ## [1] 2
```

---

b. Utilisez les éléments de la question précédente pour supprimer toutes les valeurs spéciales du vecteur `m1`.

```
# On repart des mêmes éléments qu'à la question
# précédente, mais cette fois-ci les vecteurs
# logiques sont utilisés pour extraire des éléments
# du vecteur x
!is.na(m1)
  ##  [1]  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
  ## [11] FALSE  TRUE FALSE  TRUE
m1[!is.na(m1)]
  ## [1]    1    2    5    6  Inf    8    9 -Inf   14

!is.infinite(m1)
```

```
  ##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE
  ## [11]  TRUE FALSE  TRUE  TRUE
m1[!is.infinite(m1)]
  ##  [1]   1   2  NA NaN   5   6   8   9  NA  NA NaN  14

# On n'a qu'à combiner les deux expressions pour
# obtenir le résultat souhaité
m1[!is.na(m1) & !is.infinite(m1) ]
  ## [1]  1  2  5  6  8  9 14
```

Conversion de type et type facteur

On a vu que quand c'est nécessaire, R modifie le type d'un vecteur pour s'adapter à de nouvelles données.

```
# Création du vecteur logique n1
n1 <- c(FALSE, TRUE, FALSE)

# Conversion en cas de concaténation avec un vecteur
# de type numérique
c(n1, 3)
  ## [1] 0 1 0 3

# Conversion en cas de concaténation avec un vecteur
# de type caractère
c(n1, "a")
  ## [1] "FALSE" "TRUE"  "FALSE" "a"
```

Mais il est aussi parfois très utile de **convertir explicitement des vecteurs d'un type dans un autre**, grâce aux fonctions `as.numeric()`, `as.character()` et `as.logical()`.

```
# Âge codé en caractères
age <- c("56", "14", "78")
as.numeric(age)
  ## [1] 56 14 78

# Indicatrice codée en numérique
indic <- c(1, 0, 0, 1, 0)
as.logical(indic)
  ## [1]  TRUE FALSE FALSE  TRUE FALSE
```

Ces opérations peuvent néanmoins produire des `NA`, en particulier quand un vecteur caractère est converti en vecteur numérique.

```
# Conversion du département en numérique
dep <- c("75", "92", "93", "13", "2A", "2B")
as.numeric(dep)
  ## Warning: NAs introduits lors de la conversion automatique
  ## [1] 75 92 93 13 NA NA
```

Le type « facteur » est un type de vecteur particulier, **à mi-chemin entre le vecteur caractère et le vecteur numérique** :

- les valeurs stockées par R sont des entiers ;
- MAIS à chaque entier est associé un « label » permettant d'**afficher une chaîne de caractère à la place du nombre correspondant**.

Les objets de type facteur sont créés le plus souvent avec la **fonction `as.factor()`**.

```
# Création du vecteur de type factor n2
n2 <- as.factor(c("banane", "pomme", "poire", "banane", "banane"))
n2
  ## [1] banane pomme  poire  banane banane
  ## Levels: banane poire pomme

# Caractéristiques de n2
str(n2)
  ##  Factor w/ 3 levels "banane","poire",..: 1 3 2 1 1
```

La fonction `str()` révèle que les valeurs stockées sont 1, 3, 2, 1, 1, valeurs qui sont « formatées »" par le biais des « labels » (`levels`) banane, poire, pomme.

Remarque On retrouve en fait exactement la **même logique que le formatage de variable dans SAS ou l'utilisation de labels de variables dans Stata**.

Quand une variable de type caractère comporte un nombre limité de modalités distinctes, le type facteur peut induire d'**importants gains de performance** : il est en effet **plus efficace de stocker et de manipuler des nombres entiers que des chaînes de caractère parfois longues**.

R étant un logiciel à l'origine pensé pour la statistique mathématique où les variables proprement caractère sont peu nombreuses, **la plupart des fonctions de base proposent par défaut de convertir les variables de type caractère en variables de type facteur**. C'est notamment le cas des **fonctions d'importation standards** (`read.table()`, `read.dbf()`) mais aussi de la fonction de construction des objets de type **`data.frame`** (*cf.* module 3).

Manipuler les matrices

Les matrices peuvent être vues comme le prolongement en deux dimensions des vecteurs : si ce n'est l'existence de deux jeux de positions au lieu d'un seul et de quelques fonctions spécifiques, leurs principes d'utilisation sont les mêmes.

Le type d'objet utilisé pour stocker des données statistiques, le `data.frame` (*cf.* module 3) présente des points communs avec les matrices (accès aux objets par deux positions, utilisation de fonctions adaptées aux objets en deux dimensions, etc.).

Créer et accéder aux éléments d'une matrice

La **fonction `matrix()`** est la manière la plus simple de créer des matrices.

```
matrix(1:8, nrow = 2, ncol = 4)
  ##      [,1] [,2] [,3] [,4]
  ## [1,]    1    3    5    7
  ## [2,]    2    4    6    8
```

R utilise les valeurs du premier argument (un vecteur de données) pour remplir la matrice dont les dimensions sont indiquées par les arguments `nrow` (nombre de lignes) et `ncol` (nombre de colonnes).

Par défaut, **R remplit la matrice colonne par colonne** : d'abord la première colonne de haut en bas, puis la deuxième de haut en bas, etc. L'argument `byrow` (`FALSE` par défaut) permet de remplir la matrice ligne par ligne.

```
matrix(1:8, nrow = 2, ncol = 4, byrow = TRUE)
  ##      [,1] [,2] [,3] [,4]
  ## [1,]    1    2    3    4
  ## [2,]    5    6    7    8
```

Comme un vecteur, **une matrice a un type** (fonction `typeof()`). Sa longueur (fonction `length()`) correspond à la longueur de son vecteur de données. Ses dimensions sont accessibles *via* les fonctions `dim()`, `nrow()` et `ncol()`.

```
# Création de la matrice o1
o1 <- matrix(letters[1:15], nrow = 3, ncol = 5)
o1
  ##      [,1] [,2] [,3] [,4] [,5]
  ## [1,] "a"  "d"  "g"  "j"  "m"
  ## [2,] "b"  "e"  "h"  "k"  "n"
  ## [3,] "c"  "f"  "i"  "l"  "o"

# Caractéristiques de o1
str(o1)
```

```
  ##  chr [1:3, 1:5] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" ...
typeof(o1)
  ## [1] "character"
length(o1)
  ## [1] 15
dim(o1)
  ## [1] 3 5
nrow(o1)
  ## [1] 3
ncol(o1)
  ## [1] 5
```

On peut toujours à partir d'une matrice **revenir au vecteur de données** en utilisant les fonctions `c()` ou `as.vector()`.

```
# Reconstitution du vecteur de données de
# la matrice o1
c(o1)
  ##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o"
as.vector(o1)
  ##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o"
```

Pour sélectionner un élément dans une matrice, il suffit d'**utiliser l'opérateur `[` avec deux nombres correspondant à la position de l'élément séparés par une virgule** :

```
o1
  ##      [,1] [,2] [,3] [,4] [,5]
  ## [1,] "a"  "d"  "g"  "j"  "m"
  ## [2,] "b"  "e"  "h"  "k"  "n"
  ## [3,] "c"  "f"  "i"  "l"  "o"

# Sélection de l'élément en ligne 2 et colonne 3
o1[2, 3]
  ## [1] "h"
# Note : Le premier nombre correspond à la ligne,
# le second à la colonne
```

Pour sélectionner **une ligne ou une colonne entière**, il suffit de n'indiquer qu'un seul nombre mais bien **toujours la virgule `,`**.

```
o1
  ##      [,1] [,2] [,3] [,4] [,5]
  ## [1,] "a"  "d"  "g"  "j"  "m"
  ## [2,] "b"  "e"  "h"  "k"  "n"
```

```
  ## [3,] "c"  "f"  "i"  "l"  "o"

# Sélection de toute la première ligne
o1[1, ]
  ## [1] "a" "d" "g" "j" "m"

# Sélection de toute la cinquième colonne
o1[, 5]
  ## [1] "m" "n" "o"
```

Il est également possible de **sélectionner les lignes et les colonnes d'une matrice par le biais de vecteurs logiques**.

```
o1
  ##      [,1] [,2] [,3] [,4] [,5]
  ## [1,] "a"  "d"  "g"  "j"  "m"
  ## [2,] "b"  "e"  "h"  "k"  "n"
  ## [3,] "c"  "f"  "i"  "l"  "o"

# Sélection de toute la deuxième ligne
o1[c(FALSE, TRUE, FALSE), ]
  ## [1] "b" "e" "h" "k" "n"

# Sélection des colonnes 2 et 4
o1[, c(FALSE, TRUE, FALSE, TRUE, FALSE)]
  ##      [,1] [,2]
  ## [1,] "d"  "j"
  ## [2,] "e"  "k"
  ## [3,] "f"  "l"
```

Comme pour les vecteurs, il est également **possible d'assigner des noms à une matrice à l'aide des fonctions `rownames()` et `colnames()`**. Quand une matrice dispose de noms, ils peuvent être **utilisés en lieu et place des positions de ligne et de colonne**.

```
# Ajout de noms de lignes à o1
rownames(o1) <- c("pierre", "feuille", "ciseaux")
colnames(o1) <- c("pouce", "index", "majeur", "annulaire", "auriculaire")
o1
  ##         pouce index majeur annulaire auriculaire
  ## pierre  "a"   "d"   "g"    "j"       "m"
  ## feuille "b"   "e"   "h"    "k"       "n"
  ## ciseaux "c"   "f"   "i"    "l"       "o"
```

```
# Sélection d'éléments par le nom
o1["pierre", "index"]
  ## [1] "d"
o1["feuille", ]
  ##       pouce       index      majeur   annulaire auriculaire
  ##         "b"         "e"         "h"         "k"         "n"
o1[, "annulaire"]
  ##  pierre feuille ciseaux
  ##     "j"     "k"     "l"
```

À retenir Comme pour les vecteurs, il existe donc **trois méthodes pour sélectionner des lignes ou des colonnes** dans une matrice *via* l'opérateur `[` :

— utiliser un **vecteur de positions** ;
— utiliser un **vecteur de noms** (quand des noms sont définis) ;
— utiliser un **vecteur logique**.

```
o1
  ##         pouce index majeur annulaire auriculaire
  ## pierre  "a"   "d"   "g"    "j"       "m"
  ## feuille "b"   "e"   "h"    "k"       "n"
  ## ciseaux "c"   "f"   "i"    "l"       "o"

# On cherche à sélectionner la première et de la troisième ligne de o1

# Méthode 1 : par les positions
o1[c(1, 3), ]
  ##         pouce index majeur annulaire auriculaire
  ## pierre  "a"   "d"   "g"    "j"       "m"
  ## ciseaux "c"   "f"   "i"    "l"       "o"

# Méthode 2 : par les noms
o1[c("pierre", "ciseaux"), ]
  ##         pouce index majeur annulaire auriculaire
  ## pierre  "a"   "d"   "g"    "j"       "m"
  ## ciseaux "c"   "f"   "i"    "l"       "o"

# Méthode 3 : avec un vecteur logique
o1[c(TRUE, FALSE, TRUE), ]
  ##         pouce index majeur annulaire auriculaire
  ## pierre  "a"   "d"   "g"    "j"       "m"
  ## ciseaux "c"   "f"   "i"    "l"       "o"
```

```
# Remarque : dans les trois cas, on extrait des lignes sans
# toucher aux colonnes donc on laisse une position vide après
# la virgule dans [, ]
```

Les deux premières méthodes permettent de modifier l'ordre des éléments ou de les répéter, mais pas la troisième :

```
o1
  ##         pouce index majeur annulaire auriculaire
  ## pierre  "a"   "d"   "g"    "j"       "m"
  ## feuille "b"   "e"   "h"    "k"       "n"
  ## ciseaux "c"   "f"   "i"    "l"       "o"

o1[c(3, 1, 1, 3), ]
  ##         pouce index majeur annulaire auriculaire
  ## ciseaux "c"   "f"   "i"    "l"       "o"
  ## pierre  "a"   "d"   "g"    "j"       "m"
  ## pierre  "a"   "d"   "g"    "j"       "m"
  ## ciseaux "c"   "f"   "i"    "l"       "o"


o1[c("ciseaux", "pierre", "pierre", "ciseaux"), ]
  ##         pouce index majeur annulaire auriculaire
  ## ciseaux "c"   "f"   "i"    "l"       "o"
  ## pierre  "a"   "d"   "g"    "j"       "m"
  ## pierre  "a"   "d"   "g"    "j"       "m"
  ## ciseaux "c"   "f"   "i"    "l"       "o"

o1[c(TRUE, FALSE, TRUE), ]
  ##         pouce index majeur annulaire auriculaire
  ## pierre  "a"   "d"   "g"    "j"       "m"
  ## ciseaux "c"   "f"   "i"    "l"       "o"
# Note : il est impossible de changer l'ordre dans lequel apparaissent
# les éléments extraits (ni de les répéter) quand on utilise un vecteur
# logique pour mener l'extraction.
```

Ces différentes méthodes sont **particulièrement utiles en pratique** (*cf.* module 3) :

— l'extraction par les positions est à la base des **tris sur une table de données** ;
— l'extraction avec un vecteur logique est à la base de la **sélection d'observations ou de variables dans une table de données**.

Cas pratique 2.8 Créer et accéder aux éléments d'une matrice

a. Déterminez la valeur, le type et les dimensions des matrices suivantes (sans utiliser le logiciel). Vérifiez ensuite ce qu'il en est.

```
p1 <- matrix(1:10, ncol = 2)
p2 <- matrix(1:10, nrow = 2, byrow = TRUE)
p3 <- matrix(rep(c(TRUE, 1, "a"), times = 5), nrow = 3)
p4 <- matrix(rep(c(TRUE, 1, "a"), each = 5), nrow = 3)
```

```
# Matrice p1
p1
  ##      [,1] [,2]
  ## [1,]    1    6
  ## [2,]    2    7
  ## [3,]    3    8
  ## [4,]    4    9
  ## [5,]    5   10
typeof(p1)
  ## [1] "integer"
dim(p1)
  ## [1] 5 2
nrow(p1)
  ## [1] 5
ncol(p1)
  ## [1] 2

# Matrice p2
p2
  ##      [,1] [,2] [,3] [,4] [,5]
  ## [1,]    1    2    3    4    5
  ## [2,]    6    7    8    9   10
typeof(p2)
  ## [1] "integer"
dim(p2)
  ## [1] 2 5
nrow(p2)
  ## [1] 2
ncol(p2)
  ## [1] 5

# Matrice p3
p3
  ##      [,1]   [,2]   [,3]   [,4]   [,5]
```

```
  ## [1,] "TRUE" "TRUE" "TRUE" "TRUE" "TRUE"
  ## [2,] "1"    "1"    "1"    "1"    "1"
  ## [3,] "a"    "a"    "a"    "a"    "a"
typeof(p3)
  ## [1] "character"
dim(p3)
  ## [1] 3 5
nrow(p3)
  ## [1] 3
ncol(p3)
  ## [1] 5

# Matrice p4
p4
  ##      [,1]   [,2]   [,3] [,4] [,5]
  ## [1,] "TRUE" "TRUE" "1"  "1"  "a"
  ## [2,] "TRUE" "TRUE" "1"  "a"  "a"
  ## [3,] "TRUE" "1"    "1"  "a"  "a"
typeof(p4)
  ## [1] "character"
dim(p4)
  ## [1] 3 5
nrow(p4)
  ## [1] 3
ncol(p4)
  ## [1] 5
```

---

b. On définit la matrice `p5 <- matrix(15:1, nrow = 3)`. Sélectionnez l'élément en position 1, 4, puis toute la troisième ligne et toute la deuxième colonne. Que se passe-t-il quand vous tapez `p5[c(1, 2), c(3, 4)]` ?

```
# Création de p5
p5 <- matrix(15:1, nrow = 3)
p5
  ##      [,1] [,2] [,3] [,4] [,5]
  ## [1,]   15   12    9    6    3
  ## [2,]   14   11    8    5    2
  ## [3,]   13   10    7    4    1

# Sélection des éléments demandés
p5[1, 4]
```

```
  ## [1] 6
p5[3, ]
  ## [1] 13 10  7  4  1
p5[, 2]
  ## [1] 12 11 10

p5[c(1, 2), c(3, 4)]
  ##      [,1] [,2]
  ## [1,]    9    6
  ## [2,]    8    5
# Taper p5[c(1, 2), c(3, 4)] permet de sélectionner
# une sous-matrice définie par les lignes 1 et 2
# d'une part et les colonnes 3 et 4 d'autre part.
```

---

c. Assignez les noms `c("Jacques", "Pierre", "Paul")` et `c("orange", "pomme", "poire", "banane", "abricot")` aux lignes et aux colonnes de `p5` respectivement. Que vaut la valeur au croisement de `Pierre` et de `pomme` ?

```
# Création de p5
rownames(p5) <- c("Jacques", "Pierre", "Paul")
colnames(p5) <- c("orange", "pomme", "poire", "banane", "abricot")
p5
  ##         orange pomme poire banane abricot
  ## Jacques     15    12     9      6       3
  ## Pierre      14    11     8      5       2
  ## Paul        13    10     7      4       1

# Sélection des éléments demandés
p5["Pierre", "pomme"]
  ## [1] 11
```

---

d. Utilisez la fonction `order()` pour trier la matrice `p5` selon les valeurs de sa première colonne pour obtenir :

```
  ##         orange pomme poire banane abricot
  ## Paul        13    10     7      4       1
  ## Pierre      14    11     8      5       2
  ## Jacques     15    12     9      6       3
```

Indication Que vaut `p5[c(3, 2, 1), ]` ? Comment utiliser la fonction `order()` pour automatiser cette opération ?

```
p5
  ##         orange pomme poire banane abricot
  ## Jacques     15    12     9      6       3
  ## Pierre      14    11     8      5       2
  ## Paul        13    10     7      4       1

# L'idée de base est que l'opérateur [ appliqué à une matrice
# permet non seulement de sélectionner des lignes ou des colonnes
# mais aussi de les réarranger comme on le souhaite.

# Par exemple, en tapant p5[c(2, 1, 3), ] on indique vouloir
# obtenir une matrice :
# - dont la première ligne est la deuxième ligne de p5
# - dont la deuxième ligne est la première ligne de p5
# - dont la troisième ligne est la troisième ligne de p5
p5[c(2, 1, 3), ]
  ##         orange pomme poire banane abricot
  ## Pierre      14    11     8      5       2
  ## Jacques     15    12     9      6       3
  ## Paul        13    10     7      4       1

# Qu'en est-il de p5[c(3, 2, 1), ] ?
p5[c(3, 2, 1), ]
  ##         orange pomme poire banane abricot
  ## Paul        13    10     7      4       1
  ## Pierre      14    11     8      5       2
  ## Jacques     15    12     9      6       3

# p5[c(3, 2, 1), ] retourne le résultat désiré car la permutation
# c(3, 2, 1) est celle qui permet de réordonner la première colonne
# de p5
col1 <- p5[, 1]
col1
  ## Jacques  Pierre    Paul
  ##      15      14      13
col1[c(3, 2, 1)]
  ##    Paul  Pierre Jacques
  ##      13      14      15

# Pour totalement automatiser cette opération, il ne reste plus
# qu'à déterminer automatiquement la bonne permutation pour
# effectuer le tri souhaité. C'est précisément ce que fait
# la fonction order() :
```

```
order(col1)
  ## [1] 3 2 1
col1[order(col1)]
  ##    Paul  Pierre Jacques
  ##      13      14      15

# En combinant l'ensemble de ces éléments, on obtient donc
# automatiquement le résultat souhaité :
p5[order(p5[, 1]), ]
  ##         orange pomme poire banane abricot
  ## Paul        13    10     7      4       1
  ## Pierre      14    11     8      5       2
  ## Jacques     15    12     9      6       3
```

(Optionnel) Effectuer des opérations sur les matrices

La plupart des opérations applicables à des vecteurs le sont également à des matrices, en particulier l'ensemble des **opérateurs arithmétiques ou logiques**.

```
q1 <- matrix(1:10, nrow = 2)
q1
  ##      [,1] [,2] [,3] [,4] [,5]
  ## [1,]    1    3    5    7    9
  ## [2,]    2    4    6    8   10
q2 <- matrix(2, nrow = 2, ncol = 5)
q2
  ##      [,1] [,2] [,3] [,4] [,5]
  ## [1,]    2    2    2    2    2
  ## [2,]    2    2    2    2    2

# Opérations arithmétiques ou logiques
q1 + q2
  ##      [,1] [,2] [,3] [,4] [,5]
  ## [1,]    3    5    7    9   11
  ## [2,]    4    6    8   10   12
q1 <= 3
  ##      [,1]  [,2]  [,3]  [,4]  [,5]
  ## [1,] TRUE  TRUE FALSE FALSE FALSE
  ## [2,] TRUE FALSE FALSE FALSE FALSE
```

Certaines opérations sont néanmoins spécifiques aux matrices :

— les fonctions de **concaténation par ligne (`rbind()`) et par colonne (`cbind()`)**

```
q3 <- matrix(1:10, nrow = 2)
q3
  ##      [,1] [,2] [,3] [,4] [,5]
  ## [1,]    1    3    5    7    9
  ## [2,]    2    4    6    8   10
q4 <- matrix(letters[1:10], nrow = 2)
q4
  ##      [,1] [,2] [,3] [,4] [,5]
  ## [1,] "a"  "c"  "e"  "g"  "i"
  ## [2,] "b"  "d"  "f"  "h"  "j"

# Concaténation par ligne
rbind(q3, q4)
  ##      [,1] [,2] [,3] [,4] [,5]
  ## [1,] "1"  "3"  "5"  "7"  "9"
  ## [2,] "2"  "4"  "6"  "8"  "10"
  ## [3,] "a"  "c"  "e"  "g"  "i"
  ## [4,] "b"  "d"  "f"  "h"  "j"

# Concaténation par colonne
cbind(q3, q4)
  ##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
  ## [1,] "1"  "3"  "5"  "7"  "9"  "a"  "c"  "e"  "g"  "i"
  ## [2,] "2"  "4"  "6"  "8"  "10" "b"  "d"  "f"  "h"  "j"
```

— les fonctions liées au **calcul matriciel** : transposition (fonction `t()`), produit matriciel (opérateur `%*%`), calcul de déterminant (fonction `det()`), inversion de matrice (fonction `solve()`).

```
q5 <- matrix(rnorm(6), nrow = 2)
q5
  ##            [,1]        [,2]       [,3]
  ## [1,]  0.2914462 0.001105352 -0.5895209
  ## [2,] -0.4432919 0.074341324 -0.5686687
q6 <- matrix(rnorm(6), nrow = 2)
q6
  ##            [,1]       [,2]      [,3]
  ## [1,] -0.1351786 -1.5235668 0.3329504
  ## [2,]  1.1780870  0.5939462 1.0630998

# Calcul matriciel sur q5 et t(q6)
t(q6)
  ##            [,1]      [,2]
  ## [1,] -0.1351786 1.1780870
```

```
  ## [2,] -1.5235668 0.5939462
  ## [3,]  0.3329504 1.0630998
q5 %*% t(q6)
  ##            [,1]       [,2]
  ## [1,] -0.2373626 -0.2827141
  ## [2,] -0.2426789 -1.0826333
```

— certaines **fonctions d'agrégation** adaptées au cadre matriciel : somme et moyenne selon les lignes (`rowSums()` et `rowMeans()`) ou selon les colonnes (`colSums()` et `colMeans()`).

```
q7 <- matrix(1:10, nrow = 2)
q7
  ##      [,1] [,2] [,3] [,4] [,5]
  ## [1,]    1    3    5    7    9
  ## [2,]    2    4    6    8   10

# Calcul selon les lignes et les colonnes de q7
rowSums(q7)
  ## [1] 25 30
rowMeans(q7)
  ## [1] 5 6
colSums(q7)
  ## [1]  3  7 11 15 19
colMeans(q7)
  ## [1] 1.5 3.5 5.5 7.5 9.5
```

— **la fonction `apply()` pour appliquer n'importe quelle fonction selon les lignes ou les colonnes d'une matrice**

```
q7
  ##      [,1] [,2] [,3] [,4] [,5]
  ## [1,]    1    3    5    7    9
  ## [2,]    2    4    6    8   10

# Récupération du maximum de q7 ligne par ligne
apply(q7, 1, max)
  ## [1]  9 10

# Récupération du maximum de q7 colonne par colonne
apply(q7, 2, max)
  ## [1]  2  4  6  8 10

# Note : le deuxième argument de apply() correspond
# à la dimension selon laquelle on applique la fonction :
# 1 pour les lignes, 2 pour les colonnes.
```

Cas pratique 2.9 (Optionnel) Effectuer des opérations sur les matrices

a. On définit la matrice `r1 <- matrix((1:15)^2, ncol = 5, byrow = TRUE)`.

i. Déterminez le nombre d'éléments supérieurs ou égaux à 60 dans l'ensemble de la matrice, puis dans chaque ligne et dans chaque colonne.

```
r1 <- matrix((1:15)^2, ncol = 5, byrow = TRUE)
r1
  ##      [,1] [,2] [,3] [,4] [,5]
  ## [1,]    1    4    9   16   25
  ## [2,]   36   49   64   81  100
  ## [3,]  121  144  169  196  225

# Le point de départ est l'évaluation de l'expression
r1 >= 60
  ##       [,1]  [,2]  [,3]  [,4]  [,5]
  ## [1,] FALSE FALSE FALSE FALSE FALSE
  ## [2,] FALSE FALSE  TRUE  TRUE  TRUE
  ## [3,]  TRUE  TRUE  TRUE  TRUE  TRUE

# On obtient ainsi une matrice de type logique susceptible
# d'être utilisée dans des fonctions d'agrégation
is.logical(r1 >= 60)
  ## [1] TRUE

# Nombre d'éléments supérieurs ou égaux à 60 dans l'ensemble
# de la matrice
sum(r1 >= 60)
  ## [1] 8

# Nombre d'éléments supérieurs ou égaux à 60 par ligne
rowSums(r1 >= 60)
  ## [1] 0 3 5

# Nombre d'éléments supérieurs ou égaux à 60 par colonne
colSums(r1 >= 60)
  ## [1] 1 1 2 2 2
```

---

ii. Sélectionnez la sous-matrice des colonnes dont le total est strictement supérieur à 200.

```
# Pour construire la sous-matrice des colonnes
# dont le total est supérieur à 200, on commence
# par calculer le total selon les colonnes
colSums(r1)
  ## [1] 158 197 242 293 350

# On peut dès lors évaluer l'expression correspondante
colSums(r1) > 200
  ## [1] FALSE FALSE  TRUE  TRUE  TRUE

# Il ne reste plus qu'à utiliser l'opérateur [
# pour sélectionner les colonnes à l'aide du vecteur
# logique ainsi créé
r1[, colSums(r1) > 200]
  ##      [,1] [,2] [,3]
  ## [1,]    9   16   25
  ## [2,]   64   81  100
  ## [3,]  169  196  225
```

---

b. On définit les matrices `r2 <- matrix(rep("a", times = 6), ncol = 2)` et `r3 <- matrix(rep("b", times = 6), nrow = 2)`. Tentez de les concaténer par les lignes et les colonnes. Que se passe-t-il ? Tentez alors de concaténer `r2` et la transposée de `r3`.

```
r2 <- matrix(rep("a", times = 6), ncol = 2)
r2
  ##      [,1] [,2]
  ## [1,] "a"  "a"
  ## [2,] "a"  "a"
  ## [3,] "a"  "a"
r3 <- matrix(rep("b", times = 6), nrow = 2)
r3
  ##      [,1] [,2] [,3]
  ## [1,] "b"  "b"  "b"
  ## [2,] "b"  "b"  "b"

# Tentative de concaténation par les lignes
rbind(r2, r3)
  ## Error in rbind(r2, r3): le nombre de colonnes des matrices doit correspon

# Tentative de concaténation par les colonnes
```

```
cbind(r2, r3)
  ## Error in cbind(r2, r3): le nombre de lignes des matrices doit correspondre (v

# Le problème vient du fait que les dimensions des
# matrices r2 et r3 ne correspondent pas.

# En revanche, cela devrait mieux fonctionner avec
# la transposée de r3
t(r3)
  ##      [,1] [,2]
  ## [1,] "b"  "b"
  ## [2,] "b"  "b"
  ## [3,] "b"  "b"
rbind(r2, t(r3))
  ##      [,1] [,2]
  ## [1,] "a"  "a"
  ## [2,] "a"  "a"
  ## [3,] "a"  "a"
  ## [4,] "b"  "b"
  ## [5,] "b"  "b"
  ## [6,] "b"  "b"
cbind(r2, t(r3))
  ##      [,1] [,2] [,3] [,4]
  ## [1,] "a"  "a"  "b"  "b"
  ## [2,] "a"  "a"  "b"  "b"
  ## [3,] "a"  "a"  "b"  "b"
```

---

c. On définit la matrice `r4 <- matrix(rnorm(8), nrow = 2)`. Utilisez la fonction `apply()` pour calculer l'écart-type de `r4` (fonction `sd()`) ligne par ligne puis colonne par colonne.

```
r4 <- matrix(rnorm(8), nrow = 2)
r4
  ##            [,1]       [,2]     [,3]      [,4]
  ## [1,] -0.3041839  0.2670988 1.207868 0.7002136
  ## [2,]  0.3700188 -0.5425200 1.160403 1.5868335

# Le principe de la fonction apply() est d'appliquer
# une certaine fonction "le long" d'une dimension d'une matrice
# (ligne ou colonne
```

```r
# Appliquer la fonction sd à la matrice r4 ligne par ligne
apply(r4, 1, sd)
  ## [1] 0.6423801 0.9378169

# Appliquer la fonction sd à la matrice r4 colonne par colonne
apply(r4, 2, sd)
  ## [1] 0.47673332 0.57248696 0.03356296 0.62693488

# Note : le deuxième argument de apply() correspond
# à la dimension selon laquelle on applique la fonction :
# 1 pour les lignes, 2 pour les colonnes.
```

---

d. (Optionnel) On définit la matrice `r5 <- matrix(c("aaaa", "bb", "ccc", "d", "eee", "f"), ncol = 2)`. Utilisez la fonction `apply` pour calculer le nombre maximum de caractère de `r5` ligne par ligne puis colonne par colonne.
Indication Créez la fonction `maxnchar()` qui renvoie, pour un vecteur caractère donné, la longueur en nombre de caractères de son élément le plus long. Utilisez ensuite cette fonction avec `apply()` pour obtenir le résultat attendu.

```r
r5 <- matrix(c("aaaa", "bb", "ccc", "d", "eee", "f"), ncol = 2)
r5
  ##      [,1]   [,2]
  ## [1,] "aaaa" "d"
  ## [2,] "bb"   "eee"
  ## [3,] "ccc"  "f"

# On est exactement dans le même cas que précédemment,
# sinon qu'il n'existe aucune fonction qui calcule
# directement le nombre maximal de caracères d'un vecteur
# de type caractère.

# Ce n'est pas réellement une difficulté, dans la mesure
# où il est très facile dans R de créer ses propres fonctions.
# Ici la fonction maxnchar(x) est créée pour renvoyer
# automatiquement le nombre maximal de caractères d'un
# vecteur de type caractère.
maxnchar <- function(x) max(nchar(x))
maxnchar(c("a", "bb", "ccc", "dddd"))
  ## [1] 4

# On peut manuellement appliquer maxnchar() à la première
```

```
# ligne ou à la première colonne de r5
maxnchar(r5[1, ])
  ## [1] 4
maxnchar(r5[, 1])
  ## [1] 4

# Pour l'appliquer automatiquement ligne par ligne ou colonne
# par colonne, il ne reste plus qu'à l'utiliser dans apply()
apply(r5, 1, maxnchar)
  ## [1] 4 3 3
apply(r5, 2, maxnchar)
  ## [1] 4 3

# Remarque 1 : en fait il n'est pas absolument indispensable
# de donner un nom à la fonction maxnchar().
# On peut également la définir à la volée puis l'utiliser
# immédiatement dans le apply() :
apply(r5, 2, function(x) max(nchar(x)))
  ## [1] 4 3

# Remarque 2 : une méthode (plus simple) consisterait également à
# appliquer la fonction max() à une transformation de la table r5
apply(nchar(r5), 1, max)
  ## [1] 4 3 3
apply(nchar(r5), 2, max)
  ## [1] 4 3
```

Manipuler les listes

Du point de vue de la statistique appliquée, la principale limitation des matrices est qu'elles ne peuvent, comme les vecteurs, contenir qu'un seul type de données. **Il est impossible de construire une matrice dont certaines variables sont de type numérique** (par exemple l'âge des personnes enquêtées) **et d'autres de type caractère** (par exemple leur secteur d'activité). Les matrices ne constituent donc pas un type d'objet susceptible de stocker l'information statistique habituellement mobilisée dans les enquêtes sociales.

Les **listes** constituent en revanche un type d'objet beaucoup plus riche qui permet précisément de rassembler des types d'objets très différents : **une liste peut contenir tous les types d'objet (vecteurs numériques, caractères, logiques, matrices, etc.), y compris d'autres listes**. Cette très grande souplesse fait de la liste l'objet de

prédilection pour **stocker une information complexe et structurée**, en particulier les **résultats de procédures statistiques complexes** (régression, classification, etc.).

Plus encore, le type d'objet utilisé pour stocker des données statistiques, le `data.frame` (*cf.* module 3), est un **cas particulier de liste**. La connaissance et la compréhension du fonctionnement des listes dans R **facilite ainsi considérablement le travail sur des données statistiques**.

Créer et accéder aux éléments d'une liste

La **fonction `list()`** crée une nouvelle liste.

```
s1 <- list(
  1:4
  , c("a","b","c")
  , TRUE
  , matrix(rnorm(4), ncol = 2)
)
s1
  ## [[1]]
  ## [1] 1 2 3 4
  ##
  ## [[2]]
  ## [1] "a" "b" "c"
  ##
  ## [[3]]
  ## [1] TRUE
  ##
  ## [[4]]
  ##            [,1]       [,2]
  ## [1,]  0.5584864 -0.5732654
  ## [2,] -1.2765922 -1.2246126
```

L'affichage d'une liste diffère sensiblement de celui d'une matrice ou d'un vecteur : on distingue **deux niveaux de positions**, d'abord celles indiquées entre double-crochets `[[` puis celle indiquées entre crochets simples `[`.

Comme un vecteur, **une liste a une longueur qui correspond à son nombre d'éléments** au sens du nombre d'éléments intervenant dans la fonction `list()` (positions en double-crochets `[[]`). Quand on affiche sa structure, R affiche également celle des éléments qui composent la liste.

```
# Caractéristiques de s1
length(s1)
  ## [1] 4
str(s1)
```

```
  ## List of 4
  ##  $ : int [1:4] 1 2 3 4
  ##  $ : chr [1:3] "a" "b" "c"
  ##  $ : logi TRUE
  ##  $ : num [1:2, 1:2] 0.558 -1.277 -0.573 -1.225
```

On constate ici que **la liste `s1` comporte des éléments de type très différents** : un vecteur de nombres entiers, un vecteur caractère, un vecteur logique et même une matrice numérique.

Comme pour les vecteurs, il est possible de **nommer les éléments d'une liste**, soit lors de sa création soit en utilisant la fonction `names()`.

```
 # Affichage des noms de s1
names(s1)
  ## NULL

# Ajout de noms à s1
names(s1) <- c("chat", "chien", "lapin", "poisson rouge")
s1
  ## $chat
  ## [1] 1 2 3 4
  ##
  ## $chien
  ## [1] "a" "b" "c"
  ##
  ## $lapin
  ## [1] TRUE
  ##
  ## $`poisson rouge`
  ##            [,1]       [,2]
  ## [1,]  0.5584864 -0.5732654
  ## [2,] -1.2765922 -1.2246126
str(s1)
  ## List of 4
  ##  $ chat         : int [1:4] 1 2 3 4
  ##  $ chien        : chr [1:3] "a" "b" "c"
  ##  $ lapin        : logi TRUE
  ##  $ poisson rouge: num [1:2, 1:2] 0.558 -1.277 -0.573 -1.225

# Définition directe d'une liste nommée
s2 <- list(
  "pierre" = 1:3
  , "feuille" = FALSE
  , "ciseaux" = letters[1:3]
)
```

```
s2
  ## $pierre
  ## [1] 1 2 3
  ##
  ## $feuille
  ## [1] FALSE
  ##
  ## $ciseaux
  ## [1] "a" "b" "c"
```

Plusieurs opérateurs permettent d'**accéder aux éléments d'une liste** :

— **`[` renvoie la *sous-liste* correspondant aux indices, noms ou positions logiques demandés** :

```
str(s1)
  ## List of 4
  ##  $ chat         : int [1:4] 1 2 3 4
  ##  $ chien        : chr [1:3] "a" "b" "c"
  ##  $ lapin        : logi TRUE
  ##  $ poisson rouge: num [1:2, 1:2] 0.558 -1.277 -0.573 -1.225

 # Utilisation de [
s1[1]
  ## $chat
  ## [1] 1 2 3 4

str(s1[1])
  ## List of 1
  ##  $ chat: int [1:4] 1 2 3 4

s1[c(2, 3)]
  ## $chien
  ## [1] "a" "b" "c"
  ##
  ## $lapin
  ## [1] TRUE

s1[-4]
  ## $chat
  ## [1] 1 2 3 4
  ##
  ## $chien
  ## [1] "a" "b" "c"
  ##
```

```
  ## $lapin
  ## [1] TRUE

s1[c("lapin", "chien")]
  ## $lapin
  ## [1] TRUE
  ##
  ## $chien
  ## [1] "a" "b" "c"

s1[c(TRUE, FALSE, FALSE, TRUE)]
  ## $chat
  ## [1] 1 2 3 4
  ##
  ## $`poisson rouge`
  ##            [,1]       [,2]
  ## [1,]  0.5584864 -0.5732654
  ## [2,] -1.2765922 -1.2246126
```

— **`[[` renvoie l'*élément* correspondant à l'indice ou au nom demandé** (un seul indice ou un seul nom autorisé dans ce cas) ;

```
str(s1)
  ## List of 4
  ##  $ chat         : int [1:4] 1 2 3 4
  ##  $ chien        : chr [1:3] "a" "b" "c"
  ##  $ lapin        : logi TRUE
  ##  $ poisson rouge: num [1:2, 1:2] 0.558 -1.277 -0.573 -1.225

# Utilisation de [[
s1[[1]]
  ## [1] 1 2 3 4

str(s1[[1]])
  ##  int [1:4] 1 2 3 4

s1[[c(2, 4)]]
  ## Error in s1[[c(2, 4)]]: indice hors limites
# Note : [[ ne permet de sélectionner qu'un seul élément
# à la fois.

s1[["lapin"]]
  ## [1] TRUE
```

— **`$` renvoie l'élément correspondant au nom demandé** (ne fonctionne qu'avec des listes nommées).

```
str(s1)
  ## List of 4
  ##  $ chat         : int [1:4] 1 2 3 4
  ##  $ chien        : chr [1:3] "a" "b" "c"
  ##  $ lapin        : logi TRUE
  ##  $ poisson rouge: num [1:2, 1:2] 0.558 -1.277 -0.573 -1.225

# Utilisation de $
s1$chat
  ## [1] 1 2 3 4

str(s1$chat)
  ##  int [1:4] 1 2 3 4
```

Pour effectuer des opérations sur un élément d'une liste, il suffit de le sélectionner avec `[[` ou `$`.

```
str(s1)
  ## List of 4
  ##  $ chat         : int [1:4] 1 2 3 4
  ##  $ chien        : chr [1:3] "a" "b" "c"
  ##  $ lapin        : logi TRUE
  ##  $ poisson rouge: num [1:2, 1:2] 0.558 -1.277 -0.573 -1.225

# Opérations sur le premier élément de s1
s1[[1]]
  ## [1] 1 2 3 4
sum(s1[[1]])
  ## [1] 10
mean(s1$chat)
  ## [1] 2.5
```

Cas pratique 2.10 Créer et accéder aux éléments d'une liste

a. Devinez les valeurs, la longueur et la structure des trois listes suivantes, puis vérifiez-les dans le logiciel.

```
t1 <- list("a", "b", 3, "d")
t2 <- list(c("a", "b", 3, "d"))
t3 <- list(list("a", "b"), 3, "d")
```

```
# t1 est une liste de longueur 4
str(t1)
  ## List of 4
  ##  $ : chr "a"
  ##  $ : chr "b"
  ##  $ : num 3
  ##  $ : chr "d"
# Chaque élément de t1 est un vecteur de longueur 1

# t2 est une liste de longueur 1
str(t2)
  ## List of 1
  ##  $ : chr [1:4] "a" "b" "3" "d"
# L'élément de t2 est un vecteur caractère de longueur 4

# t3 est une liste de longueur 3
str(t3)
  ## List of 3
  ##  $ :List of 2
  ##   ..$ : chr "a"
  ##   ..$ : chr "b"
  ##  $ : num 3
  ##  $ : chr "d"
# Le premier élément de t3 est une liste (comptant
# elle-même deux éléments, deux vecteurs caractères)
# le deuxième est un vecteur numérique et le troisième
# un vecteur caractère.
```

---

b. On définit les objets suivants `t4 <- rep(1:3, each = 4)`, `t5 <- letters[c(5, 2, 3)]` et `t6 <- c(TRUE, FALSE, FALSE)`. Créez la liste `t7` à partir de ces trois objets (dans l'ordre) et affectez à chaque élément de `t7` le nom de son objet d'origine. Proposez trois méthodes pour accéder au deuxième élément de `t7`.

```
# Création des objets t4, t5 et t6
t4 <- rep(1:3, each = 4)
t4
  ##  [1] 1 1 1 1 2 2 2 2 3 3 3 3
t5 <- letters[c(5, 2, 3)]
t5
  ## [1] "e" "b" "c"
t6 <- c(TRUE, FALSE, FALSE)
```

```
t6
  ## [1]  TRUE FALSE FALSE

# Création de la liste t4 nommée
t7 <- list("t4" = t4, "t5" = t5, "t6" = t6)
t7
  ## $t4
  ##  [1] 1 1 1 1 2 2 2 2 3 3 3 3
  ##
  ## $t5
  ## [1] "e" "b" "c"
  ##
  ## $t6
  ## [1]  TRUE FALSE FALSE

# Accéder à l'élément en deuxième position
# - Méthode 1 : [[ avec la position
t7[[2]]
  ## [1] "e" "b" "c"
# - Méthode 2 : [[ avec le nom
t7[["t5"]]
  ## [1] "e" "b" "c"
# - Méthode 3 : $ avec le nom
t7$t5
  ## [1] "e" "b" "c"

# Remarque : on ne peut pas utiliser de vecteur logique
# avec [[ ou $
t7[[c(FALSE, TRUE, FALSE)]]
  ## Error in t7[[c(FALSE, TRUE, FALSE)]]: attempt to select less than one ele
# C'est possible en revanche avec [, mais l'objet retourné
# n'est pas exactement le même :
t7[c(FALSE, TRUE, FALSE)]
  ## $t5
  ## [1] "e" "b" "c"

str(t7[c(FALSE, TRUE, FALSE)])
  ## List of 1
  ##  $ t5: chr [1:3] "e" "b" "c"
str(t7[["t5"]])
  ##  chr [1:3] "e" "b" "c"
# Avec [ l'objet retourné est toujours une liste (la sous-liste
# correspondant aux positions, noms ou valeurs TRUE du vecteur
# logique utilisés) alors qu'avec [[ il s'agit de l'élément lui-même
```

```
# (ici un vecteur de type caractère).
```

---

c. On définit la liste `t8 <- list(matrix(1:6, nrow = 2), matrix(letters[1:6], ncol = 2))`. Quelles sont les dimensions de chaque élément de la liste ? Combien le premier élément de la liste comporte-t-il de valeurs strictement supérieures à 1,8 en tout ? ligne par ligne ?

```
# Définition de la liste t8
t8 <- list(matrix(1:6, nrow = 2), matrix(letters[1:6], ncol = 2))
t8
  ## [[1]]
  ##      [,1] [,2] [,3]
  ## [1,]    1    3    5
  ## [2,]    2    4    6
  ##
  ## [[2]]
  ##      [,1] [,2]
  ## [1,] "a"  "d"
  ## [2,] "b"  "e"
  ## [3,] "c"  "f"
str(t8)
  ## List of 2
  ##  $ : int [1:2, 1:3] 1 2 3 4 5 6
  ##  $ : chr [1:3, 1:2] "a" "b" "c" "d" ...

# Dimensions des éléments de t8
dim(t8[[1]])
  ## [1] 2 3
dim(t8[[2]])
  ## [1] 3 2

# Evaluation d'une clause logique et agrégation
# sur le premier élément de la liste t8
t8[[1]]
  ##      [,1] [,2] [,3]
  ## [1,]    1    3    5
  ## [2,]    2    4    6
t8[[1]] > 1.8
  ##       [,1] [,2] [,3]
  ## [1,] FALSE TRUE TRUE
  ## [2,]  TRUE TRUE TRUE
```

```
sum(t8[[1]] > 1.8)
  ## [1] 5
rowSums(t8[[1]] > 1.8)
  ## [1] 2 3
```

---

d. On définit la liste `t9 <- list(t7, t8)`. Quelle est la nature des éléments de la liste `t9` ? Accédez dans `t9` au premier élément de la liste correspondant à `t7`, puis au premier élément de la liste correspondant à `t8`.

```
# Définition de la liste t9
t9 <- list(t7, t8)
str(t9)
  ## List of 2
  ##  $ :List of 3
  ##   ..$ t4: int [1:12] 1 1 1 1 2 2 2 2 3 3 ...
  ##   ..$ t5: chr [1:3] "e" "b" "c"
  ##   ..$ t6: logi [1:3] TRUE FALSE FALSE
  ##  $ :List of 2
  ##   ..$ : int [1:2, 1:3] 1 2 3 4 5 6
  ##   ..$ : chr [1:3, 1:2] "a" "b" "c" "d" ...
# Note : t9 est une liste emboîtée. t9 contient deux listes,
# qui elles-mêmes contiennent d'autres éléments (trois
# vecteurs pour la première, deux matrices pour la seconde).
str(t9[[1]])
  ## List of 3
  ##  $ t4: int [1:12] 1 1 1 1 2 2 2 2 3 3 ...
  ##  $ t5: chr [1:3] "e" "b" "c"
  ##  $ t6: logi [1:3] TRUE FALSE FALSE
str(t9[[2]])
  ## List of 2
  ##  $ : int [1:2, 1:3] 1 2 3 4 5 6
  ##  $ : chr [1:3, 1:2] "a" "b" "c" "d" ...

# Accès au premier élément de la liste correspondant à t7 dans t9
t9[[1]]$t4
  ##  [1] 1 1 1 1 2 2 2 2 3 3 3 3
t9[[1]][[1]]
  ##  [1] 1 1 1 1 2 2 2 2 3 3 3 3

# Accès au premier élément de la liste correspondant à t8 dans t9
t9[[2]][[1]]
```

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

Effectuer des opérations sur les listes

Comme pour les vecteurs, il est possible de manipuler des listes en utilisant la fonction `c()` et les **opérations ensemblistes** (fonctions `intersect()` et `setdiff()`).

```
# Création de u1 et u2
u1 <- list(1:5, c("a", "b", "c"))
u1
  ## [[1]]
  ## [1] 1 2 3 4 5
  ##
  ## [[2]]
  ## [1] "a" "b" "c"
u2 <- list(1:5, c(FALSE, TRUE, FALSE))
u2
  ## [[1]]
  ## [1] 1 2 3 4 5
  ##
  ## [[2]]
  ## [1] FALSE  TRUE FALSE

# Concaténation de listes avec c()
c(u1, u2)
  ## [[1]]
  ## [1] 1 2 3 4 5
  ##
  ## [[2]]
  ## [1] "a" "b" "c"
  ##
  ## [[3]]
  ## [1] 1 2 3 4 5
  ##
  ## [[4]]
  ## [1] FALSE  TRUE FALSE

# Opérations ensemblistes sur des listes
intersect(u1, u2)
```

```
  ## [[1]]
  ## [1] 1 2 3 4 5
setdiff(u1, u2)
  ## [[1]]
  ## [1] "a" "b" "c"
setdiff(u2, u1)
  ## [[1]]
  ## [1] FALSE  TRUE FALSE
```

De façon analogue à la fonction `apply()` pour les matrices, **la fonction `lapply()` permet d'appliquer la même fonction à chaque élément d'une liste**.

```
# Création de la liste u3
u3 <- list(1:5, 6:10, 11:15)
u3
  ## [[1]]
  ## [1] 1 2 3 4 5
  ##
  ## [[2]]
  ## [1]  6  7  8  9 10
  ##
  ## [[3]]
  ## [1] 11 12 13 14 15

# Somme de chaque élément de la liste
lapply(u3, sum)
  ## [[1]]
  ## [1] 15
  ##
  ## [[2]]
  ## [1] 40
  ##
  ## [[3]]
  ## [1] 65
# Note : le premier argument de lapply() la liste
# sur les éléments de laquelle on souhaite appliquer
# une fonction et le second la fonction en question.

# Extraction du second élément de chaque élément
# de la liste
lapply(u3, function(x) x[2])
  ## [[1]]
  ## [1] 2
  ##
```

```
  ## [[2]]
  ## [1] 7
  ##
  ## [[3]]
  ## [1] 12
```

Quand la chose est possible, **la fonction `sapply()` simplifie le résultat de la fonction `lapply()` pour obtenir en sortie une matrice ou un vecteur et non une liste**.

```
u3
  ## [[1]]
  ## [1] 1 2 3 4 5
  ##
  ## [[2]]
  ## [1]  6  7  8  9 10
  ##
  ## [[3]]
  ## [1] 11 12 13 14 15

# Maximum de chaque élément de la liste
sapply(u3, max)
  ## [1]  5 10 15
# Note : la syntaxe de sapply() est identique à celle
# de lapply().

# Extraction des premier et troisième éléments
# de chaque élément de la liste
sapply(u3, function(x) x[c(1, 3)])
  ##      [,1] [,2] [,3]
  ## [1,]    1    6   11
  ## [2,]    3    8   13
# Note : quand sapply() ne peut pas renvoyer un vecteur,
# il renvoie une matrice : quand sapply() ne peut pas renvoyer
# une matrice, il renvoie un liste.
```

La fonction `do.call()` permet enfin d'appliquer une fonction à l'ensemble des éléments d'une liste sans avoir à les indiquer explicitement. Elle est particulièrement utile pour **concaténer tous les éléments d'une liste avec `cbind()` ou `rbind()`**.

```
# Création de la liste u4
u4 <- list(
  matrix(1:10, nrow = 2)
  , matrix(11:20, nrow = 2)
```

```
  , matrix(21:30, nrow = 2)
)
u4
  ## [[1]]
  ##      [,1] [,2] [,3] [,4] [,5]
  ## [1,]    1    3    5    7    9
  ## [2,]    2    4    6    8   10
  ##
  ## [[2]]
  ##      [,1] [,2] [,3] [,4] [,5]
  ## [1,]   11   13   15   17   19
  ## [2,]   12   14   16   18   20
  ##
  ## [[3]]
  ##      [,1] [,2] [,3] [,4] [,5]
  ## [1,]   21   23   25   27   29
  ## [2,]   22   24   26   28   30

# Concaténation "manuelle" des éléments de u4
rbind(u4[[1]], u4[[2]], u4[[3]])
  ##      [,1] [,2] [,3] [,4] [,5]
  ## [1,]    1    3    5    7    9
  ## [2,]    2    4    6    8   10
  ## [3,]   11   13   15   17   19
  ## [4,]   12   14   16   18   20
  ## [5,]   21   23   25   27   29
  ## [6,]   22   24   26   28   30

# Concaténation automatique avec do.call()
do.call(rbind, u4)
  ##      [,1] [,2] [,3] [,4] [,5]
  ## [1,]    1    3    5    7    9
  ## [2,]    2    4    6    8   10
  ## [3,]   11   13   15   17   19
  ## [4,]   12   14   16   18   20
  ## [5,]   21   23   25   27   29
  ## [6,]   22   24   26   28   30
# Note : le premier argument de do.call() est le nom
# de la fonction à appliquer et le second la liste
# sur laquelle l'appliquer.
```

Cas pratique 2.11 Effectuer des opérations sur les listes

a. On définit les listes `v1 <- list(c(1, 2), c("a", "b", "c"), c(FALSE))` et `v2 <- list(c("k", "j"))`. Comparez `list(v1, v2)` et `c(v1, v2)`. D'où provient selon vous la différence ?

```
# Création de v1 et v2
v1 <- list(c(1, 2), c("a", "b", "c"), FALSE)
str(v1)
  ## List of 3
  ##  $ : num [1:2] 1 2
  ##  $ : chr [1:3] "a" "b" "c"
  ##  $ : logi FALSE
v2 <- list(c("k", "j"))
str(v2)
  ## List of 1
  ##  $ : chr [1:2] "k" "j"

# Comparaison de list(v1, v2) et de c(v1, v2)
str(list(v1, v2))
  ## List of 2
  ##  $ :List of 3
  ##   ..$ : num [1:2] 1 2
  ##   ..$ : chr [1:3] "a" "b" "c"
  ##   ..$ : logi FALSE
  ##  $ :List of 1
  ##   ..$ : chr [1:2] "k" "j"
str(c(v1, v2))
  ## List of 4
  ##  $ : num [1:2] 1 2
  ##  $ : chr [1:3] "a" "b" "c"
  ##  $ : logi FALSE
  ##  $ : chr [1:2] "k" "j"

# list(v1, v2) est une liste imbriquée : elle
# comporte deux éléments qui sont eux-mêmes des
# listes (comptant l'une trois vecteurs et l'autre
# un vecteur)

# c(v1, v2) est une liste de quatre éléments contenant
# directement les quatre mêmes vecteurs que list(v1, v2).
# Il n'y a pas de liste imbriquée dans c(v1, v2).
```

---

b. On définit `v3 <- c(v1, v2)`. Utilisez la fonction `lapply()` avec `typeof()` pour déterminer le type de chaque élément de la liste `v3`. Comparez le résultat obtenu avec celui produit par `sapply()`.

```r
# Création de v3
v3 <- c(v1, v2)

# Pour connaître le type d'un objet, on utilise
# la fonction typeof().
typeof(v3[[2]])
  ## [1] "character"
# Le lapply() va juste servir à appliquer systématiquement
# la fonction typeof() à chaque élément de v3
lapply(v3, typeof)
  ## [[1]]
  ## [1] "double"
  ##
  ## [[2]]
  ## [1] "character"
  ##
  ## [[3]]
  ## [1] "logical"
  ##
  ## [[4]]
  ## [1] "character"

# lapply() retourne toujours une liste en sortie
str(lapply(v3, typeof))
  ## List of 4
  ##  $ : chr "double"
  ##  $ : chr "character"
  ##  $ : chr "logical"
  ##  $ : chr "character"

# Quand c'est possible, sapply() simplifie le résultat
# de lapply() sous la forme d'un vecteur ou d'une matrice
sapply(v3, typeof)
  ## [1] "double"    "character" "logical"   "character"

# Ici sapply() renvoie un vecteur de type caractère
str(sapply(v3, typeof))
  ##  chr [1:4] "double" "character" "logical" "character"
```

---

c. (Optionnel) En vous inspirant de la question précédente, extrayez automatiquement de v3 la sous-liste des objets de type caractère.

```
v3
  ## [[1]]
  ## [1] 1 2
  ##
  ## [[2]]
  ## [1] "a" "b" "c"
  ##
  ## [[3]]
  ## [1] FALSE
  ##
  ## [[4]]
  ## [1] "k" "j"

# Pour sélectionner une sous-liste à partir d'une
# liste, il suffit d'utiliser l'opérateur [
v3[c(TRUE, TRUE, FALSE, FALSE)]
  ## [[1]]
  ## [1] 1 2
  ##
  ## [[2]]
  ## [1] "a" "b" "c"

# L'objectif est alors de produire un vecteur logique
# permettant d'identifier les éléments de type caractère
# de v3.

# Méthode 1 : avec typeof()
sapply(v3, typeof) == "character"
  ## [1] FALSE  TRUE FALSE  TRUE
v3[sapply(v3, typeof) == "character"]
  ## [[1]]
  ## [1] "a" "b" "c"
  ##
  ## [[2]]
  ## [1] "k" "j"

# Méthode 2 : avec is.character()
sapply(v3, is.character)
  ## [1] FALSE  TRUE FALSE  TRUE
v3[sapply(v3, is.character)]
  ## [[1]]
```

```
  ## [1] "a" "b" "c"
  ##
  ## [[2]]
  ## [1] "k" "j"
```

---

d. (Optionnel) Que renvoie `unlist(v3)` ? Que fait la fonction `unlist()` à votre avis (pensez à utiliser l'aide avec `?`) ? En utilisant la fonction `do.call()`, reproduisez le comportement de `unlist()` (au moins dans le cas simple de `v3`).

```
str(v3)
  ## List of 4
  ##  $ : num [1:2] 1 2
  ##  $ : chr [1:3] "a" "b" "c"
  ##  $ : logi FALSE
  ##  $ : chr [1:2] "k" "j"

# Test de unlist(v3)
unlist(v3)
  ## [1] "1"     "2"     "a"     "b"     "c"     "FALSE" "k"
  ## [8] "j"
# Il est clair que unlist() (comme son nom l'indique)
# va chercher à transformer une liste en un objet plus
# simple, en l'occurrence un vecteur.

# On a vu que quand on dispose de deux vecteurs,
# il suffit d'utiliser la fonction c() pour les
# concaténer
c(1:2, rep(3, times = 5))
  ## [1] 1 2 3 3 3 3 3

# On peut ainsi obtenir le même résultat que unlist()
# en appliquant la fonction c() à tous les éléments de v3
c(v3[[1]], v3[[2]], v3[[3]], v3[[4]])
  ## [1] "1"     "2"     "a"     "b"     "c"     "FALSE" "k"
  ## [8] "j"

# Pour appliquer automatiquement la fonction c()) v3,
# on peut utiliser la fonction do.call()
do.call(c, v3)
  ## [1] "1"     "2"     "a"     "b"     "c"     "FALSE" "k"
  ## [8] "j"
```

```
# Note : le premier argument de do.call() est le nom
# de la fonction à appliquer (ici la fonction c()) et
# le second la liste sur laquelle l'appliquer (ici v3).
```

Module 3

Travailler avec des données statistiques

L'objectif de ce troisième et dernier module est de **réutiliser dans un cadre « métier » les briques élémentaires du langage** introduites dans le module précédent :

— présentation du **type `data.frame`** et de ses relations avec les vecteurs, les matrices et les listes ;
— **opérations courantes sur les tables de données statistiques** : sélection d'observations et de variables, création et modification de variable, tris, fusions, etc. ;
— utilisation de R pour la **statistique descriptive et la production de graphiques**

En dernière partie, des **liens complémentaires** sont fournis vers le support de la formation R perfectionnement que j'ai conçue ainsi que vers des **exemples d'utilisation plus spécifiques** du logiciel (analyse de données multidimensionnelle, régression).

Manipuler les `data.frame`

Dans R, la majeure partie des données statistiques se présente sous la forme de **`data.frame`** : ces objets permettent en effet de **représenter sous la forme d'une table** (*i.e.* d'un objet à deux dimensions) **des données de nature tant quantitative** (variables numériques) **que qualitative** (variables de type caractère ou facteur).

Créer des data.frame et y sélectionner des éléments

Pour créer un objet de type `data.frame`, il suffit d'**utiliser la fonction `data.frame()`**.

```
# Création du data.frame df1
df1 <- data.frame(
  var1 = 1:10
  , var2 = letters[1:10]
  , var3 = rep(c(TRUE, FALSE), times = 5)
)

# Caractéristiques de df1
str(df1)
  ## 'data.frame':  10 obs. of  3 variables:
  ##  $ var1: int  1 2 3 4 5 6 7 8 9 10
  ##  $ var2: chr  "a" "b" "c" "d" ...
  ##  $ var3: logi  TRUE FALSE TRUE FALSE TRUE FALSE ...

# Premières lignes de df1
head(df1)
  ##   var1 var2  var3
  ## 1    1    a  TRUE
  ## 2    2    b FALSE
  ## 3    3    c  TRUE
  ## 4    4    d FALSE
  ## 5    5    e  TRUE
  ## 6    6    f FALSE
```

Il est impératif que tous les éléments qui composent un `data.frame` soient de même longueur.

```
# Création du data.frame df3
df3 <- data.frame(
```

```
  var1 = 1:10
  , var2 = 1:15
)
  ## Error in data.frame(var1 = 1:10, var2 = 1:15): les arguments impliquent des nor
```

Remarque Par défaut, **la fonction `data.frame()` convertit les variables caractères en facteurs** (*cf.* module 2). Pour éviter ce comportement (pas toujours souhaitable), il suffit d'utiliser l'**argument `stringsAsFactors = FALSE`**.

```
# Création du data.frame df2
df2 <- data.frame(
  var1 = 1:10
  , var2 = letters[1:10]
  , var3 = rep(c(TRUE, FALSE), times = 5)
  , stringsAsFactors = FALSE
)

# Caractéristiques de df2
str(df2)
  ## 'data.frame':  10 obs. of  3 variables:
  ##  $ var1: int  1 2 3 4 5 6 7 8 9 10
  ##  $ var2: chr  "a" "b" "c" "d" ...
  ##  $ var3: logi  TRUE FALSE TRUE FALSE TRUE FALSE ...
# Note : dans df2 var2 est de type caractère alors que dans
# df1 elle a été automatiquement convertie en factor.
```

Pour empêcher la conversion de caractères en facteurs **pour toute une session**, il suffit de modifier l'option globale `stringsAsFactors`.

```
# Modification de l'option globale stringsAsFactors
options(stringsAsFactors = FALSE)

# Désormais l'option stringsAsFactors n'est plus nécessaire
# dans chaque appel de fonction
df3 <- data.frame(
  var1 = 1:10
  , var2 = letters[1:10]
  , var3 = rep(c(TRUE, FALSE), times = 5)
)
str(df3)
  ## 'data.frame':  10 obs. of  3 variables:
  ##  $ var1: int  1 2 3 4 5 6 7 8 9 10
  ##  $ var2: chr  "a" "b" "c" "d" ...
```

```
  ##  $ var3: logi  TRUE FALSE TRUE FALSE TRUE FALSE ...
```

Du point de vue de sa structure, un `data.frame` est en réalité une **liste dont tous les éléments ont la même longueur** : c'est ce qui permet de le représenter sous la forme d'un **tableau à deux dimensions**.

```
# Un data.frame est une liste...
is.list(df1)
  ## [1] TRUE

# ... dont tous les éléments sont de même longueur
lapply(df1, length)
  ## $var1
  ## [1] 10
  ##
  ## $var2
  ## [1] 10
  ##
  ## $var3
  ## [1] 10
```

De ce fait, **les `data.frame` empruntent leurs caractéristiques tantôt aux listes, tantôt aux matrices** :

— Comme une matrice, un `data.frame` a **deux dimensions** (fonction `dim()`) ; mais comme une liste, sa **longueur** (fonction `length()`) correspond à son nombre d'éléments (son nombre de variables).

```
# Dimensions de df1 : comme une matrice
dim(df1)
  ## [1] 10  3
nrow(df1)
  ## [1] 10
ncol(df1)
  ## [1] 3

# Longueur de df1 : comme une liste
length(df1)
  ## [1] 3
```

— Comme avec une matrice, on accède aux noms de lignes et de colonne d'un `data.frame` avec les fonctions **`rownames()` et `colnames()`** ; mais comme avec une liste, les noms de colonnes sont aussi directement accessibles avec **`names()`**.

```
# rownames() et colnames() : comme avec une matrice
rownames(df1)
  ##  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
colnames(df1)
  ## [1] "var1" "var2" "var3"

# names() : comme avec une liste
names(df1)
  ## [1] "var1" "var2" "var3"
```

— Comme avec une matrice, il est possible d'accéder aux éléments d'un `data.frame` en **indiquant leurs deux positions dans un opérateur `[`** ; mais comme avec une liste, **il est également possible d'utiliser les opérateurs `[[` et `$`**.

```
df1
  ##    var1 var2  var3
  ## 1     1    a  TRUE
  ## 2     2    b FALSE
  ## 3     3    c  TRUE
  ## 4     4    d FALSE
  ## 5     5    e  TRUE
  ## 6     6    f FALSE
  ## 7     7    g  TRUE
  ## 8     8    h FALSE
  ## 9     9    i  TRUE
  ## 10   10    j FALSE
# On cherche à accéder à l'élément en ligne 8, colonne 2 de df1

# - comme une matrice : avec `[` et deux positions
df1[8, 2]
  ## [1] "h"
df1[8, "var2"]
  ## [1] "h"

# - comme une liste : avec `[[` pour sélectionner la colonne,
# puis [ pour sélectionner la ligne
df1[[2]][8]
  ## [1] "h"
df1[["var2"]][8]
  ## [1] "h"

# - comme une liste : avec `$` pour sélectionner la colonne,
# puis [ pour sélectionner la ligne
df1$var2[8]
```

```
  ## [1] "h"
```

Les **fonctions `as.matrix()`, `as.list()` et `as.data.frame()`** permettent de convertir un `data.frame` en liste ou en matrice, et inversement.

```
# Conversion de df1 en matrice
as.matrix(df1)
  ##       var1 var2 var3
  ##  [1,] " 1" "a"  " TRUE"
  ##  [2,] " 2" "b"  "FALSE"
  ##  [3,] " 3" "c"  " TRUE"
  ##  [4,] " 4" "d"  "FALSE"
  ##  [5,] " 5" "e"  " TRUE"
  ##  [6,] " 6" "f"  "FALSE"
  ##  [7,] " 7" "g"  " TRUE"
  ##  [8,] " 8" "h"  "FALSE"
  ##  [9,] " 9" "i"  " TRUE"
  ## [10,] "10" "j"  "FALSE"
# Note : au passage les variables ont toutes été converties
# en caractères, car une matrice ne peut avoir qu'un seul
# et unique type

# Conversion de df1 en liste
as.list(df1)
  ## $var1
  ##  [1]  1  2  3  4  5  6  7  8  9 10
  ##
  ## $var2
  ##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
  ##
  ## $var3
  ##  [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
# Note : on n'a pas à proprement parler affaire ici à une
# "conversion" (un data.frame est une liste) mais plutôt
# à la suppression de certains attributs spécifiques aux
# data.frame (noms de ligne notamment)
rownames(df1)
  ##  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
rownames(as.list(df1))
  ## NULL

# Conversion d'une matrice en data.frame
as.data.frame(matrix(1:10, ncol = 5))
  ##   V1 V2 V3 V4 V5
```

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

# Conversion d'une liste en data.frame
as.data.frame(list(a = 1:5, b = letters[5:1]))
  ##   a b
  ## 1 1 e
  ## 2 2 d
  ## 3 3 c
  ## 4 4 b
  ## 5 5 a
# Note : dans ce cas il est également impératif
# que tous les éléments de la liste aient bien la même
# longueur.
```

Cas pratique 3.1 Sélectionner des variables et des observations dans une table

Ce cas pratique aborde plusieurs **manipulations courantes de sélection de variables et d'observations dans une table**. Comme la plupart des cas pratiques de ce module, il repose sur l'utilisation des données de l'enquête Emploi en continu 2012 restreinte au quatrième trimestre et aux individus en première ou sixième interrogation. Ces données correspondent au fichier `eect4.rds` contenu dans le fichier donnees.zip.

a. Après avoir modifié le répertoire de travail avec `setwd()`, utilisez la fonction `readRDS()` pour charger le fichier `eect4.rds` dans l'objet `eec` (*cf.* le module 1 pour l'utilisation de la fonction `readRDS()`).

```
# Définition du répertoire de travail (répertoire personnel sous AUS)
setwd("U:/R_initiation/donnees")
```

```
# Chargement des données de l'EEC du 2015T4 depuis le fichier eect4.rds
eec <- readRDS("eect4.rds")

# Caractéristiques de l'objet eec
str(eec)
  ## 'data.frame':  34913 obs. of  19 variables:
  ##  $ IDENT    : chr  "G0A56JP6" "G0A56JP6" "G0A56JR6" "G0A56JS6" ...
  ##  $ TRIM     : chr  "4" "4" "4" "4" ...
  ##  $ NOI      : chr  "01" "02" "01" "01" ...
  ##  $ REG      : chr  "11" "11" "11" "11" ...
```

```
##  $ AGE      : chr  "66" "29" "27" "29" ...
##  $ SEXE     : chr  "2" "1" "2" "2" ...
##  $ CSE      : chr  "56" "81" "38" "37" ...
##  $ DIP11    : chr  "71" "42" "10" "11" ...
##  $ ACTEU    : chr  "1" "2" "1" "1" ...
##  $ SALRED   : int  596 NA 2700 2666 11967 NA 2000 2800 2333 3500 ...
##  $ STC      : chr  "2" NA "2" "2" ...
##  $ TAM1D    : chr  NA NA NA NA ...
##  $ AIDREF   : chr  NA "5" NA NA ...
##  $ TPP      : chr  "1" NA "1" "1" ...
##  $ NBAGENF  : chr  "0" "0" "0" "0" ...
##  $ DUHAB    : chr  "7" NA "7" "7" ...
##  $ PUB3FP   : chr  "4" NA "4" "4" ...
##  $ NAIA     : chr  "1946" "1983" "1985" "1983" ...
##  $ EXTRI1613: num  1777 1777 2045 1898 1754 ...
```

---

b. Pour simplifier le travail sur cette table, on souhaite normaliser la casse des noms de variable. Proposez une méthode pour passer l'ensemble des noms de variable en minuscules et appliquez-la.
Indication Pensez à utiliser les fonctions `names()` et `tolower()`.

```
# Pour accéder aux noms de variable de eec, on peut
# utiliser au choix les fonctions names() ou colnames()
names(eec)
##  [1] "IDENT"     "TRIM"      "NOI"       "REG"       "AGE"
##  [6] "SEXE"      "CSE"       "DIP11"     "ACTEU"     "SALRED"
## [11] "STC"       "TAM1D"     "AIDREF"    "TPP"       "NBAGENF"
## [16] "DUHAB"     "PUB3FP"    "NAIA"      "EXTRI1613"

# Pour changer la casse des noms de variables, il suffit
# de remplacer ce vecteur de noms par sa version
# en minuscules. Pour ce faire, on utilise la fonction tolower()
tolower(names(eec))
##  [1] "ident"     "trim"      "noi"       "reg"       "age"
##  [6] "sexe"      "cse"       "dip11"     "acteu"     "salred"
## [11] "stc"       "tam1d"     "aidref"    "tpp"       "nbagenf"
## [16] "duhab"     "pub3fp"    "naia"      "extri1613"

# Il ne reste plus qu'à remplacer les noms originaux par
# les noms passés en minuscules
names(eec) <- tolower(names(eec))
```

```
str(eec)
  ## 'data.frame':  34913 obs. of  19 variables:
  ##  $ ident    : chr  "G0A56JP6" "G0A56JP6" "G0A56JR6" "G0A56JS6" ...
  ##  $ trim     : chr  "4" "4" "4" "4" ...
  ##  $ noi      : chr  "01" "02" "01" "01" ...
  ##  $ reg      : chr  "11" "11" "11" "11" ...
  ##  $ age      : chr  "66" "29" "27" "29" ...
  ##  $ sexe     : chr  "2" "1" "2" "2" ...
  ##  $ cse      : chr  "56" "81" "38" "37" ...
  ##  $ dip11    : chr  "71" "42" "10" "11" ...
  ##  $ acteu    : chr  "1" "2" "1" "1" ...
  ##  $ salred   : int  596 NA 2700 2666 11967 NA 2000 2800 2333 3500 ...
  ##  $ stc      : chr  "2" NA "2" "2" ...
  ##  $ tam1d    : chr  NA NA NA NA ...
  ##  $ aidref   : chr  NA "5" NA NA ...
  ##  $ tpp      : chr  "1" NA "1" "1" ...
  ##  $ nbagenf  : chr  "0" "0" "0" "0" ...
  ##  $ duhab    : chr  "7" NA "7" "7" ...
  ##  $ pub3fp   : chr  "4" NA "4" "4" ...
  ##  $ naia     : chr  "1946" "1983" "1985" "1983" ...
  ##  $ extri1613: num  1777 1777 2045 1898 1754 ...
```

---

c. On souhaite créer deux nouvelles tables ne contenant que les variables sur lesquelles portent différents aspects de l'étude :

   i. `eec2` qui ne contienne que les variables `ident`, `noi`, `acteu` et `extri1613`.

```
# On utilise l'opérateur `[` avec le vecteur caractère des
# noms des variables à conserver
eec2 <- eec[, c("ident", "noi", "acteu", "extri1613")]
head(eec2)
  ##           ident noi acteu extri1613
  ## 315164 G0A56JP6  01     1  1776.563
  ## 315165 G0A56JP6  02     2  1776.563
  ## 315166 G0A56JR6  01     1  2044.920
  ## 315167 G0A56JS6  01     1  1897.519
  ## 315168 G0A56JT6  01     1  1754.300
  ## 315169 G0A56JU6  01     3  1643.808
```

---

ii. `eec3` qui contienne toutes les variables de `eec` à l'exception de `cse`.

Indication Comment créeriez-vous le vecteur des noms des variables de la table `eec` à l'exception de `cse` ? Utilisez-le comme au i. pour sélectionner toutes les variables sauf `cse`.

```
# La méthode la plus générale pour répondre à la question
# consiste à construire le vecteur des noms des variables
# de eec à l'exception de cse puis de l'utiliser comme au i.

# Le vecteur des noms de variables de eec est obtenu avec la
# fonction names()
names(eec)
  ##  [1] "ident"     "trim"      "noi"       "reg"       "age"
  ##  [6] "sexe"      "cse"       "dip11"     "acteu"     "salred"
  ## [11] "stc"       "tam1d"     "aidref"    "tpp"       "nbagenf"
  ## [16] "duhab"     "pub3fp"    "naia"      "extri1613"

# Pour supprimer un ou plusieurs éléments de ce vecteur, il
# suffit d'utiliser la fonction setdiff() (cf. module 2) :
setdiff(names(eec), c("cse"))
  ##  [1] "ident"     "trim"      "noi"       "reg"       "age"
  ##  [6] "sexe"      "dip11"     "acteu"     "salred"    "stc"
  ## [11] "tam1d"     "aidref"    "tpp"       "nbagenf"   "duhab"
  ## [16] "pub3fp"    "naia"      "extri1613"

# Il n'y a plus qu'à utiliser ce vecteur caractère pour
# sélectionner les variables correspondantes de eec
eec3 <- eec[, setdiff(names(eec), c("cse"))]
head(eec3)
  ##           ident trim noi reg age sexe dip11 acteu salred  stc
  ## 315164 G0A56JP6    4  01  11  66    2    71     1    596    2
  ## 315165 G0A56JP6    4  02  11  29    1    42     2     NA <NA>
  ## 315166 G0A56JR6    4  01  11  27    2    10     1   2700    2
  ## 315167 G0A56JS6    4  01  11  29    2    11     1   2666    2
  ## 315168 G0A56JT6    4  01  11  31    1    11     1  11967    2
  ## 315169 G0A56JU6    4  01  11  24    2    10     3     NA <NA>
  ##        tam1d aidref  tpp nbagenf duhab pub3fp naia extri1613
  ## 315164  <NA>   <NA>    1       0     7      4 1946  1776.563
  ## 315165  <NA>      5 <NA>       0  <NA>   <NA> 1983  1776.563
  ## 315166  <NA>   <NA>    1       0     7      4 1985  2044.920
  ## 315167  <NA>   <NA>    1       0     7      4 1983  1897.519
  ## 315168  <NA>   <NA>    1       0     7      4 1981  1754.300
  ## 315169  <NA>      5 <NA>       0  <NA>   <NA> 1988  1643.808
```

```
# Remarque : une méthode un peu moins directe repose sur
# l'utilisation de vecteurs logiques
# L'idée est de renvoyer le vecteur logique des variables
# à conserver, de la façon suivante :
names(eec) != "cse"
  ##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE
  ## [11]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
eec3b <- eec[, names(eec) != "cse"]
identical(eec3, eec3b)
  ## [1] TRUE

# Cette méthode ne peut facilement fonctionner en tant que
# telle que quand on ne souhaite supprimer qu'une seule
# variable. Pour supprimer plusieurs variables en restant
# dans la même logique, on peut utiliser l'opérateur %in%
# et la négation !.

# Ainsi pour supprimer conjointement cse et extri1613 :
!(names(eec) %in% c("cse", "extri1613"))
  ##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE
  ## [11]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
eec3c <- eec[, !(names(eec) %in% c("cse", "extri1613"))]

# Avec setdiff(), on aurait fait :
eec3d <- eec[, setdiff(names(eec), c("cse", "extri1613"))]
identical(eec3c, eec3d)
  ## [1] TRUE
```

---

d. On souhaite désormais créer une nouvelle table `eec4` contenant toutes les variables mais uniquement pour les individus appartenant à la population active (`acteu` vaut `"1"` ou `"2"`). Comment procéderiez-vous ?

```
# On commence par évaluer l'expression logique correspondant
# à la sélection que l'on souhaite effectuer
str(eec$acteu == "1" | eec$acteu == "2")
  ##  logi [1:34913] TRUE TRUE TRUE TRUE TRUE FALSE ...
# Note : On utilise ici str() afin que le vecteur logique
# ne s'affiche pas en entier (il compte plus de 30 000 éléments !)

# On peut aussi utiliser l'opérateur `%in%`
```

```r
str(eec$acteu %in% c("1","2"))
  ##  logi [1:34913] TRUE TRUE TRUE TRUE TRUE FALSE ...

# Nombre d'individus concernés
sum(eec$acteu %in% c("1","2"))
  ## [1] 19126

# Dès lors que les observations à sélectionner sont identifiées
# par un vecteur logique, on peut le réutiliser dans
# l'opérateur `[` pour restreindre la table aux observations souhaitées.
eec4 <- eec[eec$acteu %in% c("1","2"), ]
nrow(eec4)
  ## [1] 19126

# On vérifie qu'il n'y a bien plus aucun inactif (acteu == "3") dans eec4
sum(eec4$acteu == "3")
  ## [1] 0
```

Créer ou modifier des variables dans un `data.frame`

Pour créer une nouvelle variable dans un `data.frame`, **le plus simple est d'utiliser l'opérateur `$`**.

```r
# Création du data.frame df5
df5 <- data.frame(
  var1 = letters[1:4]
  , var2 = rep(c(FALSE, TRUE), times = 2)
  , stringsAsFactors = FALSE
)
df5
  ##   var1  var2
  ## 1    a FALSE
  ## 2    b  TRUE
  ## 3    c FALSE
  ## 4    d  TRUE

# Ajout de la variable var3 avec $
df5$var3 <- (1:4)^2
df5
  ##   var1  var2 var3
  ## 1    a FALSE    1
```

```
  ## 2    b  TRUE    4
  ## 3    c FALSE    9
  ## 4    d  TRUE   16
```

Pour créer une variable **à partir d'une ou plusieurs autres de la table**, il suffit d'**utiliser l'opérateur $ plusieurs fois**.

```
# Création de la variable var4 à partir de var3
df5$var4 <- df5$var3 * 2
df5
  ##   var1  var2 var3 var4
  ## 1    a FALSE    1    2
  ## 2    b  TRUE    4    8
  ## 3    c FALSE    9   18
  ## 4    d  TRUE   16   32

# Conversion de var2 de logique vers numérique
df5$var2 <- as.numeric(df5$var2)
df5
  ##   var1 var2 var3 var4
  ## 1    a    0    1    2
  ## 2    b    1    4    8
  ## 3    c    0    9   18
  ## 4    d    1   16   32
# Note : modifier à la volée une variable existante ne pose
# aucun problème
```

Pour effectuer un **recodage manuel selon une ou plusieurs conditions** (comme un IF THEN ELSE dans SAS), trois méthodes sont disponibles :

1. Pour les variables dichotomiques uniquement, **utiliser des opérateurs logiques** pour créer un nouveau vecteur.

   ```
   # Création de la variable var5 valant TRUE si var4 > 10 et var2 = 1
   df5$var5 <- df5$var4 > 10 & df5$var2 == 1
   df5
     ##   var1 var2 var3 var4  var5
     ## 1    a    0    1    2 FALSE
     ## 2    b    1    4    8 FALSE
     ## 3    c    0    9   18 FALSE
     ## 4    d    1   16   32  TRUE
   ```

2. Créer la variable recodée progressivement en **utilisant l'opérateur [**.

   ```
   # Création de la variable var6 identique à var5
   df5$var6 <- "Non"
   df5$var6[df5$var4 > 10 & df5$var2 == 1] <- "Oui"
   df5
   ```

```
##   var1 var2 var3 var4  var5 var6
## 1    a    0    1    2 FALSE  Non
## 2    b    1    4    8 FALSE  Non
## 3    c    0    9   18 FALSE  Non
## 4    d    1   16   32  TRUE  Oui
```

3. **Utiliser la fonction `ifelse()`.**

```
# Création de la variable var7 identique à var5 et var6
df5$var7 <- ifelse(df5$var4 > 10 & df5$var2 == 1, "Oui", "Non")
df5
```

```
##   var1 var2 var3 var4  var5 var6 var7
## 1    a    0    1    2 FALSE  Non  Non
## 2    b    1    4    8 FALSE  Non  Non
## 3    c    0    9   18 FALSE  Non  Non
## 4    d    1   16   32  TRUE  Oui  Oui
```

La fonction `ifelse()` prend trois arguments : l'expression logique à évaluer, la valeur à renvoyer si l'expression est vraie, la valeur à renvoyer si l'expression est fausse. Il est possible **d'imbriquer des fonctions `ifelse()`** pour effectuer des recodages complexes.

Remarque Savoir tirer parti de la fonction `within()`

Quand on met en oeuvre un recodage, on est fréquemment amené à **répéter le nom du `data.frame` sur lequel on travaille**. **La fonction `within()` permet d'alléger l'écriture d'un recodage** et de faciliter la compréhension d'un code en évitant cette répétition.

```
# Concaténation manuelle des variables var1 à var4
df5$var7 <- paste0(df5$var1, df5$var2, df5$var3, df5$var4)

# Syntaxe allégée avec la fonction within()
# Création de la variable var5, concaténation de
# toutes les autres variables de la table df5
df5 <- within(df5, {
  var8 <- paste0(var1, var2, var3, var4)
})
df5[, c("var7", "var8")]
```

```
##     var7   var8
## 1   a012   a012
## 2   b148   b148
## 3  c0918  c0918
## 4 d11632 d11632
```

Le premier argument de `within()` est le nom du `data.frame` sur lequel porte le recodage, le second est la série d'instructions à appliquer (les accolades sont obligatoires s'il y a plus d'une instruction).

Cas pratique 3.2 Recoder des variables dans des données statistiques

Ce cas pratique vise à appliquer les opérations de création et de modification de variables présentées dans cette partie à des données statistiques classiques. Comme le précédent, il porte sur les données de l'enquête Emploi en continu au 2015T4.

a. La variable `cse` code la Profession et catégorie socioprofessionnelle (PCS) des individus en 42 postes (*cf.* cette page pour plus de détails). Pour des raisons de lisibilité, on souhaite créer la variable agrégée `cs` qui ne conserve que la première position de la nomenclature.

   i. Proposez une première méthode (un peu fastidieuse) s'appuyant sur des recodages manuels (avec l'opérateur `[`).

```
# On commence par regarder les valeurs prises par eec$cse
table(eec$cse)
  ##
  ##   00   11   12   13   21   22   23   31   33   34   35   37   38
  ##    4  174   54  174  577  495   96  329  290  550  192  793  859
  ##   42   43   44   45   46   47   48   52   53   54   55   56   62
  ##  671 1028    7  340 1326  801  367 1679  312 1158  855 1426  885
  ##   63   64   65   67   68   69   81
  ## 1063  466  321  796  581  191  336

# On recode les différents cas un à un avec `[`
eec$cs <- eec$cse
eec$cs[eec$cs %in% c("11", "12", "13")] <- "1"
eec$cs[eec$cs %in% c("21", "22", "23")] <- "2"
eec$cs[eec$cs %in% c("31", "33", "34", "35", "37", "38")] <- "3"
eec$cs[eec$cs %in% c("42", "43", "44", "45", "46", "47", "48")] <- "4"
eec$cs[eec$cs %in% c("52", "53", "54", "55", "56")] <- "5"
eec$cs[eec$cs %in% c("62", "63", "64", "65", "67", "68", "69")] <- "6"
eec$cs[eec$cs  == "81"] <- "8"

# On vérifie ce qu'il reste dans eec$cs
```

```
table(eec$cs)
  ##
  ##   00    1    2    3    4    5    6    8
  ##    4  402 1168 3013 4540 5430 4303  336
eec$cs[eec$cs  == "00"] <- "0"
table(eec$cs)
  ##
  ##    0    1    2    3    4    5    6    8
  ##    4  402 1168 3013 4540 5430 4303  336

# Amélioration : la même chose mais dans un within()
eec <- within(eec, {
  cs <- cse
  cs[cs %in% c("11", "12", "13")] <- "1"
  cs[cs %in% c("21", "22", "23")] <- "2"
  cs[cs %in% c("31", "33", "34", "35", "37", "38")] <- "3"
  cs[cs %in% c("42", "43", "44", "45", "46", "47", "48")] <- "4"
  cs[cs %in% c("52", "53", "54", "55", "56")] <- "5"
  cs[cs %in% c("62", "63", "64", "65", "67", "68", "69")] <- "6"
  cs[cs  == "81"] <- "8"
  cs[cs  == "00"] <- "0"
})
table(eec$cs)
  ##
  ##    0    1    2    3    4    5    6    8
  ##    4  402 1168 3013 4540 5430 4303  336
```

---

ii. Effectuez le même recodage en utilisant la fonction `substr()`.

```
# La fonction substr() permet de sélectionner
# des caractères dans une chaîne
substr(c("abcd", "efgh", "ijkl"), start = 2, stop = 3)
  ## [1] "bc" "fg" "jk"

# Application à l'EEC
eec$cs2 <- substr(eec$cse, start = 1, stop = 1)

# On vérifie qu'on obtient bien la même chose par les deux méthodes
identical(eec$cs, eec$cs2)
  ## [1] TRUE
```

---

b. La variable de position sur le marché du travail (`acteu`) comporte des valeurs manquantes dans le fichier `eec` à votre disposition. On souhaite imputer cette variable de façon déterministe :
   — si la personne est âgée de moins de 67 ans, on considère qu'elle est active occupée (`acteu` vaut `"1"`) ;
   — si la personne est âgée de 67 ans ou plus, on considère qu'elle est inactive (`acteu` vaut `"3"`).

Remarque Le fichier original de l'enquête Emploi en continu ne comporte aucune valeur manquante pour la variable `acteu`, celles-ci ont été ajoutées pour l'exercice.

---

i. Utilisez la fonction `table()` pour affichez le nombre de valeurs `NA` dans la variable `acteu`. Créez la table `eec_pb` ne comportant que les individus pour lesquels la variable `acteu` vaut `NA`.
   Indication Pour créer la table `eec_pb`, pensez à utiliser la fonction `is.na()`.

```
# La fonction table() produit un tri à plat d'une variable
table(eec$acteu)
  ##
  ##     1     2     3
  ## 17096  2030 15663

# Pour afficher les valeurs NA, il convient d'utiliser
# l'argument useNA = "always"
table(eec$acteu, useNA = "always")
  ##
  ##     1     2     3  <NA>
  ## 17096  2030 15663   124

# Création de la table eec_pb
eec_pb <- eec[is.na(eec$acteu), ]
head(eec_pb)
  ##           ident trim noi reg age sexe  cse dip11 acteu salred
  ## 316001 G0M5AV17    4  02  24  33    2   21    50  <NA>     NA
  ## 316013 G0M5IFHE    4  02  31  47    2   68    71  <NA>   1060
  ## 316464 G0Q6BT49    4  01  11  55    1   38    31  <NA>   3450
  ## 318522 G1553YV9    4  01  73  28    2   42    10  <NA>   1411
  ## 318700 G166PJ9A    4  01  31  85    2 <NA>    71  <NA>     NA
```

```
## 319625 G1D6901B    4  02  52  47    2   54    31  <NA>   1100
##         stc tam1d aidref  tpp nbagenf duhab pub3fp naia
## 316001    1  <NA>   <NA>    1       1     7   <NA> 1979
## 316013    2  <NA>   <NA>    2       0     3      4 1964
## 316464    2  <NA>   <NA>    1       4     7      4 1957
## 318522    2  <NA>   <NA>    1       3     7      1 1984
## 318700 <NA>  <NA>   <NA> <NA>       0  <NA>   <NA> 1927
## 319625    2  <NA>   <NA>    1       8     6      4 1965
##        extri1613   cs  cs2
## 316001  2466.816    2    2
## 316013  1189.652    6    6
## 316464  1559.192    3    3
## 318522  9846.237    4    4
## 318700  1328.881 <NA> <NA>
## 319625  1623.205    5    5
```

---

ii. Dans la table `acteu`, créez la variable redressée `acteu_red` en mettant en oeuvre la procédure d'imputation (très frustre) décrite ci-dessus.

```
# On effectue l'imputation décrite ci-dessus
eec <- within(eec, {
  acteu_red <- acteu
  acteu_red[is.na(acteu) & age < "67"] <- "1"
  acteu_red[is.na(acteu) & age >= "67"] <- "3"
})
table(eec$acteu_red, useNA = "always")
  ##
  ##     1     2     3  <NA>
  ## 17202  2030 15681     0
```

---

iii. Recréez la table `eec_pb` et contrôlez que l'imputation s'est déroulée correctement (en vérifiant que les valeurs imputées sont cohérentes avec l'âge des individus).

```
# Re-création de la table eec_pb et contrôle
# des valeurs de acteu_red
eec_pb <- eec[is.na(eec$acteu), ]
```

```
head(eec_pb[, c("age", "acteu", "acteu_red")])
  ##        age acteu acteu_red
  ## 316001  33  <NA>         1
  ## 316013  47  <NA>         1
  ## 316464  55  <NA>         1
  ## 318522  28  <NA>         1
  ## 318700  85  <NA>         3
  ## 319625  47  <NA>         1
```

---

c. Le vecteur de poids de l'enquête (variable `extri1613`) présente des valeurs extrêmes relativement élevées. Afin d'éviter que les estimations ne soient trop affectées par quelques individus atypiques, on souhaite limiter le poids des individus en les « rabotant » à la valeur du 99ème percentile.

   i. Utilisez la fonction `quantile()` pour calculer le 99ème percentile de la distribution des poids.

```
# La fonction quantile() calcule les quantiles d'une distribution
quantile(eec$extri1613)
  ##         0%        25%        50%        75%       100%
  ##   156.3798  1088.2021  1334.5818  1636.7540 25347.1387

# Pour récupérer un quantile en particulier, on utilise l'argument
# probs
seuil <- quantile(eec$extri1613, probs = 0.99)
seuil
  ##      99%
  ## 3958.567
```

---

   ii. Récupérer la valeur du 99ème percentile et utilisez-la pour créer une nouvelle pondération (`newpond`) dans laquelle les poids ont été « rabotés » à son niveau.

```
# On utilise la valeur de seuil pour créer une version
# modifiée de la pondération (par exemple avec ifelse())
eec$newpond <- ifelse(
  eec$extri1613 > seuil
  , seuil
```

```
  , eec$extri1613
)

# Caractéristiques générales de newpond
summary(eec$newpond)
  ##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  ##   156.4  1088.2  1334.6  1419.0  1636.8  3958.6
```

Modifier la structure d'un `data.frame`

Comme pour les vecteurs ou les matrices, plusieurs opérations permettent de **modifier la structure d'un `data.frame`** :

— **trier un `data.frame` avec `order()`** : contrairement aux vecteurs, il n'est pas possible d'utiliser la fonction `sort()` pour trier un `data.frame`. En revanche, **la fonction `order()` renvoie la permutation permettant de trier une table selon une ou plusieurs variables**.

```
# Création de la table df6
df6 <- data.frame(
  var1 = letters[c(3, 4, 2, 5, 1, 5, 2, 4, 3, 1)]
  , var2 = rnorm(10))
df6
  ##    var1       var2
  ## 1     c -0.6264538
  ## 2     d  0.1836433
  ## 3     b -0.8356286
  ## 4     e  1.5952808
  ## 5     a  0.3295078
  ## 6     e -0.8204684
  ## 7     b  0.4874291
  ## 8     d  0.7383247
  ## 9     c  0.5757814
  ## 10    a -0.3053884

# Tri selon la variable var1
# - Etape 1 : obtention de la permutation correspondante
order(df6$var1)
  ##  [1]  5 10  3  7  1  9  2  8  4  6
# Utilisée sur le vecteur df6$var1, cette permutation
# renvoie un vecteur trié
df6$var1
```

```
  ##  [1] "c" "d" "b" "e" "a" "e" "b" "d" "c" "a"
order(df6$var1)
  ##  [1]  5 10  3  7  1  9  2  8  4  6
df6$var1[order(df6$var1)]
  ##  [1] "a" "a" "b" "b" "c" "c" "d" "d" "e" "e"

# - Etape 2 : utilisation de la permutation pour trier df6
df6[order(df6$var1), ]
  ##    var1       var2
  ## 5     a  0.3295078
  ## 10    a -0.3053884
  ## 3     b -0.8356286
  ## 7     b  0.4874291
  ## 1     c -0.6264538
  ## 9     c  0.5757814
  ## 2     d  0.1836433
  ## 8     d  0.7383247
  ## 4     e  1.5952808
  ## 6     e -0.8204684

# Tri selon la variable var1 puis la variable var2
# - Etape 1 : obtention de la permutation correspondante
order(df6$var1, df6$var2)
  ##  [1] 10  5  3  7  1  9  2  8  6  4
# - Etape 2 : utilisation de la permutation pour trier
df6[order(df6$var1, df6$var2), ]
  ##    var1       var2
  ## 10    a -0.3053884
  ## 5     a  0.3295078
  ## 3     b -0.8356286
  ## 7     b  0.4874291
  ## 1     c -0.6264538
  ## 9     c  0.5757814
  ## 2     d  0.1836433
  ## 8     d  0.7383247
  ## 6     e -0.8204684
  ## 4     e  1.5952808

# Tri selon la variable var1 puis les valeurs décroissantes
# de var2
df6 <- df6[order(df6$var1, - df6$var2), ]
df6
  ##    var1       var2
  ## 5     a  0.3295078
```

```
## 10    a -0.3053884
## 7     b  0.4874291
## 3     b -0.8356286
## 9     c  0.5757814
## 1     c -0.6264538
## 8     d  0.7383247
## 2     d  0.1836433
## 4     e  1.5952808
## 6     e -0.8204684
```

— ne sélectionner **que les valeurs distinctes pour certaines variables avec `unique()`** : la fonction `unique()` utilisée sur les vecteurs est également applicable aux `data.frame`.

```
# Ajout de la variable var3
df6$var3 <- rep(1:2, each = 5)
df6
##    var1       var2 var3
## 5     a  0.3295078    1
## 10    a -0.3053884    1
## 7     b  0.4874291    1
## 3     b -0.8356286    1
## 9     c  0.5757814    1
## 1     c -0.6264538    2
## 8     d  0.7383247    2
## 2     d  0.1836433    2
## 4     e  1.5952808    2
## 6     e -0.8204684    2

# Sélection de toutes les valeurs distinctes de var1 et var3
unique(df6[,c("var1", "var3")])
##    var1 var3
## 5     a    1
## 7     b    1
## 9     c    1
## 1     c    2
## 8     d    2
## 4     e    2
```

— **ajouter des lignes ou des colonnes à un `data.frame`** : les fonctions `cbind()` et `rbind()` utilisées avec les matrices sont également applicables aux `data.frame`.

```r
# Création du data.frame df7
df7 <- data.frame(
  var1 = c("f","f")
  , var2 = rnorm(2)
  , var3 = 3
)
df7
  ##   var1      var2 var3
  ## 1    f 1.5117812    3
  ## 2    f 0.3898432    3

# Création du data.frame df8 par concaténation des lignes
# de df6 et de df7
df8 <- rbind(df6, df7)
df8
  ##    var1       var2 var3
  ## 5     a  0.3295078    1
  ## 10    a -0.3053884    1
  ## 7     b  0.4874291    1
  ## 3     b -0.8356286    1
  ## 9     c  0.5757814    1
  ## 1     c -0.6264538    2
  ## 8     d  0.7383247    2
  ## 2     d  0.1836433    2
  ## 4     e  1.5952808    2
  ## 6     e -0.8204684    2
  ## 11    f  1.5117812    3
  ## 21    f  0.3898432    3

# Note : il faut que les deux data.frame aient exactement
# les mêmes variables avec le même nom pour que cela fonctionne
rbind(df6, df7[, c("var1", "var3")])
  ## Error in rbind(deparse.level, ...): les nombres de colonnes des arguments ne co
```

— **fusionner des données sur la base d'un identifiant** : la fonction `merge()` permet de fusionner deux `data.frame` (pas plus) sur la base d'un identifiant. À noter que **les tables n'ont pas besoin d'être triées au préalable**.

```r
# Création du data.frame df9
df9 <- data.frame(
  var3 = 2:4
  , var4 = c(TRUE, FALSE, TRUE)
)
```

```
df9
  ##   var3  var4
  ## 1    2  TRUE
  ## 2    3 FALSE
  ## 3    4  TRUE

# Fusion de df8 et de df9 selon la variable var3
merge(df8, df9, by = "var3")
  ##   var3 var1       var2  var4
  ## 1    2    c -0.6264538  TRUE
  ## 2    2    d  0.7383247  TRUE
  ## 3    2    d  0.1836433  TRUE
  ## 4    2    e  1.5952808  TRUE
  ## 5    2    e -0.8204684  TRUE
  ## 6    3    f  1.5117812 FALSE
  ## 7    3    f  0.3898432 FALSE
# Par défaut, merge() se restreint aux valeurs communes aux deux tables.

# Conservation de toutes les observations de df8
merge(df8, df9, by = "var3", all.x = TRUE)
  ##    var3 var1       var2  var4
  ## 1     1    a  0.3295078    NA
  ## 2     1    a -0.3053884    NA
  ## 3     1    b  0.4874291    NA
  ## 4     1    b -0.8356286    NA
  ## 5     1    c  0.5757814    NA
  ## 6     2    c -0.6264538  TRUE
  ## 7     2    d  0.7383247  TRUE
  ## 8     2    d  0.1836433  TRUE
  ## 9     2    e  1.5952808  TRUE
  ## 10    2    e -0.8204684  TRUE
  ## 11    3    f  1.5117812 FALSE
  ## 12    3    f  0.3898432 FALSE

# Conservation de toutes les observations de df9
merge(df8, df9, by = "var3", all.y = TRUE)
  ##   var3 var1       var2  var4
  ## 1    2    c -0.6264538  TRUE
  ## 2    2    d  0.7383247  TRUE
  ## 3    2    d  0.1836433  TRUE
  ## 4    2    e  1.5952808  TRUE
  ## 5    2    e -0.8204684  TRUE
  ## 6    3    f  1.5117812 FALSE
  ## 7    3    f  0.3898432 FALSE
```

```
  ## 8    4 <NA>          NA  TRUE
```

À noter qu'il peut y avoir **plusieurs variables de fusion** et qu'**il n'est pas indispensable qu'elles aient le même nom**.

```
# Création du data.frame df10
df10 <- data.frame(
  v1 = c("c","f")
  , v3 = c(2, 3)
  , v5 = c("Rouge", "Bleu")
)
df10
  ##   v1 v3    v5
  ## 1  c  2 Rouge
  ## 2  f  3  Bleu

# Fusion de df8 et de df10
merge(df8, df10, by.x = c("var3", "var1"), by.y = c("v3", "v1"), all = TRUE)
  ##    var3 var1       var2    v5
  ## 1     1    a  0.3295078  <NA>
  ## 2     1    a -0.3053884  <NA>
  ## 3     1    b  0.4874291  <NA>
  ## 4     1    b -0.8356286  <NA>
  ## 5     1    c  0.5757814  <NA>
  ## 6     2    c -0.6264538 Rouge
  ## 7     2    d  0.7383247  <NA>
  ## 8     2    d  0.1836433  <NA>
  ## 9     2    e -0.8204684  <NA>
  ## 10    2    e  1.5952808  <NA>
  ## 11    3    f  1.5117812  Bleu
  ## 12    3    f  0.3898432  Bleu
```

Cas pratique 3.3 Modifier la structure de données statistiques

a. À partir de la table `eec`, on souhaite produire une nouvelle table (`eec5`) qui ne comporte qu'un individu par ménage, le plus âgé. Les ménages sont identifiés par la variable `ident` et la variable `age` code l'âge des individus.

   i. Comment détermineriez-vous le nombre de ménages dans la table `eec` ?
   Indication Pensez à utiliser la fonction `unique()`.

```
# La fonction unique(), appliquée à un vecteur ou à une table,
# permet de déterminer des valeurs distinctes.
# Déterminer le nombre de valeurs distinctes revient ainsi
# à déterminer la longueur du vecteur ou le nombre de lignes
# de la table renvoyé par la fonction unique()
length(unique(eec$ident))
  ## [1] 18903
```

---

ii. On cherche d'abord à constituer une table ne comportant qu'un seul individu par ménage, quel que soit son âge. Comment procéderiez-vous ?
Indication Pensez à utiliser la fonction `duplicated()`.

```
# Pour un vecteur donné, la fonction duplicated() renvoie
# TRUE si l'élément figure déjà parmi les éléments du vecteur
# d'indice inférieur.
# Par exemple :
duplicated(c(1, 2, 3, 1, 1, 4, 3))
  ## [1] FALSE FALSE FALSE  TRUE  TRUE FALSE  TRUE
# Indique que les éléments en position 4, 5 et 8 sont en double
# (ils apparaissent déjà dans le vecteur).

# Cette fonction peut être utilisée pour ne sélectionner
# qu'un seul individu par ménage
str(!duplicated(eec$ident))
  ##  logi [1:34913] TRUE FALSE TRUE TRUE TRUE TRUE ...
# Note : On utilise ici str() afin que le vecteur logique
# ne s'affiche pas en entier (il compte plus de 30 000 éléments !)

# Il ne reste donc plus qu'à utiliser ce vecteur logique
# pour sélectionner les observations de eec.
eec5 <- eec[!duplicated(eec$ident), ]
head(eec5)
  ##           ident trim noi reg age sexe  cse dip11 acteu salred
  ## 315164 G0A56JP6    4  01  11  66    2   56    71     1    596
  ## 315166 G0A56JR6    4  01  11  27    2   38    10     1   2700
  ## 315167 G0A56JS6    4  01  11  29    2   37    11     1   2666
  ## 315168 G0A56JT6    4  01  11  31    1   37    11     1  11967
  ## 315169 G0A56JU6    4  01  11  24    2 <NA>    10     3     NA
  ## 315170 G0A56JV6    4  01  11  27    1   46    10     1   2000
  ##         stc tam1d aidref  tpp nbagenf duhab pub3fp naia
  ## 315164    2  <NA>   <NA>    1       0     7      4 1946
```

```
## 315166    2  <NA>   <NA>    1       0     7      4 1985
## 315167    2  <NA>   <NA>    1       0     7      4 1983
## 315168    2  <NA>   <NA>    1       0     7      4 1981
## 315169 <NA>  <NA>      5 <NA>       0  <NA>   <NA> 1988
## 315170    2  <NA>   <NA>    1       0     6      4 1985
##        extri1613   cs  cs2 acteu_red  newpond
## 315164  1776.563    5    5         1 1776.563
## 315166  2044.920    3    3         1 2044.920
## 315167  1897.519    3    3         1 1897.519
## 315168  1754.300    3    3         1 1754.300
## 315169  1643.808 <NA> <NA>         3 1643.808
## 315170  2191.007    4    4         1 2191.007
```

---

iii. Comment adapteriez-vous la réponse à la question précédente pour sélectionner l'individu le plus âgé du ménage (sans chercher à maîtriser celui qui est sélectionné quand plusieurs membres d'un même ménage ont le même âge) ?
Indication Pensez à trier judicieusement la table avec `order()`.

```
# duplicated() procède sur le vecteur des identifiant
# tel qu'il est trié dans la base eec

# En triant la base eec judicieusement, il est possible
# de déterminer quel individu du ménage est conservé dans
# la table eec5.

# En l'occurrence, il suffit de trier la table eec :
# - par ménage
# - PUIS par âge décroissant

# Pour mener à bien le tri, on utilise la fonction order() :
# - permutation pour trier par identifiant
str(order(eec$ident))
  ##  int [1:34913] 1 2 3 4 5 6 7 8 9 10 ...
# - permutation pour trier par identifiant puis par âge
str(order(eec$ident, eec$age))
  ##  int [1:34913] 2 1 3 4 5 6 8 7 9 10 ...
# - permutation pour trier par identifiant puis par âge décroissant
str(order(eec$ident, - eec$age))
  ## Error in -eec$age: argument incorrect pour un opérateur unitaire

# Note : On utilise ici str() afin que le vecteur logique
```

```
# ne s'affiche pas en entier (il compte plus de 30 000 éléments !)

# Le dernier appel de la fonction order() produit une erreur :
# la variable age étant de type caractère, l'opérateur -
# ne peut lui être appliqué.
# Pour ce faire, il suffit de convertir la variable
# age au préalable en vecteur de type numérique
str(order(eec$ident, - as.numeric(eec$age)))
  ##  int [1:34913] 1 2 3 4 5 6 7 8 9 10 ...

# On peut donc procéder au tri en tant que tel :
eec <- eec[order(eec$ident, - as.numeric(eec$age)), ]
eec[1:10, c("ident", "age")]
  ##           ident age
  ## 315164 G0A56JP6  66
  ## 315165 G0A56JP6  29
  ## 315166 G0A56JR6  27
  ## 315167 G0A56JS6  29
  ## 315168 G0A56JT6  31
  ## 315169 G0A56JU6  24
  ## 315170 G0A56JV6  27
  ## 315171 G0A56JV6  25
  ## 315172 G0A56JW6  35
  ## 315173 G0A56JX6  31
# Il semble bien que la table soit triée par ident puis par âge décroissant.

# Il ne reste plus qu'à utiliser la même méthode
# qu'à la question précédente :
eec5 <- eec[!duplicated(eec$ident), ]
```

---

iv. (Optionnel) Comment adapteriez-vous la réponse à la question précédente pour effectuer un tirage au sort à probabilités égales quand plusieurs membres d'un même ménage ont le même âge ?
Indication Pensez à utiliser une variable aléatoire (générée par exemple avec `rnorm()` ou `runif()`) dans la fonction `order()`.

```
# En l'état et quand plusieurs individus ont l'âge le plus élevé
# dans un ménage, l'individu sélectionné dans eec5 est celui
# qui était situé en premier dans la base eec.

# Néanmoins l'ordre de la base eec n'est pas aléatoire,
```

```
# et on peut souhaiter qu'en cas d'égalité l'individu
# sélectionné le soit aléatoirement.

# Une méthode très simple pour atteindre cet objectif consiste
# à ajouter une variable aléatoire comme ultime variable de tri.

# Pour ce faire, on utilise la fonction de génération
# de variables aléatoires rnorm(). Par exemple
rnorm(5)
  ## [1] -0.62124058 -2.21469989  1.12493092 -0.04493361 -0.01619026
# génère un vecteur de longueur 5 tiré dans une loi
# normale centrée réduite.

# Ici on va avoir besoin d'un vecteur de longueur
# le nombre de ligne de eec, aussi on va utiliser :
alea <- rnorm(nrow(eec))
str(alea)
  ##  num [1:34913] 0.944 0.821 0.594 0.919 0.782 ...

# On peut alors complèter la fonction order() de la sous-question
# précédente en ajoutant le vecteur alea comme troisième variable
# de tri :
eec <- eec[order(eec$ident, - as.numeric(eec$age), alea), ]
eec[1:10, c("ident", "age")]
  ##           ident age
  ## 315164 G0A56JP6  66
  ## 315165 G0A56JP6  29
  ## 315166 G0A56JR6  27
  ## 315167 G0A56JS6  29
  ## 315168 G0A56JT6  31
  ## 315169 G0A56JU6  24
  ## 315170 G0A56JV6  27
  ## 315171 G0A56JV6  25
  ## 315172 G0A56JW6  35
  ## 315174 G0A56JX6  31
eec6 <- eec[!duplicated(eec$ident), ]
```

---

b. Retour sur les PCS. Entre le niveau de la variable `cse` (niveau 3) et le niveau le plus agrégé de la variable `cs` créée dans le cas pratique 3.1 (niveau 1), il existe un niveau intermédiaire (niveau 2). La correspondance entre le niveau 3 et le niveau 2 n'est pas directe, et en règle générale on utilise la table de passage `pcs2003_c_n4_n1.dbf` (téléchargée depuis le site de l'Insee) pour la réaliser.

i. Utilisez le *package* `foreign` pour importer cette table dans R (*cf.* module 1). La nomenclature comporte quatre niveaux, mais le quatrième (variable `N4`) ne nous intéresse pas : agrégez la table de façon à ne conserver que les valeurs distinctes pour les niveaux 2 et 3 de la nomenclature.

Indication Pensez à utiliser la fonction `unique()`.

```
# La fonction read.dbf() du package foreign permet de
# facilement importer des fichiers .dbf sous forme de data.frame
library(foreign)
passage <- read.dbf("pcs2003_c_n4_n1.dbf")
str(passage)
  ## 'data.frame':  497 obs. of  4 variables:
  ##  $ N4: Factor w/ 497 levels "111a","111b",..: 1 2 3 4 5 6 7 8 9 10 ...
  ##  $ N3: int  11 11 11 11 11 11 12 12 12 12 ...
  ##  $ N2: int  10 10 10 10 10 10 10 10 10 10 ...
  ##  $ N1: int  1 1 1 1 1 1 1 1 1 1 ...
  ##  - attr(*, "data_types")= chr  "C" "N" "N" "N"

# Pour ne conserver que les valeurs distinctes pour
# les variables N2 et N3, il suffit d'utiliser
# la fonction unique sur le data.frame restreint
# aux variables N2 et N3 :
passage <- unique(passage[, c("N2", "N3")])
str(passage)
  ## 'data.frame':  42 obs. of  2 variables:
  ##  $ N2: int  10 10 10 21 22 23 31 32 32 32 ...
  ##  $ N3: int  11 12 13 21 22 23 31 33 34 35 ...
```

---

ii. Utilisez la fonction `merge()` pour fusionner cette table de passage avec le fichier `eec` et créer une nouvelle table (`eec6`) contenant une variable supplémentaire correspondant au niveau 2 de la PCS.

Indication Prenez garde au type des variables de fusion (qui doit être identique), ainsi qu'au nombre d'observations dans la table de départ et la table d'arrivée.

```
# La variable cse de la table eec est de type caractère, alors que
# N2 dans la table passage est de type numérique. On commence
# donc par convertir N2 en variable caractère
passage$N2 <- as.character(passage$N2)
```

```r
# On utilise alors la fonction merge() pour fusionner les tables
# eec et passage.
# Note : on utilise l'option all.x = TRUE pour conserver
# toutes les observations de eec, mais pas les observations
# de passage sans correspondance dans eec
eec6 <- merge(eec, passage, by.x = "cse", by.y = "N3", all.x = TRUE)

# Quand on compare les tris à plat de cse et N2, on constate
# que des valeurs manquantes sont apparues
table(eec6$cse, useNA = "always")
  ##
  ##    00    11    12    13    21    22    23    31    33    34
  ##     4   174    54   174   577   495    96   329   290   550
  ##    35    37    38    42    43    44    45    46    47    48
  ##   192   793   859   671  1028     7   340  1326   801   367
  ##    52    53    54    55    56    62    63    64    65    67
  ##  1679   312  1158   855  1426   885  1063   466   321   796
  ##    68    69    81  <NA>
  ##   581   191   336 15717
table(eec6$N2, useNA = "always")
  ##
  ##    10    21    22    23    31    32    36    41    46    47
  ##   402   577   495    96   329  1032  1652  2046  1326   801
  ##    48    51    54    55    56    61    66    69    81  <NA>
  ##   367  1991  1158   855  1426  2735  1377   191   336 15721
# En fait ce sont les 4 cas de 00 (absent de la table passage)
# qui sont conservés grâce à all.x = TRUE mais avec une valeur
# manquante à N2.
```

---

iii. (Difficile) À partir de la table de passage agrégée à la sous-question i., créez le vecteur `n2` dont les éléments sont les valeurs de la variable N2 et dont les noms sont les valeurs de la variable N3. Comment pourriez-vous utiliser ce vecteur pour obtenir le même résultat qu'à la question ii. ?
Indication Comparez `eec$cse[1:10]` et `n2[eec$cse[1:10]]`.

```r
# Construction du vecteur n2
n2 <- passage$N2
names(n2) <- passage$N3
str(n2)
  ##  Named chr [1:42] "10" "10" "10" "21" "22" "23" "31" "32" ...
  ##  - attr(*, "names")= chr [1:42] "11" "12" "13" "21" ...
```

```
n2[1:10]
  ##   11   12   13   21   22   23   31   33   34   35
  ## "10" "10" "10" "21" "22" "23" "31" "32" "32" "32"

# Remarque : on peut faire les deux opérations d'un seul
# coup avec la fonction setNames()
n2 <- setNames(passage$N2, passage$N3)

# Comme proposé, on compare eec$cse[1:10] et n2[eec$cse[1:10]]
eec$cse[1:10]
  ##  [1] "56" "81" "38" "37" "37" NA   "46" "37" "37" "38"
n2[eec$cse[1:10]]
  ##   56   81   38   37   37 <NA>   46   37   37   38
  ## "56" "81" "36" "36" "36"   NA "46" "36" "36" "36"
# Les deux vecteurs se ressemblent avec quelques différences :
# - le premier est un vecteur dont les éléments sont les valeurs
# de la PCS au niveau 3 pour les 10 premiers individus de la base
# - le second est un vecteur dont les éléments sont les noms sont
# les valeurs de la PCS au niveau 3 et dont les éléments sont les
# valeurs de la PCS au niveau 2 pour les 10 premiers individus
# de la base.

# Un exemple plus simple :
aConvertir <- c("b", "d", "a", "e", "a", "f")
aConvertir
  ## [1] "b" "d" "a" "e" "a" "f"
corresp <- setNames(rep(1:3, each = 2), letters[1:6])
corresp
  ## a b c d e f
  ## 1 1 2 2 3 3
corresp[aConvertir]
  ## b d a e a f
  ## 1 2 1 3 1 3

# Pour revenir à la question : il suffit alors d'utiliser
# le vecteur n2[eec6$cse] pour créer directement la variable
# convertie dans la table eec6
eec6$N2_bis <- n2[eec6$cse]
identical(eec6$N2, eec6$N2_bis)
  ## [1] TRUE

# La méthode proposée au iii. est beaucoup plus rapide
# que celle proposée au ii.
system.time({
```

```
  merge(eec, passage, by.x = "cse", by.y = "N3", all.x = TRUE)
})
  ## utilisateur     système      écoulé
  ##       0.092       0.004       0.098
system.time({
  n2[na.omit(eec$cse)]
})
  ## utilisateur     système      écoulé
  ##       0.000       0.000       0.002

# Remarque : la fonction na.omit() supprime automatiquement les NA
# dans eec$cse, ce qui accélère considérablement la seconde
# méthode. D'un point de vue opérationnel, on l'utiliserait
# de la façon suivante :
eec6$N2_ter[!is.na(eec6$cse)] <- n2[na.omit(eec6$cse)]
identical(eec6$N2, eec6$N2_ter)
  ## [1] TRUE
```

Effectuer des calculs sur un data.frame

La proximité des data.frame à la fois avec les matrices et les listes se retrouve dans le **type d'opérations qu'il est possible de leur appliquer** :

— comme avec les matrices, il est possible d'utiliser les **fonctions colSums() ou rowSums()** ainsi que la **fonction apply()** ;

```
df11 <- data.frame(
  var1 = 1:5
  , var2 = 11:15
  , var3 = 21:25
)
df11
  ##   var1 var2 var3
  ## 1    1   11   21
  ## 2    2   12   22
  ## 3    3   13   23
  ## 4    4   14   24
  ## 5    5   15   25

# rowSums(), colSums(), apply() : comme une matrice
rowSums(df11)
  ## [1] 33 36 39 42 45
```

```
colSums(df11)
  ## var1 var2 var3
  ##   15   65  115
apply(df11, 1, max)
  ## [1] 21 22 23 24 25
apply(df11, 2, min)
  ## var1 var2 var3
  ##    1   11   21
```

— comme avec les listes, il est possible d'utiliser les **fonctions `lapply()` et `sapply()`**, qui s'appliquent **colonne par colonne**.

```
# lapply(), sapply() : comme une liste
lapply(df11, sum)
  ## $var1
  ## [1] 15
  ##
  ## $var2
  ## [1] 65
  ##
  ## $var3
  ## [1] 115
sapply(df11, mean)
  ## var1 var2 var3
  ##    3   13   23
```

Une des opérations les plus utiles consiste à **appliquer une même fonction à des groupes d'observations définis par les modalités d'une autre variables** (comme avec une **instruction BY dans SAS**).

Exemple Âge moyen par région, salaire moyen par sexe, etc.

Plusieurs fonctions de R permettent de mener à bien ce type d'opération :

— la **fonction `aggregate()`** ;

```
df6
  ##    var1       var2 var3
  ## 5     a  0.3295078    1
  ## 10    a -0.3053884    1
  ## 7     b  0.4874291    1
  ## 3     b -0.8356286    1
  ## 9     c  0.5757814    1
  ## 1     c -0.6264538    2
  ## 8     d  0.7383247    2
  ## 2     d  0.1836433    2
  ## 4     e  1.5952808    2
  ## 6     e -0.8204684    2
```

```
# On souhaite calculer la moyenne de var2
# selon les modalités de var3

aggregate(df6$var2, list(df6$var1), mean)
  ##   Group.1           x
  ## 1       a  0.01205969
  ## 2       b -0.17409978
  ## 3       c -0.02533623
  ## 4       d  0.46098401
  ## 5       e  0.38740621
```

— la **fonction `tapply()`** ;

```
tapply(df6$var2, df6$var1, mean)
  ##           a           b           c           d           e
  ##  0.01205969 -0.17409978 -0.02533623  0.46098401  0.38740621
```

— la **fonction `split()` combinée à un `lapply()` ou un `sapply()`**.

```
# La fonction split(x, f) "éclate" le data.frame x en une
# liste de data.frame selon les modalités du factor f
split(df6, df6$var1)
  ## $a
  ##    var1       var2 var3
  ## 5     a  0.3295078    1
  ## 10    a -0.3053884    1
  ##
  ## $b
  ##   var1       var2 var3
  ## 7    b  0.4874291    1
  ## 3    b -0.8356286    1
  ##
  ## $c
  ##   var1       var2 var3
  ## 9    c  0.5757814    1
  ## 1    c -0.6264538    2
  ##
  ## $d
  ##   var1      var2 var3
  ## 8    d 0.7383247    2
  ## 2    d 0.1836433    2
  ##
  ## $e
  ##   var1       var2 var3
  ## 4    e  1.5952808    2
  ## 6    e -0.8204684    2

# Il ne reste alors plus qu'à appliquer
```

```r
# à chaque élément de la liste ainsi produite
# la fonction souhaitée par le biais d'un sapply()
sapply(split(df6, df6$var1), function(x) mean(x$var2))
```

```
##           a           b           c           d           e
##  0.01205969 -0.17409978 -0.02533623  0.46098401  0.38740621
```

Remarques

1. La fonction `by()` utilisée lors du module 1 fait en fait appel à la fonction `tapply()`.
2. Les performances de ces méthodes diffèrent sensiblement :

```r
# Installation et chargement de la bibliothèque de test
# de performance microbenchmark
# install.packages("microbenchmark")
library(microbenchmark)

# Compararison des trois méthodes + variante optimisée de sapply()
microbenchmark(times = 1000
  , aggregate = aggregate(df6$var2, list(df6$var1), mean)
  , sapply = sapply(split(df6, df6$var1), function(x) mean(x$var2))
  , tapply = tapply(df6$var2, df6$var1, mean)
  , sapply2 = sapply(split(df6$var2, df6$var1), mean)
)
```

```
## Unit: microseconds
##       expr     min       lq      mean    median        uq
##  aggregate 686.342 863.6960 1573.0221 1277.2875 1933.3015
##     sapply 334.523 405.0455  749.9811  561.0135  864.7185
##     tapply 110.876 139.9495  260.1750  187.6270  301.7605
##    sapply2  88.044 109.6945  209.8148  136.0720  225.1740
##        max neval
##  17538.110  1000
##   9816.966  1000
##   6087.540  1000
##   5685.734  1000
```

3. Le *package* `sqldf` permet d'utiliser le langage SQL dans R (à l'image de la PROC SQL dans SAS), aussi bien pour des agrégations (par groupe notamment) que pour des fusions.

Cas pratique 3.4 Effectuer des manipulations complexes sur des données statistiques

a. On souhaite calculer le taux de chômage au niveau national et régional. Le taux de chômage est défini par le ratio du nombre total d'individus au chômage (`acteu == "2"`) sur la taille de la population active (`acteu == %in% c("1", "2")`).

Remarque Le fichier utilisé ici ne comporte que les logements en première ou sixième interrogation : les estimations effectuées dans ce cas pratique n'ont donc aucune raison de coïncider avec les estimations officielles (qui par ailleurs sont CVS-CJO).

---

i. Calculez le taux de chômage national, d'abord non-pondéré puis pondéré par la variable `extri1613`.

```
# Taux de chômage non-pondéré
sum(eec$acteu %in% "2", na.rm = TRUE) /
  sum(eec$acteu %in% c("1", "2"), na.rm = TRUE)
  ## [1] 0.1061382
# Note : On utilise eec$acteu %in% "2" plutôt que
# eec$acteu == "2" car les deux opérateurs ne traitent
# pas de façon identique les valeurs manquantes quand
# il y en a.

# Taux de chômage pondéré
with(eec, {
  sum((acteu %in% "2") * extri1613, na.rm = TRUE) /
    sum((acteu %in% c("1", "2")) * extri1613, na.rm = TRUE)
})
  ## [1] 0.1088602
# Note : la fonction with() permet d'alléger l'écriture
# en ne répétant pas le nom de la table (sur la différence
# avec within(), cf. l'aide)
```

---

ii. Utilisez les fonctions `aggregate()`, `tapply()` et `sapply()` (avec `split()` dans le dernier cas) pour calculer un taux de chômage non-pondéré et par région (variable `reg`).

```r
# Les trois fonctions mentionnées en consigne permettent
# d'appliquer le même traitement à différents groupes
# d'observations définis selon les modalités d'une variable
# qualitative (ici la variable de région reg)

# Avec aggregate()
aggregate(eec$acteu, list(eec$reg), function(x){
  sum(x %in% "2", na.rm = TRUE) /
    sum(x %in% c("1", "2"), na.rm = TRUE)
})
  ##    Group.1          x
  ## 1       11 0.09817352
  ## 2       21 0.10631741
  ## 3       22 0.13063764
  ## 4       23 0.13454545
  ## 5       24 0.11756168
  ## 6       25 0.10245902
  ## 7       26 0.08602151
  ## 8       31 0.13998324
  ## 9       41 0.15952733
  ## 10      42 0.09769335
  ## 11      43 0.10873147
  ## 12      52 0.09057301
  ## 13      53 0.08740602
  ## 14      54 0.11594203
  ## 15      72 0.08893485
  ## 16      73 0.06815642
  ## 17      74 0.06648199
  ## 18      82 0.08660508
  ## 19      83 0.10120482
  ## 20      91 0.18770227
  ## 21      93 0.11396226
  ## 22      94 0.11764706

# Avec tapply()
tapply(eec$acteu, eec$reg, function(x){
  sum(x %in% "2", na.rm = TRUE) /
    sum(x %in% c("1", "2"), na.rm = TRUE)
})
  ##         11         21         22         23         24
  ## 0.09817352 0.10631741 0.13063764 0.13454545 0.11756168
  ##         25         26         31         41         42
  ## 0.10245902 0.08602151 0.13998324 0.15952733 0.09769335
  ##         43         52         53         54         72
```

```
  ## 0.10873147 0.09057301 0.08740602 0.11594203 0.08893485
  ##         73         74         82         83         91
  ## 0.06815642 0.06648199 0.08660508 0.10120482 0.18770227
  ##         93         94
  ## 0.11396226 0.11764706

# Avec sapply() et split()
sapply(split(eec, eec$reg), function(x){
  sum(x$acteu %in% "2", na.rm = TRUE) /
    sum(x$acteu %in% c("1", "2"), na.rm = TRUE)
})
  ##         11         21         22         23         24
  ## 0.09817352 0.10631741 0.13063764 0.13454545 0.11756168
  ##         25         26         31         41         42
  ## 0.10245902 0.08602151 0.13998324 0.15952733 0.09769335
  ##         43         52         53         54         72
  ## 0.10873147 0.09057301 0.08740602 0.11594203 0.08893485
  ##         73         74         82         83         91
  ## 0.06815642 0.06648199 0.08660508 0.10120482 0.18770227
  ##         93         94
  ## 0.11396226 0.11764706
```

---

iii. Utilisez la fonction `sapply()` avec `split()` pour calculer un taux de chômage pondéré et par région.

```
# Pour pondérer le calcul du taux de chômage, il faut
# pouvoir appliquer la fonction de calcul à deux éléments :
# la valeur de la variable acteu et la pondération extri1613.

# Or les fonctions aggregate() et tapply() ne portent
# que sur des vecteurs et ne conviennent donc pas.
# On se limite donc à la méthode avec sapply() et split().
sapply(split(eec, eec$reg), function(x){
  sum((x$acteu %in% "2") * x$extri1613, na.rm = TRUE) /
    sum((x$acteu %in% c("1", "2")) * x$extri1613, na.rm = TRUE)
})
  ##         11         21         22         23         24
  ## 0.09799410 0.10902296 0.13996041 0.13407322 0.11443175
  ##         25         26         31         41         42
  ## 0.09876409 0.10016586 0.14881667 0.16226981 0.09831550
  ##         43         52         53         54         72
```

```
## 0.11146691 0.08739635 0.08845711 0.11341163 0.08924882
##         73         74         82         83         91
## 0.06535967 0.06359372 0.09318913 0.12422405 0.19206955
##         93         94
## 0.11484820 0.10609290
```

---

b. Pour des raisons de stockage et de performances, on souhaite optimiser le type des variables de l'objet `eec`. En effet, R manipule beaucoup plus efficacement les variables de type numérique que les variables de type caractère.

i. Déterminer sous la forme d'un vecteur logique quelles variables de l'objet `eec` sont de type caractère.
Indication Utilisez la fonction `is.character()` dans une structure `*apply()`.

```
# Pour tester le fait qu'une variable est de type caractère,
# il suffit d'utiliser la fonction is.character()
is.character(eec$trim)
  ## [1] TRUE

# Pour appliquer cette fonction à l'ensemble des colonnes
# de l'objet eec et récupérer un vecteur en sortie,
# il suffit d'utiliser la fonction sapply()
sapply(eec, is.character)
  ##     ident      trim       noi       reg       age      sexe
  ##      TRUE      TRUE      TRUE      TRUE      TRUE      TRUE
  ##       cse     dip11     acteu    salred       stc     tam1d
  ##      TRUE      TRUE      TRUE     FALSE      TRUE      TRUE
  ##    aidref       tpp   nbagenf     duhab    pub3fp      naia
  ##      TRUE      TRUE      TRUE      TRUE      TRUE      TRUE
  ## extri1613        cs       cs2 acteu_red   newpond
  ##     FALSE      TRUE      TRUE      TRUE     FALSE

# On stocke ce résultat pour le réutiliser par la suite :
varChar <- sapply(eec, is.character)
```

---

ii. Créez la table `eec7` dans lequel toutes les variables de type caractère de `eec` à l'exception de ident et noi sont converties en variables de type numérique.

```
# Construction du vecteur logique des variables
# à convertir en numérique (varChar privé de ident et noi)
toNum <- varChar
toNum[c("ident", "noi")] <- FALSE
toNum
  ##     ident      trim       noi       reg       age      sexe
  ##     FALSE      TRUE     FALSE      TRUE      TRUE      TRUE
  ##       cse     dip11     acteu    salred       stc     tam1d
  ##      TRUE      TRUE      TRUE     FALSE      TRUE      TRUE
  ##    aidref       tpp   nbagenf     duhab    pub3fp      naia
  ##      TRUE      TRUE      TRUE      TRUE      TRUE      TRUE
  ## extri1613        cs       cs2 acteu_red   newpond
  ##     FALSE      TRUE      TRUE      TRUE     FALSE

# Création de l'objet eec7
eec7 <- eec
eec7[toNum] <- lapply(eec7[toNum], as.numeric)
str(eec7)
  ## 'data.frame':  34913 obs. of  23 variables:
  ##  $ ident    : chr  "G0A56JP6" "G0A56JP6" "G0A56JR6" "G0A56JS6" ...
  ##  $ trim     : num  4 4 4 4 4 4 4 4 4 4 ...
  ##  $ noi      : chr  "01" "02" "01" "01" ...
  ##  $ reg      : num  11 11 11 11 11 11 11 11 11 11 ...
  ##  $ age      : num  66 29 27 29 31 24 27 25 35 31 ...
  ##  $ sexe     : num  2 1 2 2 1 2 1 2 1 2 ...
  ##  $ cse      : num  56 81 38 37 37 NA 46 37 37 38 ...
  ##  $ dip11    : num  71 42 10 11 11 10 10 10 11 11 ...
  ##  $ acteu    : num  1 2 1 1 1 3 1 1 1 1 ...
  ##  $ salred   : int  596 NA 2700 2666 11967 NA 2000 2800 2333 2492 ...
  ##  $ stc      : num  2 NA 2 2 2 NA 2 2 2 2 ...
  ##  $ tam1d    : num  NA NA NA NA NA NA NA NA NA NA ...
  ##  $ aidref   : num  NA 5 NA NA NA 5 NA NA NA NA ...
  ##  $ tpp      : num  1 NA 1 1 1 NA 1 1 1 1 ...
  ##  $ nbagenf  : num  0 0 0 0 0 0 0 0 0 0 ...
  ##  $ duhab    : num  7 NA 7 7 7 NA 6 7 7 7 ...
  ##  $ pub3fp   : num  4 NA 4 4 4 NA 4 4 4 4 ...
  ##  $ naia     : num  1946 1983 1985 1983 1981 ...
  ##  $ extri1613: num  1777 1777 2045 1898 1754 ...
  ##  $ cs       : num  5 8 3 3 3 NA 4 3 3 3 ...
  ##  $ cs2      : num  5 8 3 3 3 NA 4 3 3 3 ...
  ##  $ acteu_red: num  1 2 1 1 1 3 1 1 1 1 ...
  ##  $ newpond  : num  1777 1777 2045 1898 1754 ...
```

iii. Pour chaque variable numérique de `eec7`, testez si la conversion en nombre entier (grâce à la fonction `as.integer()`) est sans perte. Quand c'est le cas, convertissez la variable en nombre entier.

```
# On procède de façon analogue à précédemment :

# 1) Identification des variables numériques avec un sapply()
varNum <- sapply(eec7, is.numeric)

# 2) Pour déterminer si la conversion en entier est sans perte,
# on teste l'identité par double-conversion. Par exemple :
identical(eec7$acteu, as.numeric(as.integer(eec7$acteu)))
  ## [1] TRUE

# 3) On définit alors la fonction convertirSiSansPerte() :
convertirSiSansPerte <- function(x){
  if(identical(x, as.numeric(as.integer(x)))){
    as.integer(x)
  }else{
    x
  }
}

# 4) Il ne reste plus qu'à appliquer cette fonction
# à toutes les variables numériques de eec7 via un lapply()
eec7[varNum] <- lapply(eec7[varNum], convertirSiSansPerte)
str(eec7)
  ## 'data.frame':  34913 obs. of  23 variables:
  ##  $ ident    : chr  "G0A56JP6" "G0A56JP6" "G0A56JR6" "G0A56JS6" ...
  ##  $ trim     : int  4 4 4 4 4 4 4 4 4 4 ...
  ##  $ noi      : chr  "01" "02" "01" "01" ...
  ##  $ reg      : int  11 11 11 11 11 11 11 11 11 11 ...
  ##  $ age      : int  66 29 27 29 31 24 27 25 35 31 ...
  ##  $ sexe     : int  2 1 2 2 1 2 1 2 1 2 ...
  ##  $ cse      : int  56 81 38 37 37 NA 46 37 37 38 ...
  ##  $ dip11    : int  71 42 10 11 11 10 10 10 11 11 ...
  ##  $ acteu    : int  1 2 1 1 1 3 1 1 1 1 ...
  ##  $ salred   : int  596 NA 2700 2666 11967 NA 2000 2800 2333 2492 ...
  ##  $ stc      : int  2 NA 2 2 2 NA 2 2 2 2 ...
  ##  $ tam1d    : int  NA NA NA NA NA NA NA NA NA NA ...
  ##  $ aidref   : int  NA 5 NA NA NA 5 NA NA NA NA ...
  ##  $ tpp      : int  1 NA 1 1 1 NA 1 1 1 1 ...
  ##  $ nbagenf  : int  0 0 0 0 0 0 0 0 0 0 ...
  ##  $ duhab    : int  7 NA 7 7 7 NA 6 7 7 7 ...
```

```
  ##  $ pub3fp   : int  4 NA 4 4 4 NA 4 4 4 4 ...
  ##  $ naia     : int  1946 1983 1985 1983 1981 1988 1985 1987 1977 1981 .
  ##  $ extri1613: num  1777 1777 2045 1898 1754 ...
  ##  $ cs       : int  5 8 3 3 3 NA 4 3 3 3 ...
  ##  $ cs2      : int  5 8 3 3 3 NA 4 3 3 3 ...
  ##  $ acteu_red: int  1 2 1 1 1 3 1 1 1 1 ...
  ##  $ newpond  : num  1777 1777 2045 1898 1754 ...

# Quelques comparaisons

# - taille des objets avec le package pryr
# install.packages("pryr")
library(pryr)
object_size(eec)
  ## 7.5 MB
object_size(eec7)
  ## 4.97 MB
```

```
# - vitesse d'exécution avec microbenchmark()
# install.packages("microbenchmark")
library(microbenchmark)
microbenchmark(times = 1000
  , eec = sum(eec$acteu == "2", na.rm = TRUE)
  , eec7 = sum(eec7$acteu == 2, na.rm = TRUE)
)
```

```
  ## Unit: microseconds
  ##  expr     min       lq     mean   median       uq      max neval
  ##   eec 203.749 228.7015 396.2294 331.2155 411.9825 7189.347  1000
  ##  eec7 112.163 127.5090 256.5256 185.7500 235.8195 6438.663  1000
```

Calculer des statistiques descriptives

La plupart des fonctions permettant de calculer des statistiques descriptives ont été présentées tout au long de la formation : `table()`, `summary()`, etc. **Cette partie revient sur l'utilisation de ces fonctions dans une perspective proprement statistique, en élargissant leur utilisation au cas des données pondérées**.

L'ensemble des éléments introduits dans cette partie sont mis en pratique sur les données de l'enquête Pisa 2012 (*cf.* dernière sous-partie).

Variables qualitatives

La fonction `table()` calcule les **fréquences** (non-pondérées) des modalités ou des croisements de modalités d'une ou plusieurs variables qualitatives.

```
# Fréquences des modalités de la variable pub3fp
# Signification des modalités :
# 1 : Fonction publique d'Etat
# 2 : Fonction publique territoriale
# 3 : Fonction publique hospitalitère
# 4 : Secteur privé
table(eec$pub3fp)
  ##
  ##     1     2     3     4
  ##  1415  1246   757 11701

# Utilisation de l'argument useNA pour afficher les valeurs manquantes
table(eec$pub3fp, useNA = "always")
  ##
  ##     1     2     3     4  <NA>
  ##  1415  1246   757 11701 19794

# Croisement avec le sexe
table(eec$pub3fp, eec$sexe, useNA = "always")
  ##
  ##            1     2  <NA>
  ##   1      633   782     0
  ##   2      452   794     0
  ##   3      164   593     0
  ##   4     6234  5467     0
  ##   <NA>  9099 10695     0
```

Pour améliorer l'affichage des résultats de la fonction `table()`, le plus simple est de **transformer les variables caractères utilisées en facteurs**, au préalable ou directement dans la fonction `table()`.

```
# Transformation de pub3fp en factor
eec$pub3fp <- factor(eec$pub3fp, labels = c(
  "Fonction publique d'Etat"
  , "Fonction publique territoriale"
  , "Fonction publique hospitalière"
  , "Secteur privé"
))

# Impact sur l'affichage de table()
table(eec$pub3fp, eec$sexe, useNA = "always")
```

```
##
##                                      1     2  <NA>
##   Fonction publique d'Etat         633   782     0
##   Fonction publique territoriale   452   794     0
##   Fonction publique hospitalière   164   593     0
##   Secteur privé                   6234  5467     0
##   <NA>                            9099 10695     0

# Tranformation à la volée de eec$sexe en factor
table(eec$pub3fp, factor(eec$sexe, labels = c("Homme","Femme")), useNA = "always")
##
##                                    Homme Femme  <NA>
##   Fonction publique d'Etat           633   782     0
##   Fonction publique territoriale     452   794     0
##   Fonction publique hospitalière     164   593     0
##   Secteur privé                     6234  5467     0
##   <NA>                              9099 10695     0
```

Les fonctions `addmargins()` et `prop.table()` permettent d'ajouter les marges et de calculer des pourcentages respectivement.

```
t <- table(eec$pub3fp, eec$sexe, useNA = "always")

# Ajout de marges avec la fonction addmargins()
addmargins(t)
##
##                                      1     2  <NA>   Sum
##   Fonction publique d'Etat         633   782     0  1415
##   Fonction publique territoriale   452   794     0  1246
##   Fonction publique hospitalière   164   593     0   757
##   Secteur privé                   6234  5467     0 11701
##   <NA>                            9099 10695     0 19794
##   Sum                            16582 18331     0 34913

# Calcul de pourcentages
prop.table(t) # Pourcentages de cellule
##
##                                           1           2
##   Fonction publique d'Etat       0.018130782 0.022398533
##   Fonction publique territoriale 0.012946467 0.022742245
##   Fonction publique hospitalière 0.004697391 0.016985077
##   Secteur privé                  0.178558130 0.156589236
##   <NA>                           0.260619254 0.306332885
##
```

```
  ##                                          <NA>
  ##   Fonction publique d'Etat       0.000000000
  ##   Fonction publique territoriale 0.000000000
  ##   Fonction publique hospitalière 0.000000000
  ##   Secteur privé                  0.000000000
  ##   <NA>                           0.000000000
prop.table(t, 1) # Pourcentages en ligne
  ##
  ##                                           1         2      <NA>
  ##   Fonction publique d'Etat       0.4473498 0.5526502 0.0000000
  ##   Fonction publique territoriale 0.3627608 0.6372392 0.0000000
  ##   Fonction publique hospitalière 0.2166446 0.7833554 0.0000000
  ##   Secteur privé                  0.5327750 0.4672250 0.0000000
  ##   <NA>                           0.4596848 0.5403152 0.0000000
prop.table(t, 2) # Pourcentages en colonne
  ##
  ##                                             1           2 <NA>
  ##   Fonction publique d'Etat       0.038173924 0.042659975
  ##   Fonction publique territoriale 0.027258473 0.043314604
  ##   Fonction publique hospitalière 0.009890242 0.032349572
  ##   Secteur privé                  0.375949825 0.298237958
  ##   <NA>                           0.548727536 0.583437892
```

La fonction `chisq.test()` mène le test d'indépendance du χ^2.

```
# Test du chi2 sur le lien entre eec$pub3fp et eec$sexe
chisq.test(eec$pub3fp, eec$sexe)
  ##
  ##    Pearson's Chi-squared test
  ##
  ## data:  eec$pub3fp and eec$sexe
  ## X-squared = 401.45, df = 3, p-value < 2.2e-16
```

Au-delà de ces fonctions natives, **le *package* `descr` facilite considérablement l'analyse uni- et bivariée de variables qualitatives**, en particulier quand les données ont à être pondérées (données d'enquête).

```
# Installation du package descr
# install.packages("desc")
```

```
# Chargement du package descr
library(descr)
```

La **fonction `freq()`** présente les résultats d'un tri à plat de façon plus complète et plus naturelle et son argument `w` permet de pondérer les calculs.

```
# Tri à plat non-pondéré sur la variable eec$pub3fp
freq(eec$pub3fp)
  ## eec$pub3fp
  ##                                  Frequency Percent Valid Percent
  ## Fonction publique d'Etat              1415   4.053         9.359
  ## Fonction publique territoriale       1246   3.569         8.241
  ## Fonction publique hospitalière        757   2.168         5.007
  ## Secteur privé                       11701  33.515        77.393
  ## NA's                                19794  56.695
  ## Total                               34913 100.000       100.000
```

```
# Tri à plat pondéré sur la variable eec$pub3fp
freq(eec$pub3fp, w = eec$extri1613)
  ## eec$pub3fp
  ##                                  Frequency Percent Valid Percent
  ## Fonction publique d'Etat           2148336   4.254         9.453
  ## Fonction publique territoriale     1816318   3.596         7.992
  ## Fonction publique hospitalière     1095084   2.168         4.819
  ## Secteur privé                     17666632  34.981        77.736
  ## NA's                              27777229  55.000
  ## Total                             50503600 100.000       100.000
```

De même, la **fonction `crosstab()`** simplifie l'interprétation d'un tri croisé et l'utilisation de pondérations.

```
# Tri croisé non-pondéré des variables eec$pub3fp et eec$sexe
crosstab(eec$pub3fp, eec$sexe)
  ##    Cell Contents
  ## |-------------------------|
  ## |                   Count |
  ## |-------------------------|
  ##
  ## ======================================================
  ##                                    eec$sexe
  ## eec$pub3fp                            1      2   Total
  ## ------------------------------------------------------
  ## Fonction publique d'Etat            633    782    1415
  ## ------------------------------------------------------
  ## Fonction publique territoriale      452    794    1246
  ## ------------------------------------------------------
```

```
  ## Fonction publique hospitalière     164    593     757
  ## ----------------------------------------------------
  ## Secteur privé                     6234   5467   1.17e+04
  ## ----------------------------------------------------
  ## Total                             7483   7636   1.512e+04
  ## ====================================================

# Tri croisé pondéré des variables eec$pub3fp et eec$sexe
crosstab(eec$pub3fp, eec$sexe, w = eec$extri1613)
  ##    Cell Contents
  ## |-------------------------|
  ## |                   Count |
  ## |-------------------------|
  ##
  ## ====================================================================
  ##                                   eec$sexe
  ## eec$pub3fp                               1          2      Total
  ## --------------------------------------------------------------------
  ## Fonction publique d'Etat          9.712e+05   1.177e+06   2.148e+06
  ## --------------------------------------------------------------------
  ## Fonction publique territoriale    6.663e+05   1.15e+06   1.816e+06
  ## --------------------------------------------------------------------
  ## Fonction publique hospitalière    2.426e+05   8.525e+05   1.095e+06
  ## --------------------------------------------------------------------
  ## Secteur privé                     9.539e+06   8.127e+06   1.767e+07
  ## --------------------------------------------------------------------
  ## Total                             1.142e+07   1.131e+07   2.273e+07
  ## ====================================================================

# Ajout des pourcentages en ligne et en colonne
crosstab(eec$pub3fp, eec$sexe, w = eec$extri1613, prop.r = TRUE, prop.c = TRUE)
  ##    Cell Contents
  ## |-------------------------|
  ## |                   Count |
  ## |             Row Percent |
  ## |          Column Percent |
  ## |-------------------------|
  ##
  ## ====================================================================
  ##                                   eec$sexe
  ## eec$pub3fp                               1          2      Total
  ## --------------------------------------------------------------------
  ## Fonction publique d'Etat           971236    1177100    2148336
  ##                                      45.2%      54.8%       9.5%
```

```
  ##                                          8.5%      10.4%
  ## ---------------------------------------------------------------
  ## Fonction publique territoriale     666324    1149994    1816318
  ##                                       36.7%      63.3%       8.0%
  ##                                        5.8%      10.2%
  ## ---------------------------------------------------------------
  ## Fonction publique hospitalière     242581     852503    1095084
  ##                                       22.2%      77.8%       4.8%
  ##                                        2.1%       7.5%
  ## ---------------------------------------------------------------
  ## Secteur privé                     9539319    8127313   17666632
  ##                                       54.0%      46.0%      77.7%
  ##                                       83.5%      71.9%
  ## ---------------------------------------------------------------
  ## Total                            11419460   11306910   22726370
  ##                                       50.2%      49.8%
  ## ===============================================================
```

```
# Test du chi2
crosstab(
  eec$pub3fp, eec$sexe, w = eec$extri1613 / mean(eec$extri1613)
  , prop.chisq = TRUE,  chisq = TRUE
)
```

```
  ##    Cell Contents
  ## |-------------------------|
  ## |                   Count |
  ## | Chi-square contribution |
  ## |-------------------------|
  ##
  ## ============================================================
  ##                                    eec$sexe
  ## eec$pub3fp                              1         2   Total
  ## ------------------------------------------------------------
  ## Fonction publique d'Etat              671       814    1485
  ##                                     7.585     7.662
  ## ------------------------------------------------------------
  ## Fonction publique territoriale        461       795    1256
  ##                                    45.874    46.338
  ## ------------------------------------------------------------
  ## Fonction publique hospitalière        168       589     757
  ##                                   118.598   119.797
  ## ------------------------------------------------------------
  ## Secteur privé                        6595      5618   12213
  ##                                    34.148    34.494
```

```
## ------------------------------------------------------------
## Total                                  7895      7816   15711
## ============================================================
##
## Statistics for All Table Factors
##
## Pearson's Chi-squared test
## ------------------------------------------------------------
## Chi^2 = 414.4949      d.f. = 3      p <2e-16
##
##         Minimum expected frequency: 376.5968
```

Variables quantitatives

Contrairement à d'autres logiciels statistiques (SAS tout particulièrement), **R ne possède pas une procédure permettant de calculer automatiquement l'ensemble des statistiques descriptives standards** dans le cas d'une variable de nature quantitative, mais un **ensemble de fonctions élémentaires** (*cf.* tableau).

Code R	Résultat
`sum(x)`	Somme de `x`
`mean(x)`	Moyenne de `x`
`var(x)`	Variance empirique de `x`
`sd(x)`	Écart-type empirique de `x`
`quantile(x)`	Quantiles de `x`
`summary(x)`	Moyenne et quantiles de `x`
`max(x)`	Valeur maximum de `x`
`min(x)`	Valeur minimum de `x`
`range(x)`	Valeur minimale et valeur maximale de `x`
`cor.test(x, y)`	Corrélation entre `x` et `y`

En présence de valeurs manquantes (`NA`), la plupart de ces fonctions renvoient la valeur `NA` : l'argument `na.rm = TRUE` permet de modifier ce comportement.

```
# Statistiques descriptives standards sur le salaire dans l'EEC

mean(eec$salred)
  ## [1] NA
# Il y a manifestement des valeurs manquantes
sum(is.na(eec$salred))
  ## [1] 19794
```

```
# Les valeurs manquantes correspondent à 19 794 observations
# sur 34 913, ce qui est logique : ni les inactifs ni les non-
# salariés ne touchent de salaire.

mean(eec$salred, na.rm = TRUE)
  ## [1] 1819.209
sd(eec$salred, na.rm = TRUE)
  ## [1] 1195.574
quantile(eec$salred, na.rm = TRUE)
  ##    0%   25%   50%   75%  100%
  ##    24  1219  1600  2158 30042
quantile(eec$salred, na.rm = TRUE, probs = c(0.01, 0.05, 0.95, 0.99))
  ##      1%      5%     95%     99%
  ##  180.00  500.00 3683.00 6113.94
range(eec$salred, na.rm = TRUE)
  ## [1]    24 30042

# Coefficients de corrélation
cor.test(eec$salred, as.numeric(eec$age), method = "pearson")
  ##
  ##    Pearson's product-moment correlation
  ##
  ## data:  eec$salred and as.numeric(eec$age)
  ## t = 21.844, df = 15117, p-value < 2.2e-16
  ## alternative hypothesis: true correlation is not equal to 0
  ## 95 percent confidence interval:
  ##  0.1594280 0.1903331
  ## sample estimates:
  ##       cor
  ## 0.1749237
cor.test(eec$salred, as.numeric(eec$age), method = "spearman")
  ## Warning in cor.test.default(eec$salred, as.numeric(eec$age),
  ## method = "spearman"): Cannot compute exact p-value with ties
  ##
  ##    Spearman's rank correlation rho
  ##
  ## data:  eec$salred and as.numeric(eec$age)
  ## S = 464970000000, p-value < 2.2e-16
  ## alternative hypothesis: true rho is not equal to 0
  ## sample estimates:
  ##       rho
  ## 0.1927504
cor.test(eec$salred, as.numeric(eec$age), method = "kendall")
  ##
```

```
  ##    Kendall's rank correlation tau
  ##
  ## data:  eec$salred and as.numeric(eec$age)
  ## z = 24.949, p-value < 2.2e-16
  ## alternative hypothesis: true tau is not equal to 0
  ## sample estimates:
  ##       tau
  ## 0.1371121
```

Comme dans le cas des variables qualitatives, **par défaut R ne prend pas en charge le calcul de statistiques descriptives pondérées**. C'est ce que fait en revanche le ***package*** `Hmisc`, avec la série des fonctions `wtd.` : `wtd.mean()`, `wtd.var()`, `wtd.quantile()` notamment, qui comportent un argument `weights`.

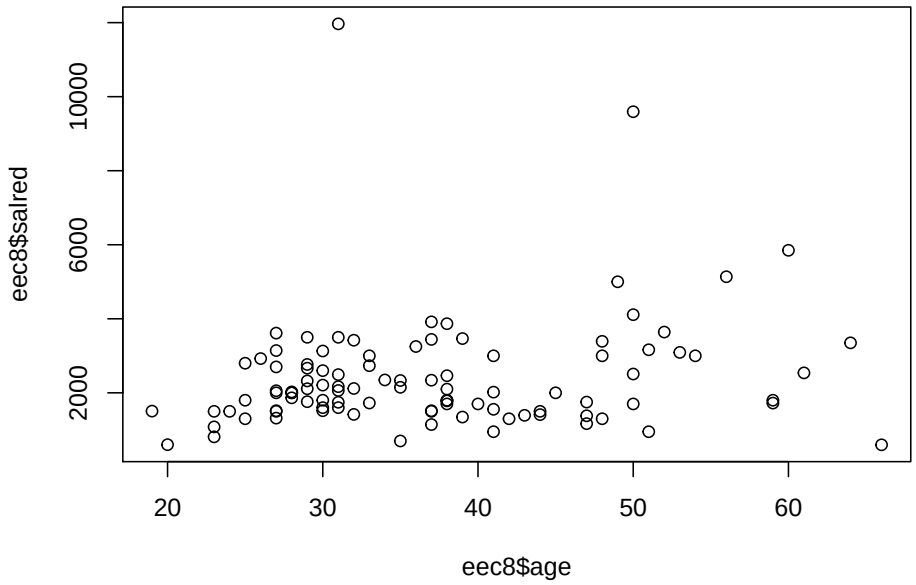
```
# Installation du package Hmisc
# install.packages("Hmisc")

# Chargement du package Hmisc
library(Hmisc)

# Statistiques pondérées avec Hmisc
wtd.mean(eec$salred, weights = eec$extri1613)
  ## [1] 1833.879
sqrt(wtd.var(eec$salred, weights = eec$extri1613))
  ## [1] 1225.035
wtd.quantile(eec$salred, weights = eec$extri1613, probs = seq(0, 1, 0.05))
  ##    0%    5%   10%   15%   20%   25%   30%   35%   40%   45%
  ##    24   507   758  1000  1150  1233  1302  1400  1459  1517
  ##   50%   55%   60%   65%   70%   75%   80%   85%   90%   95%
  ##  1600  1700  1800  1900  2000  2164  2332  2578  2984  3695
  ##  100%
  ## 30042
# Note : les fonctions wtd. du package Hmisc disposent
# également d'un paramètres na.rm, mais sa valeur est TRUE
# par défaut.
```

Graphiques

La production de graphiques est relativement simple dans R : dans la plupart des cas, **c'est la fonction `plot()` qu'il convient d'utiliser, qui adapte automatiquement le graphique aux caractéristiques de l'objet représenté**. De nombreuses **options graphiques** (taper `? plot` pour en afficher quelques unes) permettent de personnaliser assez finement l'affichage.

Pour représenter un **nuage de points**, il suffit par exemple d'appliquer `plot()` aux deux variables à représenter.

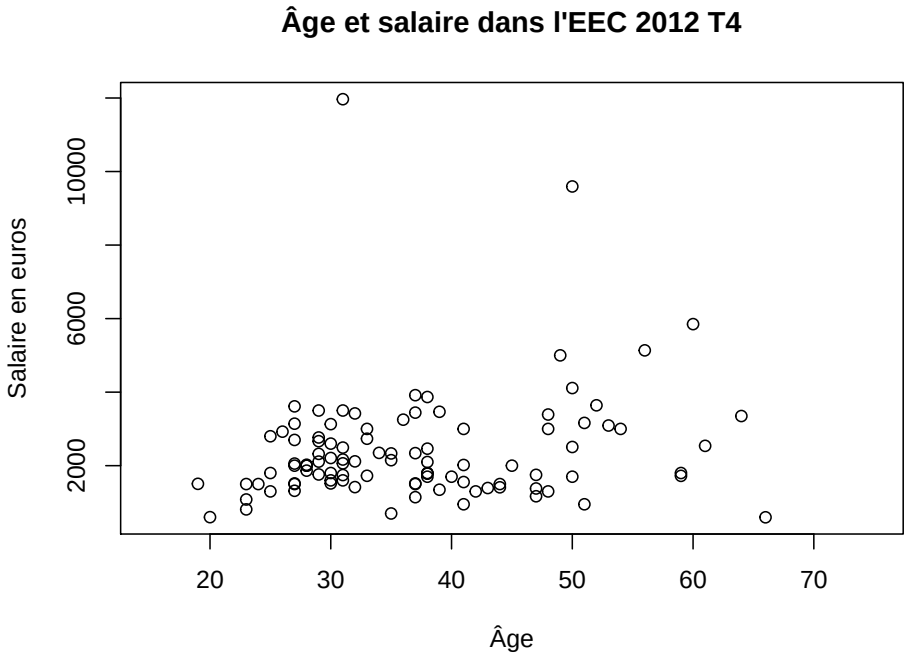
```
# On se restreint à une sous-base pour ne pas avoir un
# nuage de points trop dense
eec8 <- eec[which(!is.na(eec$salred))[1:100], ]
eec8$age <- as.numeric(eec8$age)

# Représentation du salaire en fonction de l'âge
plot(eec8$age, eec8$salred)
```



Plusieurs options de base contrôlent l'**affichage des titres et des axes** :

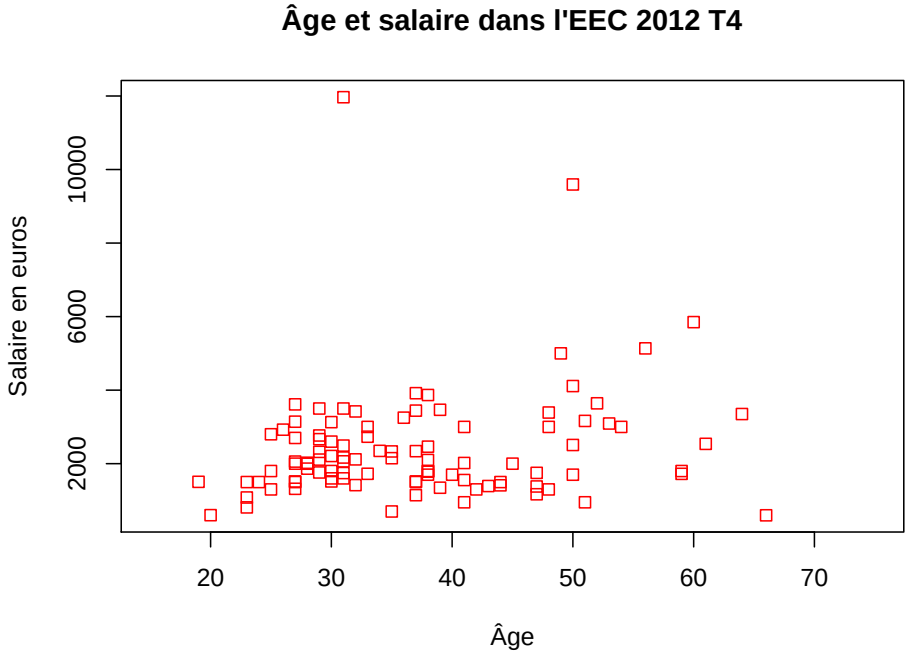
- `main` : titre principal du graphique ;
- `xlab`, `ylab` : titres des axes ;
- `xlim`, `ylim` : vecteurs de longueur 2 indiquant les limites des axes des abscisses et des ordonnées respectivement.

```
# Personnalisation du graphique précédent (1)
plot(
  eec8$age, eec8$salred
  , main = "Âge et salaire dans l'EEC 2012 T4"
  , xlab = "Âge", ylab = "Salaire en euros"
  , xlim= c(15, 75)
)
```



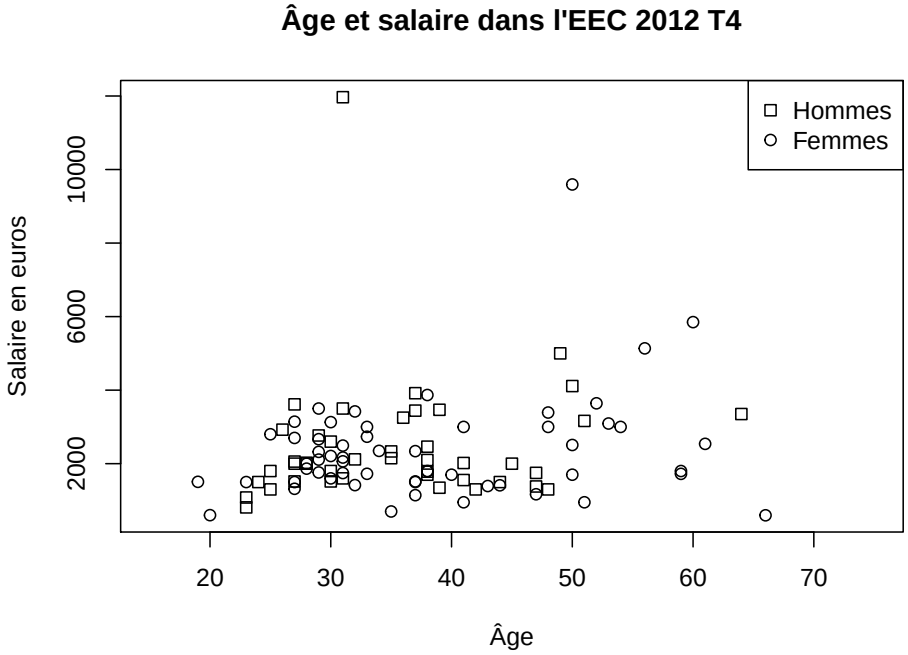
Les options `pch` et `col` permettent de **modifier la forme et la couleur des points représentés**.

```
# Personnalisation du graphique précédent (2)
plot(
  eec8$age, eec8$salred
  , main = "Âge et salaire dans l'EEC 2012 T4"
  , xlab = "Âge", ylab = "Salaire en euros"
  , xlim= c(15, 75)
  , pch = 0, col = 2
)
```



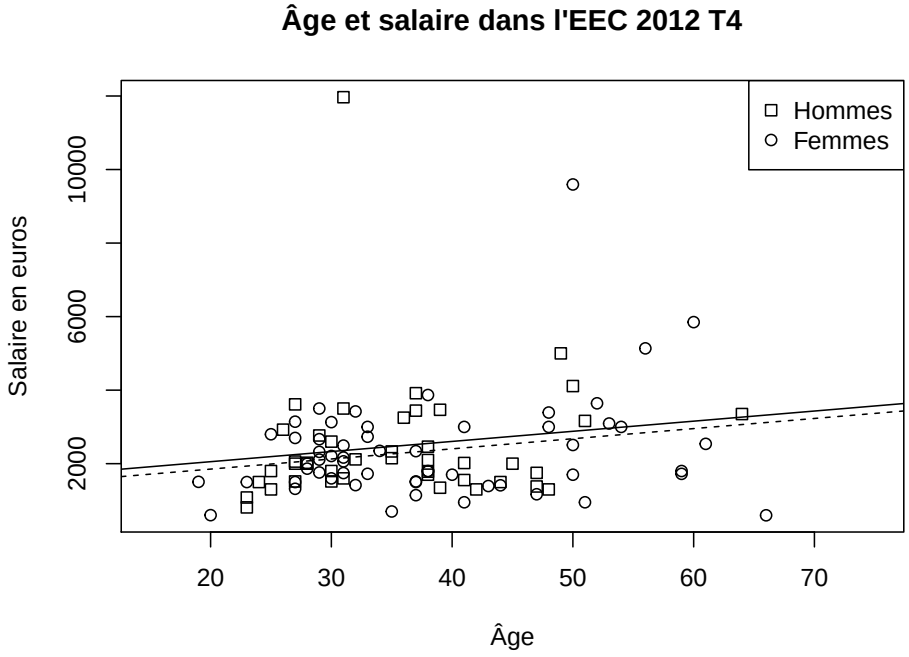
Utilisées avec des vecteurs et la fonction `legend()`, `pch` et `col` permettent de **représenter le croisement de plusieurs variables**.

```
# Utilisation de pch pour distinguer hommes et femmes
# sur le graphique
plot(
  eec8$age, eec8$salred
  , main = "Âge et salaire dans l'EEC 2012 T4"
  , xlab = "Âge", ylab = "Salaire en euros"
  , xlim= c(15, 75)
  , pch = as.numeric(eec8$sexe == "2")
)

# Ajout d'une légende
legend("topright", legend=c("Hommes","Femmes"), pch=c(0, 1))

# Sauvegarde du graphique pour la suite
g1 <- recordPlot()
```

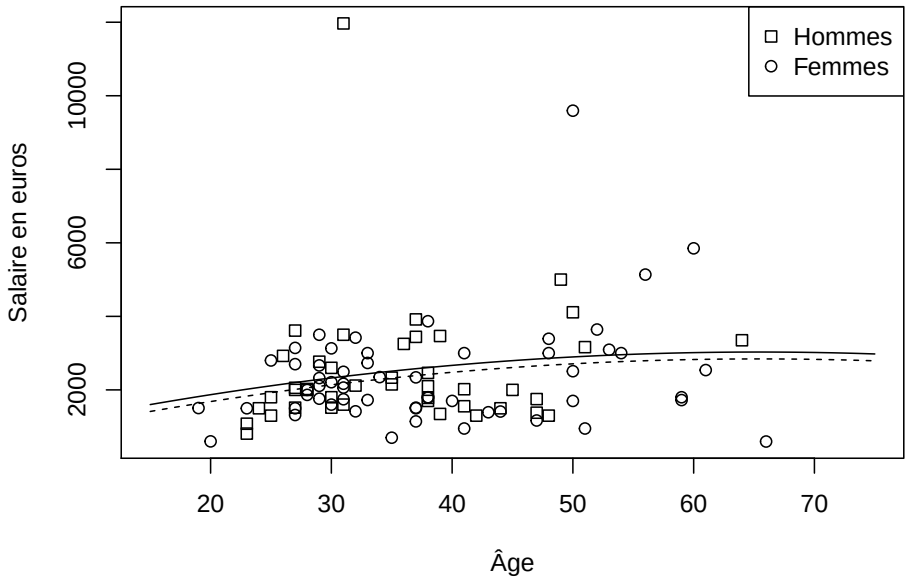


Les fonctions `abline()` et `curve()` **ajoutent respectivement des lignes et des courbes à un graphique existant**.

```
# Modèle de régression linéaire : salaire = age + sexe
# (cf. dernière partie)
eec8$femme <- eec8$sexe == "2"
m1 <- lm(salred ~ age + femme, data = eec8)

# Représentation des droites de régression correspondant
# aux hommes et aux femmes respectivement
g1
abline(a = coef(m1)[1], b = coef(m1)[2])
abline(a = coef(m1)[1] + coef(m1)[3], b = coef(m1)[2], lty = 2)
```

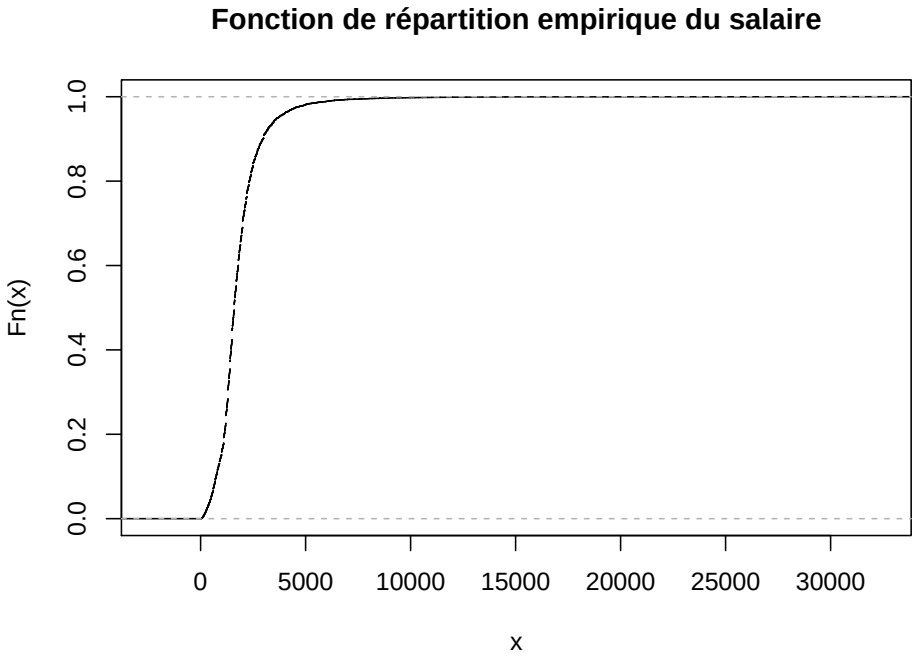



```
# Modèle de régression linéaire : salaire = age + age^2 + sexe
# (cf. sous-partie suivante)
eec8$age2 <- eec8$age^2
m2 <- lm(salred ~ age + age2 + femme, data = eec8)

# Représentation des courbes de régression correspondant
# aux hommes et aux femmes respectivement
g1
curve(coef(m2)[1] + coef(m2)[2]*x + coef(m2)[3]*x^2, add = TRUE)
curve(coef(m2)[1] + coef(m2)[4] + coef(m2)[2]*x + coef(m2)[3]*x^2, lty=2, add = TRUE
```

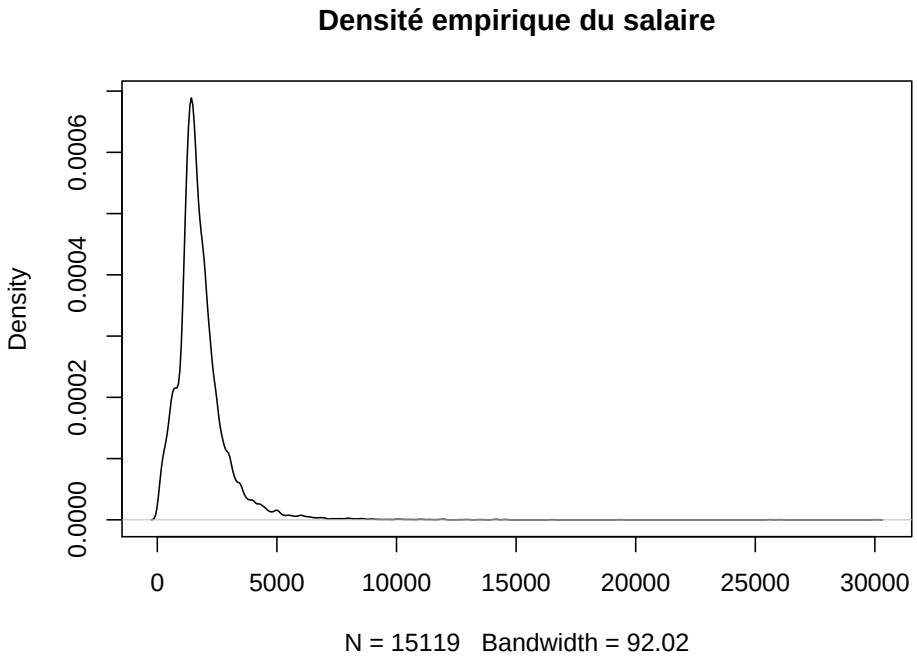


La fonction `plot()` permet également de représenter la **fonction de répartition et la densité empirique d'une distribution**, par le biais des fonctions `ecdf()` et `density()`.

```
# Fonction de répartition empirique du salaire
plot(
  ecdf(eec$salred)
  , main = "Fonction de répartition empirique du salaire"
)
```



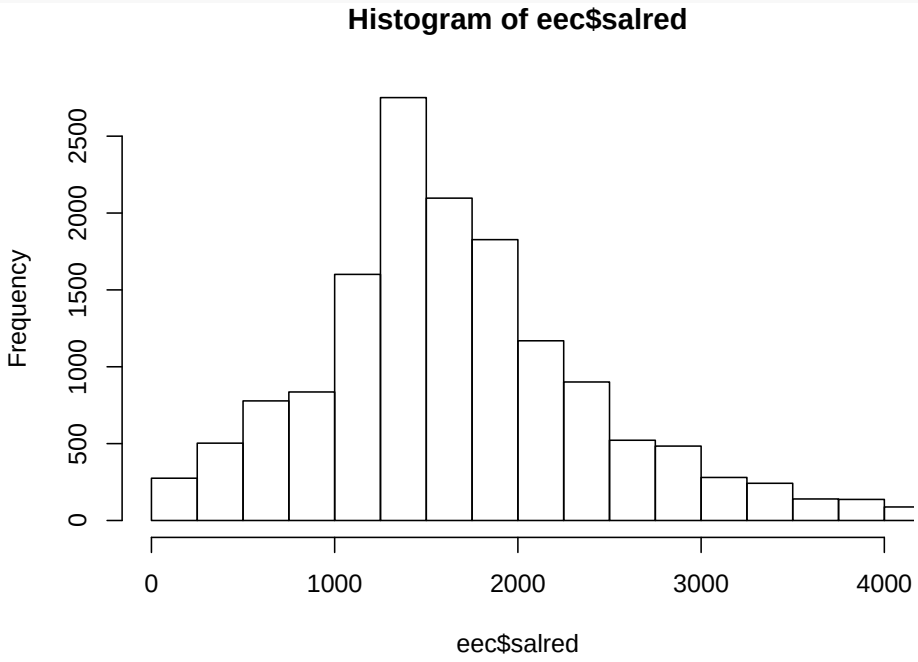

```
# Densité empirique du salaire
plot(
  density(eec$salred, na.rm = TRUE)
  , main = "Densité empirique du salaire"
)
```



Au-delà de la fonction `plot()`, de nombreuses fonctions permettent d'effectuer des représentations spécifiques dans R :

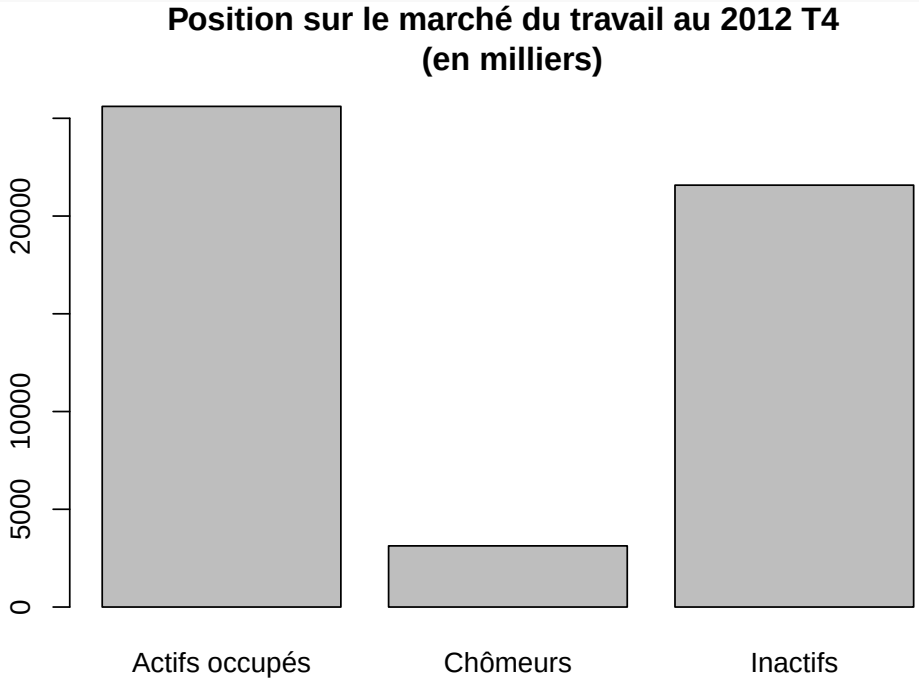
— `hist()` produit l'histogramme d'une distribution ;

```
# Histogramme du salaire dans l'EEC
hist(eec$salred, xlim = c(0, 4000), breaks = seq(0, 100000, 250))
```

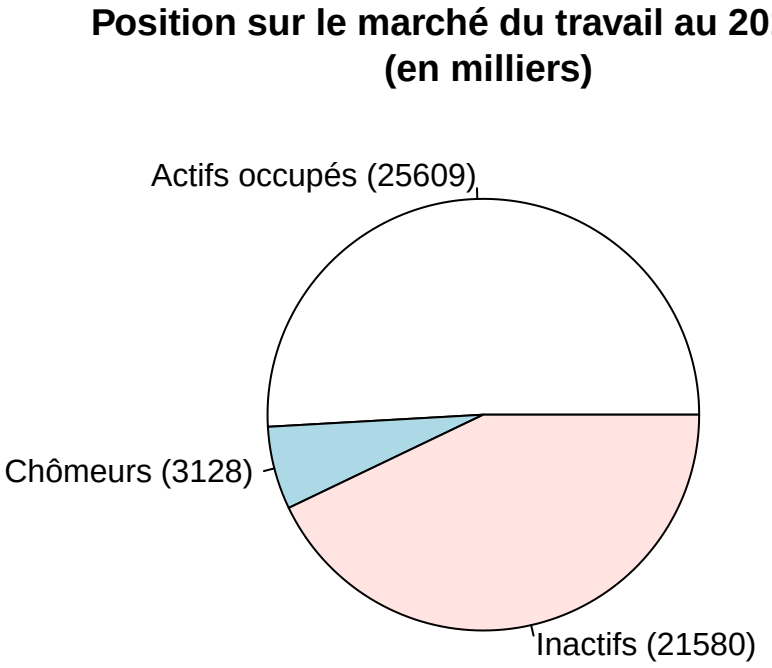


— `barplot()` et `pie()` produisent respectivement le diagramme en bâtons et le diagramme circulaire représentant la fréquence d'une variable qualitative.

```
# Distribution de la position du marché du travail
# en milliers
pos <- by(eec$extri1613, eec$acteu, sum) / 1000

# Diagramme en bâtons de la position sur le marché du travail
barplot(
  pos
  , names.arg = c("Actifs occupés", "Chômeurs", "Inactifs")
  , main = "Position sur le marché du travail au 2012 T4 \n (en milliers)"
)
```




```
# Diagramme circulaire de la position sur le marché du travail
pie(
  pos
  , labels = paste0(c("Actifs occupés", "Chômeurs", "Inactifs"), " (", round(pos),'
  , main = "Position sur le marché du travail au 2012 T4 \n (en milliers)"
)
```



Application à l'enquête Pisa 2012

L'enquête Pisa (*Program for International Student Assessment*) est une enquête réalisée **tous les trois ans** par l'Organisation de coopération et de développement économique (OCDE) dans une soixantaine de pays auprès des **élèves de 15 ans** (quelle que soit leur classe au moment de l'enquête).

Elle vise à mesurer les **acquis des élèves de 15 ans dans trois disciplines** : mathématiques, compréhension de l'écrit (ou *littératie*) et sciences. En plus des scores aux **tests standardisés** de mathématiques, compréhension de l'écrit et sciences, cette enquête comporte de très nombreuses informations sur l'origine sociale des élèves, leurs conditions d'enseignement ainsi que leur rapport aux enseignants et à l'école.

Organisation des fichiers Les fichiers de l'enquête Pisa 2012 et leur documentation sont librement téléchargeables sur le site de l'OCDE. Seuls **deux des nombreux fichiers de données** qui constituent l'enquête seront utilisés :

— le **fichier élève** `pisa_stu.sas7bdat` ;
— le **fichier établissement** `pisa_sch.sas7bdat`.

Ces deux fichiers ont été **restreints à la France** et à un **ensemble réduit de variables** :

Fichier élève (`pisa_stu.sas7bdat`)

Variable	Description
cnt	Pays
stidstd	Identifiant de l'élève

Variable	Description
schoolid	Identifiant de l'établissement
w_fstuwt	Poids de sondage final de l'élève
st01q01	Classe en nombre d'années depuis l'entrée en primaire : la $10^{ème}$ classe correspond à la seconde en France.
st04q01	Sexe : (1) Femme (2) Homme
st05q01	A suivi une scolarité pré-primaire (1) Non (2) Oui, un an ou moins (3) Oui, plus d'un an
st07q01 st07q02 st07q03	A redoublé à un moment de sa scolarité : (1) Non (2-3) Oui, une ou plusieurs fois
st08q01	Est arrivé en retard au cours des deux semaines précédant l'enquête
st09q01	A séché les cours au cours des deux semaines précédant l'enquête
anxmat	Score synthétique d'anxiété en mathématiques
disclima	Score synthétique de climat de discipline dans la classe
escs	Indicateur synthétique de statut économique, social et culturel
immig	Immigration : (1) Né en France (2) Immigré de deuxième génération (3) Immigré de première génération
hisced	Niveau d'étude le plus élevé des parents (nomenclature CITE)
pv1math	Score synthétique à l'évaluation de mathématiques
pv1read	Score synthétique à l'évaluation de compréhension de l'écrit
pv1scie	Score synthétique à l'évaluation de sciences

Fichier établissement (`pisa_sch.sas7bdat`)

Variable	Description
cnt	Pays
schoolid	Identifiant de l'établissement
senwgt_scq	Poids de sondage (la somme vaut 1 000 dans chaque pays)
sc01q01	Statut public ou privé (1) public (2) privé
sc03q01	Taille de la commune de l'établissement : (1) *Village* (2) *Small town* (3) *Town* (4) *City* (5) *Large city*

Variable	Description
sc05q01	Taille de la classe en cours de français : (01) 15 ou moins (02) 16-20 (03) 21-25 ... (08) 46-50 (09) Plus de 50 élèves

Cas pratique 3.5 Application à l'enquête Pisa 2012 : Importation et mise en forme des données

a. Utilisez la fonction `read_sas()` du *package* `haven` (*cf.* module 1) pour importer les deux fichiers dans les objets `stu` et `sch` respectivement. Afin de faciliter les exploitations futures, passez leurs noms de variables en minuscules.

```
# Import des fichiers avec la fonction read_sas()
# install.packages("haven")
library(haven)
stu <- read_sas("pisa_stu.sas7bdat")
sch <- read_sas("pisa_sch.sas7bdat")

# Passage des noms en minuscules
names(stu) <- tolower(names(stu))
names(sch) <- tolower(names(sch))
```

---

b. On souhaite pouvoir utiliser les informations au niveau de l'établissement dans des exploitations au niveau des élèves. Pour ce faire, il convient de fusionner les tables `stu` et `sch` sur la base de la variable `schoolid`.

   i. Utilisez les fonctions `unique()`, `intersect()` et `setdiff()` pour vérifier que l'identifiant `schoolid` prend bien les mêmes valeurs dans les deux tables.

```
length(intersect(stu$schoolid, sch$schoolid))
  ## [1] 226
length(unique(stu$schoolid))
  ## [1] 226
length(unique(sch$schoolid))
  ## [1] 226
```

```
# Cela fonctionne aussi plus finement avec setdiff()
setdiff(stu$schoolid, sch$schoolid)
  ## character(0)
# Aucune valeur de stu$schoolid n'est pas dans sch$schoolid
setdiff(sch$schoolid, stu$schoolid)
  ## character(0)
# Aucune valeur de sch$schoolid n'est pas dans stu$schoolid
```

---

ii. Vérifiez que la variable `schoolid` est un identifiant pour la table `sch`, à savoir : (1) qu'elle est renseignée pour chaque ligne (2) qu'elle prend une valeur distincte pour chaque ligne.

```
# Pour vérifier que schoolid identifie sch il suffit
# de comparer le nombre de valeurs distinctes au
# nombre de lignes
length(unique(sch$schoolid))
  ## [1] 226
nrow(sch)
  ## [1] 226
# Tout va bien !
```

---

iii. Utilisez la fonction `merge()` pour fusionner `stu` et `sch` par `schoolid` et créez la table `stu2`. Vérifiez que ses propriétés sont cohérentes avec le résultat des questions précédentes : même nombre de lignes que `stu`, nombre de colonnes égal à celui de `stu` et de `sch` moins 1.

```
# Fusion des deux tables
stu2 <- merge(stu, sch, by = "schoolid")

# Vérifications
nrow(stu2) == nrow(stu)
  ## [1] TRUE
ncol(stu2) == ncol(stu) + ncol(sch) - 1
  ## [1] TRUE
```

---

c. Recodage de variables

i. Recodez la variable de sexe en facteur dont les libellés sont `"Femme"` et `"Homme"` pour faciliter la lecture des tableaux et graphiques.

```
# Création de la variable sexe recodée
stu2$sexe <- factor(stu2$st04q01, labels = c("Femme", "Homme"))
```

---

ii. Un élève a redoublé à un moment dans sa scolarité dès lors qu'une des variables `st07q01`, `st07q02` ou `st07q03` vaut 2 ou 3. Créez la variable indicatrice `redoublant` valant `TRUE` si un élève a redoublé au cours de sa scolarité.

```
# Création de la variable redoublant
stu2$redoublant <- (
  stu2$st07q01 %in% c(2, 3) |
    stu2$st07q02 %in% c(2, 3) |
    stu2$st07q03 %in% c(2, 3)
)
table(stu2$redoublant)
  ##
  ## FALSE  TRUE
  ##  3356  1257
```

---

iii. Quelle est la nature de l'indicateur synthétique de statut économique, social et culturel ? Recodez-le sous la forme d'une variable qualitative à 5 modalités (en utilisant les fonctions `cut()` et `quantile()`).

```
# La variable escs est de nature quantitative, ce qui n'est pas très nature
summary(stu2$escs)
  ##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.     NA's
  ## -3.46000 -0.60000  0.04000 -0.02122  0.60000  2.20000      124

# On la recode en une variable qualitative selon ses quintiles
# pour facilier les interprétations
stu2$escsq <-  cut(
  stu2$escs
  , quantile(stu2$escs, probs = c(0, 0.20, 0.40, 0.60, 0.80, 1), na.rm = TR
```

```
)
table(stu2$escsq)
  ##
  ## (-3.46,-0.74] (-0.74,-0.21]  (-0.21,0.25]   (0.25,0.72]
  ##           902           896           896           905
  ##    (0.72,2.2]
  ##           889
```

Cas pratique 3.6 Application à l'enquête Pisa 2012 : Statistiques descriptives

a. En utilisant le *package* `descr`, effectuez le tri croisé entre sexe des élèves et redoublement et interprétez-le.

```
# Installation du package descr
# install.packages("descr")

# Chargement du package descr
library(descr)

# Tri croisé pondéré
crosstab(
  stu2$sexe, stu2$redoublant, weight = stu2$poids
  , prop.r = TRUE, chisq = TRUE, prop.c = TRUE
)
  ##    Cell Contents
  ## |-------------------------|
  ## |                   Count |
  ## |             Row Percent |
  ## |          Column Percent |
  ## |-------------------------|
  ##
  ## ==================================
  ##              stu2$redoublant
  ## stu2$sexe    FALSE    TRUE   Total
  ## ----------------------------------
  ## Femme         1794     581    2375
  ##               75.5%   24.5%   51.5%
  ##               53.5%   46.2%
```

```
## ---------------------------------
## Homme        1562     676    2238
##              69.8%   30.2%   48.5%
##              46.5%   53.8%
## ---------------------------------
## Total        3356    1257    4613
##              72.8%   27.2%
## =================================
##
## Statistics for All Table Factors
##
## Pearson's Chi-squared test
## ------------------------------------------------------------
## Chi^2 = 19.16612      d.f. = 1      p = 0.000012
##
## Pearson's Chi-squared test with Yates' continuity correction
## ------------------------------------------------------------
## Chi^2 = 18.87755      d.f. = 1      p = 0.0000139
##         Minimum expected frequency: 609.8344
```

Dans l'ensemble de la population, 27,9 % des élèves de 15 ans ont redoublé à un moment ou à un autre de leur scolarité. Ils sont 30,8 % parmi les hommes et 25,1 % parmi les femmes : autrement dit, les élèves ayant deroublé à un moment ou à un autre de leur scolarité sont surreprésentés parmi les hommes.

Le test d'indépendance du χ^2 permet de confirmer cette analyse : sa p-valeur est inférieure à 0,01 aussi il est possible de rejeter l'hypothèse nulle d'indépendance entre les variables de sexe et de reoublement au seuil de 1 %.

---

b. Calculez le coefficient de corrélation linéaire de Pearson entre notes en mathématiques et en sciences et menez le test de nullité de ce coefficient (avec la fonction `cor.test()`).

```
cor.test(stu2$pv1math, stu2$pv1scie)
```

```
##
##    Pearson's product-moment correlation
##
## data:  stu2$pv1math and stu2$pv1scie
## t = 141.85, df = 4611, p-value < 2.2e-16
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.8964568 0.9072244
## sample estimates:
```

```
##       cor
## 0.9019808
```

Le coefficient de corrélation entre note en mathématiques et note en sciences est positif et extrêmement élevé (0,9020). La p-valeur du test de nullité de ce coefficient est inférieure à 0,01 aussi il est possible de rejeter au seuil de 1 % l'hypothèse que ces deux scores ne soient pas corrélés.

---

c. Calculez le score moyen en mathématique selon les quintiles de statut économique, social et culturel (*cf.* le recodage du cas pratique précédent).

```
tapply(stu2$pv1math, stu2$escsq, mean)
```

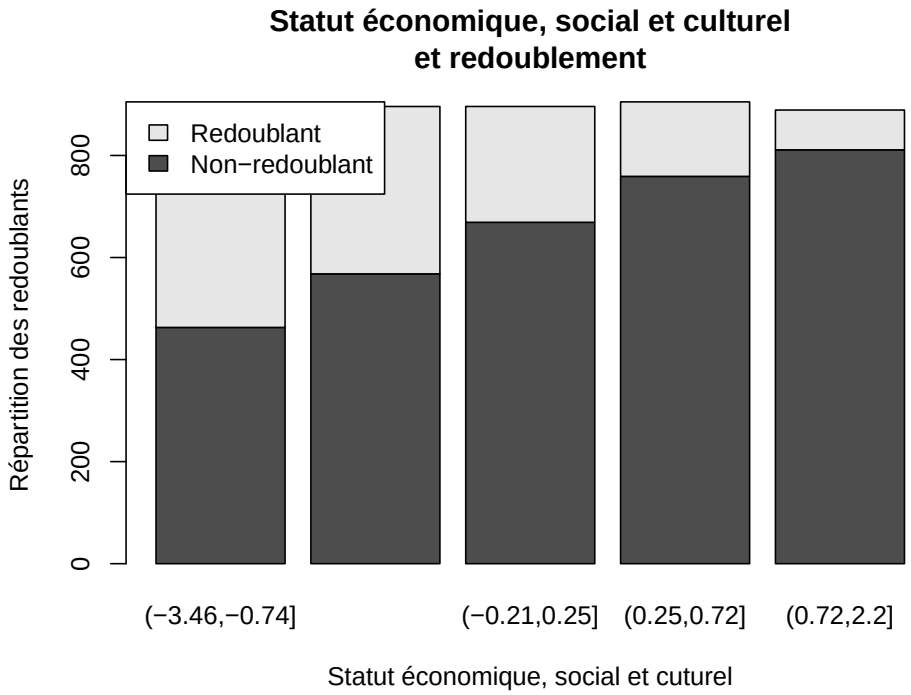
```
## (-3.46,-0.74] (-0.74,-0.21]  (-0.21,0.25]   (0.25,0.72]
##      441.6501      468.6728      499.9407      532.5550
##    (0.72,2.2]
##      567.2039
```

La moyenne du score en mathématiques est d'autant plus élevée que le statut économique, social et culturel est favorisé.

Cas pratique 3.7 Application à l'enquête Pisa 2012 : Graphiques

a. Construisez un diagramme en bâton pour illustrer la relation entre statut économique, social et culturel (en quintiles) et redoublement. Utilisez les options de mise en forme pour améliorer sa présentation (ajouter un titre avec `main()`, modifiez les titres des axes avec `xlab()` et `ylab()`, etc.).

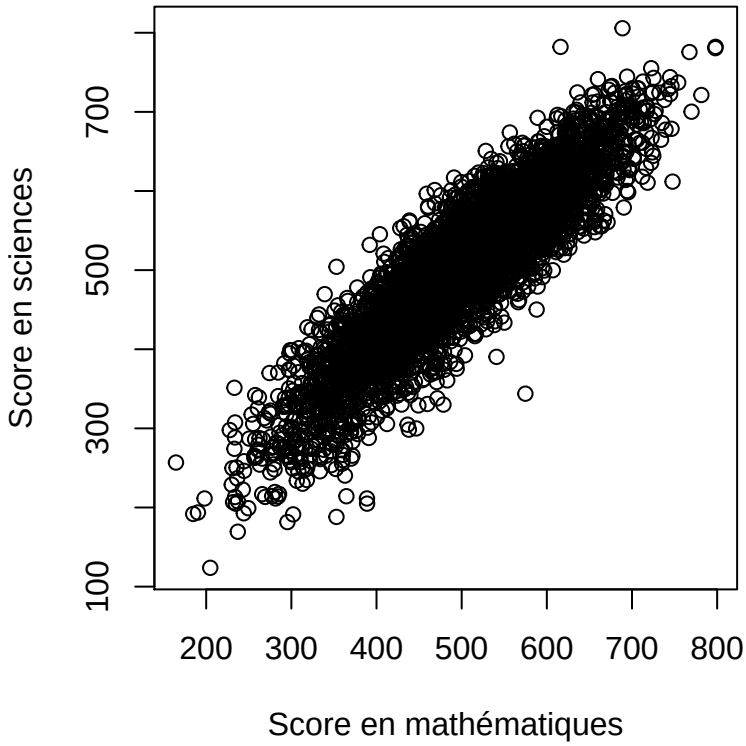
```
barplot(
  table(stu2$redoublant, stu2$escsq)
  , main = "Statut économique, social et culturel\net redoublement"
  , xlab = "Statut économique, social et cuturel"
  , ylab = "Répartition des redoublants"
  , legend.text = c("Non-redoublant", "Redoublant")
  , args.legend = list(x = "topleft", bg = "white")
)
```



---

b. Utilisez la fonction `plot()` pour représenter le nuage de points de la relation entre le score en mathématiques et le score en sciences.

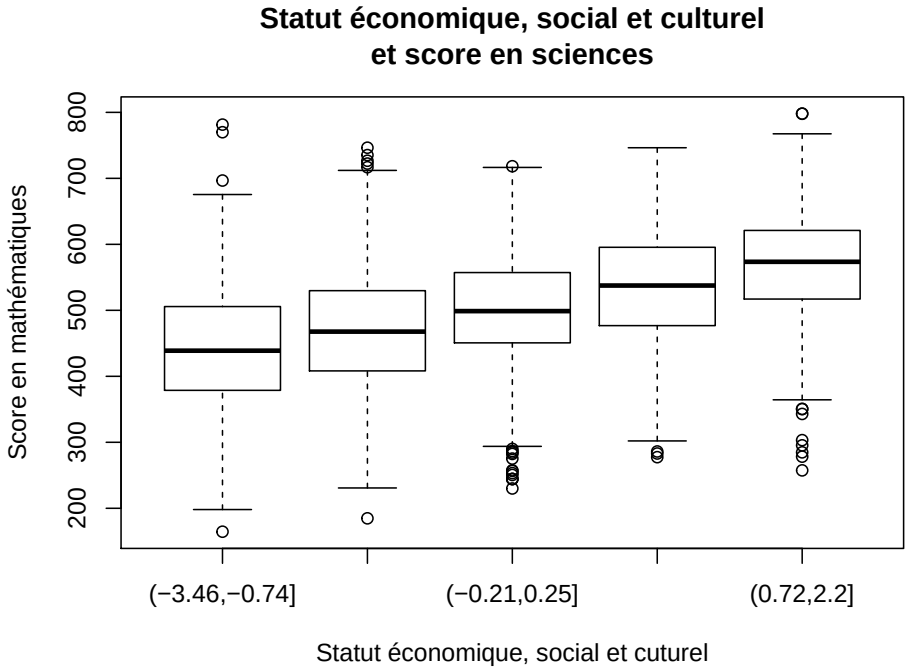
```
# Score en mathématiques et en sciences
par(pty="s")
plot(
  stu2$pv1math, stu2$pv1scie
  , xlab = "Score en mathématiques"
  , ylab = "Score en sciences"
)
```



---

c. Construisez la « boîte à moustaches » représentant la relation entre stat au moinst économique, social et culturel (en quintiles) et score en mathématiques à l'aide de la fonction `boxplot()`.

```
boxplot(
  stu2$pv1math ~ stu2$escsq
  , main = "Statut économique, social et culturel\net score en sciences"
  , xlab = "Statut économique, social et cuturel"
  , ylab = "Score en mathématiques"
)
```



d. Utilisez les fonctionnalités de RStudio (menus déroulants de la fenêtre de graphiques) pour sauvegarder ces graphiques dans la qualité et le format souhaités.

Quelques liens pour aller plus loin

Formation R perfectionnement

Le support de la formation R perfectionnement que j'ai conçue est en ligne à l'adresse : teaching.slmc.fr/perf. Elle aborde trois sujets :

1. Outils et méthodes pour se perfectionner avec R ;
2. Traitements avancés sur des données dans R : retour sur les fonctions `*apply()` et assimilées, optimisation en base R, *packages* `dplyr` et `data.table`, parallélisation et utilisation de langages de bas niveau dans R.
3. Graphiques et *reporting* avec R : *package* `ggplot2`, production automatique de documents avec Rmarkdown.

Un cycle de formation perfectionnement est également proposé par la division Formation : certaines portent sensiblement sur les mêmes sujets, mais pas toutes (notamment une consacrée à R Shiny).

Utiliser des techniques d'analyse de données multidimensionnelles

Le package `FactoMineR` (attention à la casse !) rend extrêmement simple la mise en oeuvre sous R de **techniques d'analyse de données multidimensionnelles** : analyse en composante principale (ACP), analyse des correspondances multiples (ACM) ou encore classification ascendante hiérarchique (CAH).

Ce document d'introduction (« vignette » dans la terminologie de R) présente ces méthodes et leur mise en oeuvre avec `FactoMineR`.

Estimer des modèles de régression

Les fonctions natives de R `lm()` et `glm()` permettent respectivement d'estimer des **modèles linéaires et linéaires généralisés** (dont les modèles logistiques).

Cette page (destinée à un public de non-statisticiens) introduit les méthodes de régression et propose de nombreux exemples d'estimation de modèles dans R.

Liste des cas pratiques

Index des fonctions et opérateurs