Chapitre 2 - Les objets (partie 1)

Thibault LAURENT

12 octobre 2021

Contents

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

Avant de commencer

Les informations que l'utilisateur souhaite traiter sont contenues dans des **objets**. Ces derniers, identifiés par un nom, peuvent avoir des caractéristiques très différentes et être manipulés de manière à fournir l'information souhaitée par l'utilisateur. L'objectif de ce chapitre est de vous familiariser avec les différents types d'objets,

mais aussi avec leur gestion et leur manipulation. Avant de commencer, vous devez effectuer les opérations suivantes afin de disposer de tous les éléments nécessaires à l'apprentissage de ce chapitre.

1. Créer un dossier propre à ce chapitre. Ce dossier servira à stocker les données et/ou les résultats manipulés durant une session de travail.
2. Lancer **RStudio**.
3. Indiquer le répertoire dans lequel vous allez travailler. Cette démarche s'effectue avec la commande *setwd*. Par exemple :

```
setwd("C:/Documents/m2_foad/cours_r/chap_2")
```

4. Charger le code **R** qui permet de créer le jeu de données présenté dans le chapitre 1 :

```
load(file("http://www.thibault.laurent.free.fr/cours/Ressource/diamants.RData"))
```

Remarque : nous vous recommandons d'ouvrir un Script **R** (par exemple, depuis **RStudio**, faire File < New **R** Script) et de recopier dans ce script toutes les commandes qui sont fournies dans ce document. Vous pourrez écrire vos commentaires dans ce document en utilisant le symbole #.

1 Les vecteurs

Stocker les données sous forme de vecteurs est une des premières choses à savoir faire avec **R**. Il y a plusieurs caractéristiques qui définissent un vecteur :

- le type des éléments qui constituent ce vecteur,
- sa taille, c'est-à-dire le nombre d'éléments qu'il contient,
- ses attributs, c'est-à-dire des méta-données associées aux vecteurs.

Nous allons commencer par voir quels sont les principaux types pouvant caractériser un vecteur.

1.1 Les différents types possibles

Les vecteurs sont généralement construits à partir de la fonction collecteur *c()*. Les principaux types d'objet sont présentés ci-dessous.

1.1.1 Le type numeric

Le type **numeric** contient des valeurs réelles. La fonction *is.numeric()* permet de tester qu'un objet contient bien des éléments du type **numeric**.

```
a.numeric <- c(1.2, 3.5, 5.4, 6.2, 8.1)
is.numeric(a.numeric)
```

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

Remarque : pour ceux et celles qui sont familiers avec le type **double**, on peut dire que les modes **numeric** et **double** sont identiques.

Dans l'exemple précédent, on a créé un vecteur directement à partir de la fonction *c()*. Une autre façon de faire est de créer dans un premier temps un vecteur de taille 5 avec la fonction *numeric()* (l'argument à mettre dans la fonction est la taille du vecteur, ici 5), puis de remplir ce vecteur élément après élément. Sous **R**, le 1er élément est l'indice 1 et l'accès aux éléments du vecteurs se fait avec les crochets. Ainsi, la commande précédente est équivalente à :

```
a.numeric <- numeric(5) # on crée un vecteur de numeric de taille 5
a.numeric[1] <- 1.2 # on affecte la valeur 1.2 au 1er élément du vecteur
a.numeric[2] <- 3.5 # etc.
```

```
a.numeric[3] <- 5.4
a.numeric[4] <- 6.2
a.numeric[5] <- 8.1
a.numeric
```

```
## [1] 1.2 3.5 5.4 6.2 8.1
```

1.1.2 Le type integer

Pour construire un vecteur qui contient des éléments de type **integer**, on ajoute le suffixe "L" au nombre entier. L'intérêt d'utiliser le type **integer** plutôt que **numeric** est qu'un entier prend moins d'espace mémoire qu'un **double**. On vérifie qu'un objet contient bien des éléments du type **integer** avec la fonction *is.integer()* :

```
a.integer <- c(1L, 3L, 5L, 6L, 8L)
is.integer(a.integer)
```

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

```
is.numeric(a.integer)
```

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

Remarque : un **integer** est à la fois de type **integer** et **numeric**.

Comme précédemment, on peut utiliser la fonction *integer()* pour créer un vecteur d'entiers et ensuite le remplir petit à petit.

```
a.integer <- integer(5)
a.integer[1] <- 1L
a.integer[2] <- 3L
a.integer[3] <- 5L
a.integer[4] <- 6L
a.integer[5] <- 8L
```

1.1.3 Le type character

Le type **character** contient des chaînes de caractères.

```
a.character <- c("AUREVILLE", "AUSSONNE", "AUZEVILLE-TOLOSANE", "AUZIELLE")
is.character(a.character)
```

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

On teste qu'un objet contient bien des éléments du type **character** avec la fonction *is.character()*.

Comme précédemment, on peut utiliser la fonction *character()* pour créer un vecteur de chaînes de caractères et ensuite le remplir petit à petit.

```
a.character <- character(4)
a.character[1] <- "AUREVILLE"
a.character[2] <- "AUSSONNE"
a.character[3] <- "AUZEVILLE-TOLOSANE"
a.character[4] <- "AUZIELLE"
```

1.1.4 Le type logical

Le type **logical** ne contient que les valeurs **TRUE** (**T** est aussi acceptée) et **FALSE** (**F** est aussi acceptée). On n'utilise pas les guillements comme on le fait pour définir les chaînes de caractères.

```
a.logical <- c(TRUE, FALSE, FALSE, TRUE)
is.logical(a.logical)
```

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

Comme précédemment, on peut utiliser la fonction *logical()* pour créer un vecteur de booléens et le remplir ensuite élément après élément. Comme les valeurs créées par défaut sont FALSE, on n'est donc pas obligé de changer le 2ème et 3ème élément.

```
a.logical <- logical(4)
a.logical
```

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

```
a.logical[1] <- TRUE
a.logical[4] <- TRUE
```

Exercice 1.1

- créer l'object **my__num**, un vecteur de **numeric** qui contient les valeurs suivantes: 28, 29, 35, 75, 40, 52, 23, 25, 10, 50.
- créer l'object **my__char**, un vecteur de **character** qui contient les valeurs suivantes: "M", "M", "M", "M", "F", "M", "F", "F", "F", "F".

1.2 Passage d'un type à un autre

Parfois on peut avoir besoin de passer d'un type à un autre. Prenons par exemple le vecteur **a.logical** et intéressons-nous au nombre de fois où la valeur **TRUE** est apparue. Pour cela, on peut utiliser la fonction *sum()* qui s'applique au type **numeric**. Au moment d'appliquer la fonction *sum()*, si **R** reconnaît le type **logical**, il va automatiquement le convertir en **numeric** (**TRUE** prendra la valeur 1 et **FALSE** la valeur 0). Par exemple :

```
sum(a.logical)
```

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

Il en est de même pour toutes les opérations de calcul élémentaire (+, -, /, etc) qui s'appliquent sur les types **numeric**. Si on les applique à un type **logical**, la conversion en **numeric** sera automatique :

```
a.logical + 5
```

```
## [1] 6 5 5 6
```

La conversion d'un type vers un autre se fait rarement automatiquement. Pour le faire, il faudra utiliser des fonctions prévues à cet effet.

1.2.1 Passage du numeric au character (et inversement)

On peut utiliser la fonction *as.character()* pour transformer un vecteur **numeric** (ou **logical**) en **character**.

```
as.character(a.numeric)
```

```
## [1] "1.2" "3.5" "5.4" "6.2" "8.1"
```

```
as.character(a.logical)
```

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

Pour transformer un **character** en **numeric**, cela est possible si et seulement si il la chaîne de caractères ne contient que des chiffres. Par exemple, la commande suivante va produire un message d'avertissement. Un message d'avertissement ("Warning" en anglais) est différent d'un message d'erreur ("Error" en anglais). En

effet, un message d'erreur ne retourne aucun résultat, alors qu'ici on constate qu'un résulat est retourné. Il s'agit d'un vecteur de **NA**. **NA** signifie "Non Available" et ceci correspond à des valeurs manquantes d'un point de vue statistique.

```
as.numeric(a.character)
```

```
## Warning: NAs introduits lors de la conversion automatique
```

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

Un exemple de commande qui fonctionne :

```
as.numeric(c("25.5", "12.2", "5"))
```

```
## [1] 25.5 12.2  5.0
```

1.2.2 Passage en logical (et inversement)

Pour transformer un **character** ou un **numeric** en **logical**, il faut que les chaînes de caractères soient égales à **"TRUE"**, **"T"**, **"FALSE"** ou **"F"** et que les valeurs numériques soients égales à 0 ou 1. Par exemple, cette instruction va produire des valeurs manquantes car elle ne répond pas au critère :

```
as.logical(a.character)
```

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

En revanche, ces commandes fonctionnent :

```
as.logical(c("FALSE", "T", "FALSE"))
```

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

```
as.logical(c(0, 1, 0))
```

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

1.2.3 Règles de priorité

Lorsqu'on définit un nouveau vecteur qui contient des objets de différents types, les règles suivantes s'appliquent :

- s'il y a des booléens avec des valeurs numériques, les booléens sont convertis en numériques (1 pour **TRUE** et 0 pour **FALSE**)

```
c(TRUE, 1, FALSE, pi)
```

```
## [1] 1.000000 1.000000 0.000000 3.141593
```

- s'il y a au moins une chaîne de caractère, tout est converti en **character**.

```
c(TRUE, 1, FALSE, pi, "a_string")
```

```
## [1] "TRUE"             "1"                "FALSE"            "3.14159265358979"
## [5] "a_string"
```

Remarque: certains langages de programmation ne permettent pas de faire des conversion automatiques d'un type vers l'autre. **R** est un langage interprété, ce qui signifie que l'interpréteur va exécuter les lignes du code une par une, en décidant à chaque étape ce qu'il va faire ensuite.

Exercice 1.2

- Que va-t-il se passer pour la ligne suivante:

```
c(21, 180, "F", "DU", "FR", TRUE)
```

1.3 Les attributs

Les attributs peuvent être vus comme des meta-données qui apporte de l'information sur le vecteur. On peut en définir autant qu'on souhaite avec la fonction *attr()*. Par exemple, si je souhaite créer un attribut "titre" dont l'objectif serait d'informer l'utilisateur sur le titre de l'objet créé. On constate que cet attribut est imprimé lorsqu'on affiche le vecteur :

```
attr(a.integer, "titre") <- "Je suis un vecteur"
a.integer
```

```
## [1] 1 3 5 6 8
## attr(,"titre")
## [1] "Je suis un vecteur"
```

Cependant, il est peu courant de créer de nouveaux attributs comme on vient de le faire. Une des raisons est qu'en manipulant le vecteur, l'attribut risque de se perdre. Par exemple, si j'extrais le 5ème élément du vecteur (ceci se fait avec les []), on constate que l'attribut "titre" a disparu :

```
a.integer[5]
```

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

En revanche, il existe certains attributs qu'il est important de connaître. Soit le vecteur suivant :

```
x <- c(a = 1, b = 2, c = 3)
x
```

```
## a b c
## 1 2 3
```

Les attributs associés au vecteur **x** qu'il est important de connaître sont :

- le nom des observations (si on décide que chaque élément doit être identifié par une étiquette) :

```
attr(x, "names")
```

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

Plutôt que d'utiliser cette syntaxe "lourde" pour connaître le nom des observations, on utilisera directement la fonction *names()* :

```
names(x)
```

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

- la classe du vecteur (pour identifier quel est le type des éléments du vecteur). On utilisera la fonction *class()* :

```
class(x)
```

```
## [1] "numeric"
```

- la dimension. Pour un vecteur, on utilise la fonction *length()* :

```
length(x)
```

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

Exercice 1.3

- Donner aux objets **my_num** et **my_char** précédémment créés un nom pour chaque observation. On prendra par exemple **com_1**, **com_2**, ..., **com_10**.

1.4 Comparaison sur des vecteurs

Les opérateurs de comparaison classiques sont > (supérieur), >= (supérieur ou égal), == (égal), **!=** (différent), <= (inférieur ou égal), <. Quand on applique ces opérateurs à des vecteurs, le résultat retourné est un vecteur de type **logical** qui indique donc **TRUE** si la condition a été respectée et **FALSE** sinon.

On va utiliser les vecteurs suivants pour illustrer ce paragraphe :

```
x <- c(1, 2, 3, 4, 5, 6, 7, 9, 11)
y <- c("oui", "non", "oui", "oui", "non", "non", "oui", "oui", "peut-être", "non")
```

La commande suivante retourne quels sont les éléments de **x** qui sont supérieurs ou égaux à 5.

```
x >= 5
```

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

En réalité, la commande ci-dessus est équivalente à :

```
x >= c(5, 5, 5, 5, 5, 5, 5, 5, 5)
```

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

Autrement dit, **R** s'attend à avoir des vecteurs de même taille à gauche et à droite de l'opérateur de comparaison. Si ce n'est pas le cas, il concatène au vecteur le plus petit autant de valeurs nécessaires de celui-ci pour obtenir la bonne taille. Ainsi :

```
x >= c(1, 10)
```

```
## Warning in x >= c(1, 10): la taille d'un objet plus long n'est pas multiple de
## la taille d'un objet plus court
```

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

est équivalent à :

```
x >= c(1, 10, 1, 10, 1, 10, 1, 10, 1)
```

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

Remarque: le message d'avertissement qui accompagne l'instruction précédente est très clair. **R** nous informe que les deux vecteurs qui sont comparés n'ont pas la même taille, mais un résultat sort quand même. Par ailleurs, l'instruction suivante ne renverra pas de message d'erreurs car le vecteur de droite a 3 éléments qui est un multiple de 9 (le nombre d'éléments du vecteur de gauche). Le message d'avertissement n'est donc pas systématique...

```
x >= c(1, 10, 11)
```

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

Pour les chaînes de caractère, on utilisera essentiellement l'opérateur == et l'opérateur **!=** :

```
y == "oui"
```

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

```
y != "oui"
```

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

Exercice 1.4

- est-ce que oui ou non les éléments de **my_num** sont supérieurs ou égales à la valeur 40
- est-ce que oui ou non les éléments les éléments de **my_char** sont égaux à "M"

1.5 Extraire des éléments d'un vecteur

Un sous-ensemble d'un vecteur peut être extrait en plaçant entre crochets un vecteur contenant les indices correspondant aux valeurs que l'on souhaite garder.

On peut par exemple extraire les 1er, 2ème, 4ème et 6ème valeurs du vecteur **x** ainsi :

```
x[c(1:2, 4, 6)]
```

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

A l'inverse, il est possible d'exclure ces mêmes valeurs en plaçant le signe "-" avant le vecteur d'indices :

```
x[-c(1:2, 4, 6)]
```

```
## [1]  3  5  7  9 11
```

Si un vecteur possède parmi ses attributs des valeurs pour l'attribut **names**, on peut sélectionner les éléments d'un vecteur en les appelant directement par leurs noms :

```
x <- c(a = 1, b = 2, c = 3, d = 4, e = 5, f = 6, g = 7, h = 9, i = 11)
x[c("a", "b", "d", "f")]
```

```
## a b d f
## 1 2 4 6
```

On peut également vouloir extraire d'un vecteur des éléments ayant une certaine caractéristique. Dans ces conditions, le procédé est le même en substituant le vecteur d'indice par une condition (employant les opérateurs logiques $<$, $<=$, $==$, $!=$, $>=$ et $>$). Prenons les vecteurs :

```
x <- c(1, 5, 3, 6, 4, 7, 2, 1)
y <- c("oui", "non", "oui", "oui", "non", "non", "oui", "oui", "peut-être", "non")
```

On extrait du vecteur **x** les valeurs de **x** inférieures ou égales à 3 par la commande :

```
x[x <= 3]
```

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

Cette opération consiste donc à mettre entre les crochets le vecteur de booléen suivant (de même longueur que **x**). Les éléments retournés sont ceux pour lesquels on a eu la valeur **TRUE** :

```
x <= 3
```

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

Pour sélectionner les valeurs égales à **"oui"** dans **y** on utilise l'opérateur $==$:

```
y[y == "oui"]
```

```
## [1] "oui" "oui" "oui" "oui" "oui"
```

Remarque : la négation d'un vecteur de booléen s'obtient avec le symbole **!** :

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

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

Remarque : nous verrons dans le prochain chapitre comment il est possible de sélectionner un sous-ensemble d'individus en utilisant plusieurs conditions à la fois.

Exercice 1.5

- Calculer la moyenne de **my_num** sur les éléments de **my_char** qui sont égaux à "M"
- Calculer la moyenne de **my_num** sur les éléments de **my_char** qui sont égaux à "F"

1.6 Fonctions qui génèrent des vecteurs

1.6.1 Fonction *rep()*

Pour construire un vecteur contenant 10 fois le nombre 2 :

```
rep(2, 10)
```

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

Pour construire un vecteur qui répète 5 fois chaque élément d'un vecteur :

```
rep(a.numeric, each = 5)
```

```
##  [1] 1.2 1.2 1.2 1.2 1.2 3.5 3.5 3.5 3.5 3.5 5.4 5.4 5.4 5.4 5.4 6.2 6.2 6.2 6.2
## [20] 6.2 8.1 8.1 8.1 8.1 8.1
```

1.6.2 Créer des séquences

1.6.2.1 Sur des nombres Pour construire la suite de 1 à n, on utilise le symbole *:* ainsi :

```
n <- 10
1:n
```

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

1.6.2.1.1 La fonction *seq()* Pour construire la suite de 1 à n avec un pas de 2 :

```
seq(1, n, by = 2)
```

```
## [1] 1 3 5 7 9
```

Pour construire *(K-1)* intervalles réguliers entre 1 et 10, on utilise également la fonction *seq()*, mais en utilisant l'argument d'entrée **length.out=** :

```
K <- 20
seq(1, n, length.out = K)
```

```
##  [1]  1.000000  1.473684  1.947368  2.421053  2.894737  3.368421  3.842105
##  [8]  4.315789  4.789474  5.263158  5.736842  6.210526  6.684211  7.157895
## [15]  7.631579  8.105263  8.578947  9.052632  9.526316 10.000000
```

La fonction *seq_along()* s'applique sur un vecteur et retourne le vecteur des n premiers entiers où n est la taille du vecteur.

```
seq_along(x)
```

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

Exercice 1.6 sachant que la fonction *length()* retourne le nombre d'élément d'un vecteur, proposer une autre façon de coder l'instruction précédente.

1.6.2.2 Sur des chaînes de caractères Pour construire l'alphabet en lettres minuscules :

```
letters
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z"
```

l'alphabet en majuscules :

```
LETTERS
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
## [20] "T" "U" "V" "W" "X" "Y" "Z"
```

1.6.2.3 Simulation de données

Pour construire un vecteur de taille *n=10* observations issues d'une loi gaussienne centrée et réduite $N(\mu = 0, \sigma^2 = 1)$, on utilise la fonction *rnorm()*. Le 1er argument donne la taille du vecteur à simuler, le deuxième l'espérance μ, le troisième l'écart-type σ :

```
x <- rnorm(n = n, mean = 0, sd = 1)
```

Remarque : lorsqu'on utilise une fonction prédéfinie de **R**, on n'est pas obligé de mettre le nom de tous les arguments d'entrée (ici, **n**, **mean** et **sd**) à condition de respecter l'ordre dans lequel ils apparaissent dans la fonction. Ainsi, on aurait pu utiliser la commande suivante plutôt que la précédente :

```
x <- rnorm(n, 0, 1)
```

On utilise la fonction *sample()* pour :

- faire des permutations aléatoires des éléments d'un vecteur :

```
sample(x)
```

```
##  [1]  0.76491059 -0.35747774  1.37738529  0.88681455  1.44386815 -0.01034568
##  [7] -1.84459725  0.25940664  1.25815194 -0.31649331
```

- tirer aléatoirement des éléments d'un vecteur. Dans l'exemple ci-dessous, on tire 10 observations sans remise (option **replace=FALSE**) dans un vecteur de taille 100 obtenu en simulant une loi gaussienne $N(\mu = 5, \sigma^2 = 4)$:

```
y <- rnorm(100, 5, 2)
sample(y, size = 10, replace = FALSE)
```

```
##  [1] -0.04338708  2.20506365  6.60231706  5.33340540  8.24150875  5.05267037
##  [7]  2.79039710  6.37065359  4.24400448  2.47653418
```

Remarque : il s'agit ici d'un tirage sans remise. Mais la fonction *sample()* peut aussi faire un tirage avec remise en utilisant l'option *replace=TRUE*, ce qui est très utile pour coder des algorithmes de type Bootstrap.

Exercice 1.7

Créer un vecteur **my__num__b** qui est un échantillon bootstrap de **my__num**

1.6.3 Concaténation

1.6.3.1 Concacaténation de vecteurs

Ici, le but est de prendre deux vecteurs de taille n_1 et n_2 et de construire un vecteur de taille $n_1 + n_2$. Pour cela, on utilise de nouveau la fonction *c()* : lorsque les éléments sont de type différents (par exemple **numeric** et **character**), alors c'est le type **character** qui l'emporte. Par exemple, l'instruction suivante va créer un vecteur de **character** :

```
y <- c("R", "est", "chouette")
c(x, y)
```

```
##  [1] "0.259406641500008"  "-1.84459725367722"  "-0.316493314975091"
##  [4] "1.37738529423077"   "0.764910589353403"  "1.25815194321671"
##  [7] "0.886814551007378"  "-0.010345681694477" "-0.357477744236335"
## [10] "1.44386814636818"   "R"                  "est"
## [13] "chouette"
```

1.6.3.2 Concacaténation des éléments d'un vecteur

Ici on a deux vecteurs de même taille n_1 et on souhaite concaténer les éléments de ces vecteurs 1 par 1. Pour cela, on utilise la fonction *paste()*. L'argument **sep=** permet d'indiquer quelle chaîne de caractère va séparer les éléments des deux vecteurs. Par exemple :

```
paste(letters, LETTERS, sep = " - ")
```

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

Dans le cas où la taille de chaque élément qu'on doit concaténer est différente, on répète chaque élément (équivalent de la fonction *rep()*), de sorte que chaque élément soit de la taille de l'élément le plus grand. Finalement, on va créer un objet dont la taille est égale à l'objet qui a la plus grande taille. Mais rien ne vaut un exemple pour expliquer la fonction *paste()*.

```
paste("L'observation numéro ", 1:n, " (nombre ", c("impair", "pair"), ") sur ", n,
      " est égal à ", round(x, 4), sep = "")
```

```
##  [1] "L'observation numéro 1 (nombre impair) sur 10 est égal à 0.2594"
##  [2] "L'observation numéro 2 (nombre pair) sur 10 est égal à -1.8446"
##  [3] "L'observation numéro 3 (nombre impair) sur 10 est égal à -0.3165"
##  [4] "L'observation numéro 4 (nombre pair) sur 10 est égal à 1.3774"
##  [5] "L'observation numéro 5 (nombre impair) sur 10 est égal à 0.7649"
##  [6] "L'observation numéro 6 (nombre pair) sur 10 est égal à 1.2582"
##  [7] "L'observation numéro 7 (nombre impair) sur 10 est égal à 0.8868"
##  [8] "L'observation numéro 8 (nombre pair) sur 10 est égal à -0.0103"
##  [9] "L'observation numéro 9 (nombre impair) sur 10 est égal à -0.3575"
## [10] "L'observation numéro 10 (nombre pair) sur 10 est égal à 1.4439"
```

Noter qu'il n'y a aucun message d'avertissement pour dire que les tailles des vecteurs ne sont pas multiples l'une de l'autre.

Exercice 1.8

Trouver les lignes de codes qui permettent d'afficher le résultat suivant:

```
##  [1] "L'élément 1 de my_num vaut 28 et celui de my_char vaut M"
##  [2] "L'élément 2 de my_num vaut 29 et celui de my_char vaut M"
##  [3] "L'élément 3 de my_num vaut 35 et celui de my_char vaut M"
##  [4] "L'élément 4 de my_num vaut 75 et celui de my_char vaut M"
##  [5] "L'élément 5 de my_num vaut 40 et celui de my_char vaut F"
##  [6] "L'élément 6 de my_num vaut 52 et celui de my_char vaut M"
##  [7] "L'élément 7 de my_num vaut 23 et celui de my_char vaut F"
##  [8] "L'élément 8 de my_num vaut 25 et celui de my_char vaut F"
##  [9] "L'élément 9 de my_num vaut 10 et celui de my_char vaut F"
## [10] "L'élément 10 de my_num vaut 50 et celui de my_char vaut F"
```

1.7 Obtenir des informations sur les vecteurs

Plusieurs fonctions permettent d'obtenir des informations sur un vecteur, informations qui peuvent concerner ses attributs comme son contenu en lui-même. Une première série de fonctions concerne les attributs du vecteur. Nous avons déjà vu :

- *class(x)* permet de savoir à quel type appartient les éléments de **x**.

```
class(x)
```

```
## [1] "numeric"
```

```
class(y)
```

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

- *length(x)* permet d'obtenir sa longeur.

```
z <- c(x, NA, y)
length(z)
```

```
## [1] 14
```

- *names(x)* fournit, s'il existe, le nom des éléments d'un vecteur.

D'autres fonctions permettent d'apporter des informations intéressantes sur les vecteurs :

- On obtient l'ensemble des valeurs prises par un vecteur par *unique(x)*.

```
y <- c("yes", "yes", "no", "yes", "no")
unique(y)
```

```
## [1] "yes" "no"
```

- *is.na(x)* indique quels sont les éléments du vecteur **x** qui sont manquants au moyen des booléens **TRUE** (manquant) et **FALSE** (non manquant).

```
is.na(z)
```

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

- *which()* appliquée sur un vecteur de booléen retourne les indices des composantes du vecteur égales à **TRUE**. En général, on l'utilise après avoir effectué une comparaison pour connaître les indices des observations qui satisfont une condition particulière. Ici, on souhaite connaître les indices du vecteur dont les composantes sont positives :

```
x > 0
```

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

```
which(x > 0)
```

```
## [1]  1  4  5  6  7 10
```

Exercice 1.9

- Donner les positions dans **my_num** des éléments qui sont supérieurs à 20
- Donner les positions dans **my_char** des éléments qui sont égaux à "F"

1.8 Les factor

Le type **factor** est un type particulier sur lequel il existe un certain nombre de contraintes. C'est pourquoi nous ne l'avons pas inclus avec les autres (**numeric**, **integer**, **character**, **logical**). Un **factor** est un vecteur qui contient uniquement des valeurs prédéfinies et est utilisé pour décrire une variable qualitative.

Pour créer un **factor**, on peut utiliser la fonction *factor()* sur un vecteur de **character**. Par exemple :

```
y <- c("healthy", "healthy", "failing", "failing", "healthy", "failing", "healthy",
       "failing", "failing", "failing")
yf <- factor(y)
yf
```

```
##  [1] healthy healthy failing failing healthy failing healthy failing failing
## [10] failing
## Levels: failing healthy
```

Au moment de sa création, un **factor** possède dans ses attributs les **levels** (modalités) qui sont autorisées.

```
levels(yf)
```

```
## [1] "failing" "healthy"
```

Par contre, il n'est pas possible d'ajouter une nouvelle modalité. Ici, on essaie de modifier le 2ème élément du vecteur par une modalité qui n'est pas prédéfinie et ceci crée un message d'avertissement nous signalant qu'une valeur manquante (NA) a été générée pour cause de modalité inconnue :

```
yf[2] <- "good"
```

```
## Warning in `[<-.factor`(`*tmp*`, 2, value = "good"): niveau de facteur
## incorrect, NAs générés
```

```
yf
```

```
##  [1] healthy <NA>    failing failing healthy failing healthy failing failing
## [10] failing
## Levels: failing healthy
```

Remarque : le **factor** est intéressant pour coder une variable qualitative si on connaît à l'avance le nombre de modalités.

1.9 Statistiques sur des vecteurs

En statistique, on distingue deux types de variables :

- les variables quantitatives (continues ou discrètes), qu'on peut assimiler à des vecteurs de **numeric** ou **integer**.
- les variables qualitatives (nominales ou ordinales), qu'on peut assimiler à des vecteurs de **character** ou à un **factor**.

Selon le type de variable, on n'utilisera pas les mêmes fonctions :

1.9.1 Fonctions utiles pour l'analyse de variables qualitatives

En ce qui concerne les variables qualitatives, la fonction la plus utile est la fonction *table()*, qui fournit le tableau de fréquences absolues (i.e. les effectifs) par modalités.

```
table(yf)
```

```
## yf
## failing healthy
##       6       3
```

Remarque : par défaut, en sortie, les modalités d'un objet de type **factor** sont triées par ordre alphabétique. Or, si on travaille sur une variable qualitative ordinale, cela implique que les modalités sont triées par ordre de grandeur. Considérons la variable qualitative de taille 20 qui indique si une personne a aimé un film, avec les modalités suivantes : "pas du tout", "un peu", "moyennement", "beaucoup". Pour la coder, on va le faire de la façon suivante. On assimile à chaque modalité la valeur 0 pour "pas du tout", 1 pour "un peu", 2 pour "moyennement", 3 pour "beaucoup". On part donc d'un vecteur d'entiers :

```
film_num <- c(0, 3, 1, 2, 1, 2, 3, 2, 1, 0, 1, 2, 3, 2, 1, 2, 1, 3, 3, 2)
```

On va utiliser dans la fonction *factor()*, les options **levels=** et **labels=** qui vont contenir respectivement les modalités et les étiquettes associées qu'on autorise. De plus, on ajoute l'option **ordered = TRUE** pour préciser que nous avons trié les modalités par ordre de grandeur :

```
film_factor <- factor(film_num, levels = c(0, 1, 2, 3),
                      labels = c("pas_du_tout", "un_peu", "moyennement", "beaucoup"),
```

```
                      ordered = TRUE)
film_factor
```

```
##  [1] pas_du_tout beaucoup    un_peu      moyennement un_peu      moyennement
##  [7] beaucoup    moyennement un_peu      pas_du_tout un_peu      moyennement
## [13] beaucoup    moyennement un_peu      moyennement un_peu      beaucoup
## [19] beaucoup    moyennement
## Levels: pas_du_tout < un_peu < moyennement < beaucoup
```

La fonction *tab()* appliquée à un **factor** ordonné permet de retourner les modalités triées par ordre de grandeur (et non plus par ordre alphabétique) :

```
tab <- table(film_factor)
tab
```

```
## film_factor
## pas_du_tout      un_peu moyennement    beaucoup
##           2           6           7           5
```

Le tableau de fréquences relatives est obtenu avec la fonction *prop.table()* appliquée à un objet de classe **table**.

```
prop <- prop.table(tab)
prop
```

```
## film_factor
## pas_du_tout      un_peu moyennement    beaucoup
##        0.10        0.30        0.35        0.25
```

Ici, on supperpose d'abord le vecteur des effectifs avec le vecteur des proportions, en utilisant la fonction *rbind()* :

```
tab.res <- rbind(tab, prop)
tab.res
```

```
##      pas_du_tout un_peu moyennement beaucoup
## tab          2.0    6.0        7.00     5.00
## prop         0.1    0.3        0.35     0.25
```

La fonction *addmargins()* permet d'ajouter une marge à un tableau. L'option **margin = 2** précise ici que la marge est ajoutée sur la 2ème dimension du tableau. On rappelle qu'un tableau est en 2D. La première dimension est l'espace des lignes, la seconde, l'espace des colonnes. Ici, on ajoute donc une colonne au tableau sur laquelle a été calculée la somme de chaque ligne.

```
addmargins(tab.res, margin = 2)
```

```
##      pas_du_tout un_peu moyennement beaucoup Sum
## tab          2.0    6.0        7.00     5.00  20
## prop         0.1    0.3        0.35     0.25   1
```

Le tableau de fréquences cumulées s'obtient avec la fonction *cumsum()*. Ceci s'applique bien évidemment uniquement aux variables qualitatives ordinales (et aussi aux variables quantitatives discrètes).

```
cumsum(prop)
```

```
## pas_du_tout      un_peu moyennement    beaucoup
##        0.10        0.40        0.75        1.00
```

Comment lire le tableau ci-dessus : 75% des personnes interrogées ont trouvé le film "moyen" ou plus négatif encore.

Complément : le lecteur pourra consulter la note suivante frequences_cumulees.pdf pour avoir un exemple d'analyse d'une variable quantitative discrète, dont le raisonement est proche de celui qu'on vient de voir pour une variable qualitative ordinale.

Exercice 1.10

Calculer la table d'éffectifs de **my_char** et la table de proportions

1.9.2 Fonctions utiles pour l'analyse de variables quantitatives

Pour les variables quantitatives, les fonctions sont plus nombreuses. Prenons l'exemple d'un vecteur de taille 60 issu d'une loi gaussienne $N(100, 20)$:

```
set.seed(123)
x.quant <- rnorm(60, 100, sqrt(20))
```

La somme et la moyenne sont obtenues ainsi :

```
sum(x.quant)
```

```
## [1] 6017.607
```

```
mean(x.quant)
```

```
## [1] 100.2934
```

La médiane :

```
median(x.quant)
```

```
## [1] 100.0938
```

L'écart-type (resp. variance) est obtenu par la commande :

```
sd(x.quant)
```

```
## [1] 4.071326
```

```
var(x.quant)
```

```
## [1] 16.57569
```

Pour obtenir le minimum et le maximum :

```
min(x.quant)
```

```
## [1] 91.20502
```

```
max(x.quant)
```

```
## [1] 109.6999
```

```
# ou directement
range(x.quant)
```

```
## [1]  91.20502 109.69987
```

La fonction *quantile()* retourne directement le minimum, les quartiles et le maximum :

```
quantile(x.quant)
```

```
##        0%       25%       50%       75%      100%
##  91.20502  97.79276 100.09383 103.09391 109.69987
```

Par défaut la fonction *quantile()* retourne les quantiles d'ordre 0, 25%, 50%, 75%, 100%. On peut modifier ces valeurs avec l'option **probs=**.

Remarque : il y a plusieurs définitions possibles pour caculer les quantiles. Pour les connaître, lire attentivement l'aide de la fonction :

```
?quantile
```

Rappel sur les quantiles : vous pouvez regarder cette vidéo pour un rappel sur les quantiles https://www.youtube.com/watch?v=z_Yyh4DoX68

Exercice 1.11

Calculer à partir de la formule $V(X) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2$ la variance de **x.quant** en utilisant les fonctions *mean()* et *sum()*. Que constatez-vous par rapport au résultat donné par la fonction *var()* ?

1.10 Un peu de calcul vectoriel

Toutes les opérations élémentaires (+, -, *, /) peuvent être appliquées sur les vecteurs (numériques). Les opérations sont alors effectuées élément par élément. Prenons l'exemple des vecteurs $x = (1, 0, 1, 1, 1)'$ et $y = (1, 1, 0, 0, 1)'$.

```
x <- c(1, 1, 1, 1, 1)
y <- c(1, 1, 0, 0, 1)
```

La commande :

```
(z <- 2 * x + 3 * y - x * y + 1)
```

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

affecte au nouvel objet z le vecteur (de longueur 5) suivant :

$$2\begin{pmatrix}1\\1\\1\\1\\1\end{pmatrix} + 3\begin{pmatrix}1\\1\\0\\0\\1\end{pmatrix} - \begin{pmatrix}1\times 1\\1\times 1\\0\times 1\\0\times 1\\1\times 1\end{pmatrix} + \begin{pmatrix}1\\1\\1\\1\\1\end{pmatrix}$$

Explication : sur l'exemple précédent, on remarque que les dimensions ne sont pas les mêmes entre les vecteurs x, y et le scalaire 1. Pourtant, **R** produit quand même un résultat en sortie. En fait, en ce qui concerne les vecteurs (ce n'est pas le cas pour les matrices), si tous les vecteurs ne sont pas de même longueur, le logiciel recycle autant de fois que nécessaire (avec d'éventuelles troncatures) les vecteurs les plus petits pour les ramener à la taille du plus grand. Dans l'exemple, le scalaire 1 est ainsi transformé en un vecteur $(1, 1, 1, 1, 1)'$.

Il existe un certain nombre de commandes correspondant à des fonctions mathématiques et qui, appliquées à des vecteurs, effectuent le calcul sur chaque élément du vecteur :

```
z ^ 2
```

```
## [1] 25 25  9  9 25
```

```
exp(1:10)
```

```
##  [1]     2.718282     7.389056    20.085537    54.598150   148.413159
##  [6]   403.428793  1096.633158  2980.957987  8103.083928 22026.465795
```

```
cos(seq(0, 2 * pi, by = pi / 4))
```

```
## [1]  1.000000e+00  7.071068e-01  6.123234e-17 -7.071068e-01 -1.000000e+00
## [6] -7.071068e-01 -1.836970e-16  7.071068e-01  1.000000e+00
```

```
asin(0.5)
```

```
## [1] 0.5235988
```

```
tan(pi / 2)
```

```
## [1] 1.633124e+16
```

```
log(100)
```

```
## [1] 4.60517
```

```
log10(c(10, 100, 1000))
```

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

Exercice 1.12

Calculer le logarithme du vecteur **my_num**.

1.11 Ordonner un vecteur/permuter les éléments d'un vecteur

D'autres fonctions permettent de réorganiser un vecteur. *sort(x)* ordonne un vecteur par ordre croissant :

```
set.seed(seed = 123)
x <- rnorm(10, 0, 1)
sort(x)
```

```
##  [1] -1.26506123 -0.68685285 -0.56047565 -0.44566197 -0.23017749  0.07050839
##  [7]  0.12928774  0.46091621  1.55870831  1.71506499
```

Remarque: la fonction *set.seed()* que nous reverrons plus tard précède une fonction qui va générer des nombres aléatoires. En fixant l'argument **seed=** avec un nombre fixe, cela permet de générer systématiquement le même vecteur. Pour vous en convaincre, méditer sur l'exemple suivant :

```
set.seed(seed = 123)
x <- rnorm(10, 0, 1)
x
```

```
##  [1] -0.56047565 -0.23017749  1.55870831  0.07050839  0.12928774  1.71506499
##  [7]  0.46091621 -1.26506123 -0.68685285 -0.44566197
```

```
x <- rnorm(10, 0, 1)
x
```

```
##  [1]  1.2240818  0.3598138  0.4007715  0.1106827 -0.5558411  1.7869131
##  [7]  0.4978505 -1.9666172  0.7013559 -0.4727914
```

```
x <- rnorm(10, 0, 1)
x
```

```
##  [1] -1.0678237 -0.2179749 -1.0260044 -0.7288912 -0.6250393 -1.6866933
##  [7]  0.8377870  0.1533731 -1.1381369  1.2538149
```

```
set.seed(seed = 123)
x <- rnorm(10, 0, 1)
x
```

```
##  [1] -0.56047565 -0.23017749  1.55870831  0.07050839  0.12928774  1.71506499
##  [7]  0.46091621 -1.26506123 -0.68685285 -0.44566197
```

La fonction *order(x)* ordonne les indices d'un vecteur. Par exemple, le plus petit élément de **x** correspond au 8ème, le deuxième plus petit élément est le 9ème indice, etc. :

```
order(x)
```

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

La fonction *rev()* inverse les indices de **x**

```
rev(x)
```

```
##  [1] -0.44566197 -0.68685285 -1.26506123  0.46091621  1.71506499  0.12928774
##  [7]  0.07050839  1.55870831 -0.23017749 -0.56047565
```

Exercice 1.13

Proposer une autre façon de coder la commande précédente.

La fonction *sample()* (déjà vue précédemment) génère un nouvel échantillon de façon aléatoire, à partir des valeurs du vecteur.

```
sample(x)
```

```
##  [1] -0.68685285 -0.44566197 -0.56047565  0.12928774  1.55870831 -0.23017749
##  [7]  1.71506499  0.46091621 -1.26506123  0.07050839
```

2 Les matrices

2.1 Introduction

Un objet **matrix** sous **R** peut être vu comme une généralisation du vecteur (1D) en 2 dimensions. Il contient des éléments du même type (**character**, **numeric**, **integer**, **logical**). En règle générale, elles sont surtout utiles pour contenir des valeurs numériques. Par ailleurs, en statistique, si on observe p variables quantitatives (i.e. p vecteurs de valeurs **numeric** ou **integer**) sur n observations, on disposera ces variables sous forme d'une matrice à n lignes et p colonnes où chaque colonne correspond à une variable. On pourra être amener à faire des opérations matricielles afin d'appliquer des méthodes de la statistique multidimensionnelle telles que l'ACP (analyse en composantes principales), la CAH (classification ascendante hierarchique), etc. (pour des rappels sur l'ACP, consulter cette page et pour la CAH, consulter cette page). Une matrice est donc une table à deux dimensions, la première dimension correspond aux lignes et la deuxième dimension, aux colonnes.

Une matrice de taille $n \times p$ se construit à l'aide de la fonction *matrix()* (ou *as.matrix(x)*), à partir d'un vecteur **x** de longeur np contenant les données.

Par exemple, la matrice $X = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ se construit par la commande :

```
X <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 2, ncol = 3, byrow = T)
```

L'option **nrow = 2** précise la valeur de la 1ère dimension (le nombre de lignes n), **ncol = 3** précise la valeur de la deuxième dimension (le nombre de colonnes p). Quand on précise l'option **byrow = T**, on précise qu'on va remplir la matrice à partir du vecteur **x**, ligne après ligne.

Si on se trompe sur les valeurs de la dimension de la matrice, on ne recevra pas nécessairement un message d'avertissement. Par exemple, dans notre exemple, les seules matrices qui puissent contenir un vecteur de taille 6 sont les matrices de dimensions suivantes : 6×1, 3×2, 2×3, 1×6. Si on se trompe sur ces dimensions, cela va quand même créer une matrice de la taille demandée, soit en coupant des valeurs de **x** (si $np < 6$), soit en répétant le vecteur **x** lui-même (si $np > 6$). Voir les deux exemples ci-dessous :

```
matrix(c(1, 2, 3, 4, 5, 6), nrow = 2, ncol = 2, byrow = T)
```

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

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

```
matrix(c(1, 2, 3, 4, 5, 6), nrow = 3, ncol = 3, byrow = T)
```

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

Si on ne renseigne pas toutes les options, elles seront calculées automatiquement. Ci-dessous **R** comprend qu'en créant une matrice à 3 lignes à partir d'un vecteur de longueur 6, cela implique que le nombre de colonnes est 2.

```
matrix(c(1, 4, 2, 5, 3, 6), nrow = 3)
```

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

2.2 Attributs d'une matrice

Les principales caractéristiques d'une matrice sont données par :

- la dimension de la matrice (nombre de lignes, nombre de colonnes) :

```
dim(X)
```

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

Ces 2 informations peuvent être obtenues séparément par les commandes :

```
nrow(X)
```

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

```
ncol(X)
```

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

- s'ils existent, le nom des lignes (fonction *rownames()*) et le nom des colonnes (fonction *colnames()*). Par défaut, ces dernières ne sont pas renseignées. Pour donner des noms de lignes et noms de colonnes, ceci peut se faire de la façon suivante :

```
X
```

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

```
rownames(X) <- c("ind1", "ind2")
colnames(X) <- paste("V", 1:3, sep = "")
X
```

```
##      V1 V2 V3
## ind1  1  2  3
## ind2  4  5  6
```

Exercice 1.14

Construire les matrices suivantes:

$$A = \begin{pmatrix} 1 & 0 & 1 \\ 3 & 2 & 1 \end{pmatrix} \text{ et } B = \begin{pmatrix} 1 & 0 \\ 2 & 1 \\ 3 & 2 \end{pmatrix}$$

2.3 Opérations élémentaires

On considère les deux matrices suivantes :

```
A <- matrix(c(1, 0, 1, 3, 2, 1), ncol = 3, byrow = T)
B <- matrix(c(1, 2, 3, 0, 1, 2), ncol = 2, byrow = F)
```

Les opérateurs élémentaires +, -, * et / réalisent les opérations terme à terme, sous réserve que les 2 matrices soient de la même dimension. Il est par contre possible d'effectuer une opération entre un scalaire et une matrice. A l'instar de ce que nous avons vu pour les vecteurs, la commande 1 + 2*A effectuera la somme des matrices

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} + \begin{pmatrix} 2 \times 1 & 2 \times 0 & 2 \times 1 \\ 2 \times 3 & 2 \times 2 & 2 \times 1 \end{pmatrix}$$

Pour le produit de matrice, il faut utiliser l'opérateur %*%, toujours sous réserve de l'adéquation des dimensions. Dans notre exemple, cela donne:

```
A %*% B
```

```
##      [,1] [,2]
## [1,]    4    2
## [2,]   10    4
```

La fonction *t()* permet d'obtenir la transposée d'une matrice :

```
t(X)
```

```
##    ind1 ind2
## V1    1    4
## V2    2    5
## V3    3    6
```

La fonction *crossprod(X, A)* permet de faire le produit matriciel : $X^T \times A$, où X^T (également notée X') est la transposée de la matrice X .

```
crossprod(X, A)
```

```
##    [,1] [,2] [,3]
## V1   13    8    5
## V2   17   10    7
## V3   21   12    9
```

```
# équivalent à
t(X) %*% A
```

```
##    [,1] [,2] [,3]
## V1   13    8    5
## V2   17   10    7
## V3   21   12    9
```

2.4 Extraction d'un sous-ensemble

Nous avons vu que l'on pouvait extraire d'un vecteur quelques éléments. Pour les matrices, l'opération est identique, à la différence près que l'on doit indiquer entre crochets les individus et les variables à retenir.

Par exemple, les observations de la 2ème et de la 3ème variable contenues dans la matrice A pour le 2ème individu sont données par :

```
X[2, c(2, 3)]
```

```
## V2 V3
##  5  6
```

De même, si les noms de lignes et colonnes ont été définies, on peut appeler des éléments à partir de ces noms :

```
X["ind2", c("V2", "V3")]
```

```
## V2 V3
##  5  6
```

2.5 Faire un calcul par ligne (ou colonne) : fonction *apply()*

Nous avons vu qu'une matrice servait souvent à stocker les observations sur des individus de plusieurs variables. Il est donc intéressant de savoir extraire des informations sur chacune des colonnes. Par ailleurs, les fonctions traditionnelles (*sum()*, *sd()*, *mean()*, etc) considèrent une matrice de dimension $n \times p$ comme un vecteur de longueur np. Par exemple :

```
sum(A)
```

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

```
mean(A)
```

```
## [1] 1.333333
```

Aussi, pour obtenir des informations sur chacune des colonnes, il est nécessaire d'utiliser la fonction *apply()*. Par exemple la somme des colonnes de la matrice A peut être effectuée par la commande :

```
apply(A, 2, sum)
```

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

Le premier argument correspond au nom de la matrice, le second à la dimension sur laquelle on va effectuer un calcul (1 pour un calcul par ligne et 2 pour un calcul par colonne) et le dernier argument correspond à la fonction à appliquer. Pour obtenir la moyenne par ligne, on fait :

```
apply(A, 1, mean)
```

```
## [1] 0.6666667 2.0000000
```

Exercice 1.15

Faire la moyenne par colonne des éléments de B

2.6 Fonctions utiles pour les matrices

Les fonctions *cbind()* et *rbind()* permettent de concaténer (verticalement ou horizontalement) des matrices. Ces deux fonctions marchent également avec des data frame. Par exemple :

```
rbind(A, X)
```

```
##      V1 V2 V3
##       1  0  1
##       3  2  1
## ind1  1  2  3
## ind2  4  5  6
```

```
cbind(A, t(B))
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    0    1    1    2    3
## [2,]    3    2    1    0    1    2
```

Les fonctions ci-dessous sont très souvent utilisées lorsque l'on manipule des matrices.

- *t(A)* donne la transposé de la matrice A.
- *solve(A, b)* donne le vecteur x qui résoud l'équation $Ax = b$ où A est une matrice carré.

```
solve(A%*%B, c(1, 1))
```

```
## [1] -0.5  1.5
```

- Un cas particulier d'utilisation de la fonction *solve()* est le suivant. Si on ne précise pas le vecteur b, *solve(A)* résoud le système suivant $Ax = I$ où I est la matrice identité et A doit être une matrice carrée (c'est-à-dire où le nombre de lignes égale le nombre de colonnes). Autrement dit, la solution de ce système est la matrice A^{-1} qui n'est autre que l'inverse de A. Pour calculer l'inverse d'une matrice carrée, on peut donc utiliser la fonction *solve()* de la façon suivante. Dans l'exemple suivant, pour avoir une matrice carrée, on calcule $B'B$ grâce à la fonction *crossprod()* vue précédemment :

```
solve(crossprod(B))
```

```
##            [,1]      [,2]
## [1,]  0.8333333 -1.333333
## [2,] -1.3333333  2.333333
```

Remarque : dans les modèles de régression linéaire, l'estimateur $\hat{\beta}$ des moindres carrés est souvent présenté comme étant égal à $(X'X)^{-1}X'y$. D'un point de vue numérique, pour calculer $\hat{\beta}$, il est plus efficace de le présenter comme étant la solution du système $(X'X)\beta = X'y$.

- Lorsqu'on souhaite inverser une matrice carrée symétrique définie positive, comme c'est le cas lorsqu'on souhaite inverser la matrice $(X'X)$, il est alors recommander d'utiliser une factorisation de Cholesky avec la fonction *chol()* et d'inverser le résultat de cette factorisation avec la fonction *chol2inv*:

```
chol2inv(chol(crossprod(B)))
```

```
##            [,1]      [,2]
## [1,]  0.8333333 -1.333333
## [2,] -1.3333333  2.333333
```

- Pour inverser une matrice, on peut également utiliser sa décomposition QR qui s'obtient avec la fonction *qr()* et appliquer la fonction *qr.solve()* sur cette décomposition :

```
qr.solve(qr(crossprod(B)))
```

```
##            [,1]      [,2]
## [1,]  0.8333333 -1.333333
## [2,] -1.3333333  2.333333
```

Pour un rappel sur les décompositions de matrice, on pourra consulter : https://www.math.univ-toulouse.fr/~ccourtes/Cours_Decomposition.pdf

- *diag(A)* calcule la diagonale de la matrice A carré. Si a est un vecteur, le résultat est une matrice diagonale avec le vecteur a sur la diagonale.

```
diag(A%*%B)
```

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

```
diag(1:5)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0    0    0
## [2,]    0    2    0    0    0
## [3,]    0    0    3    0    0
## [4,]    0    0    0    4    0
## [5,]    0    0    0    0    5
```

Compléments : pour plus de détails sur le calcul matriciel, le lecteur pourra consulter la note suivante Matrices.pdf

3 Les listes

3.1 Définition

Les listes sont différents des vecteurs car ils peuvent être composés d'éléments de tous types. Une liste peut contenir par exemple à la fois un vecteur de **numeric**, une matrice de **character** et autant d'éléments que l'on souhaite.

Prenons l'exemple suivant, tiré de l'ouvrage d'Hadley Wickham (https://adv-r.hadley.nz/vectors-chap.html#lists). L'objet **a** a pour composantes :

1. un vecteur d'entiers
2. une chaîne de caractères,
3. un scalaire,
4. une liste, composé de deux élements (1 scalaire et 1 scalaire).

Cet exemple, comme toutes les listes, est construit à l'aide de la fonction *list()* (et non pas la fonction *c()*) :

```
a <- list(un_vecteur = 1:3,
          un_character = "a string",
          un_scalaire = pi,
          une_liste = list(-1, -5))
```

Pour accéder aux éléments d'une liste dont les éléments ont un nom (comme c'est le cas ici) , on peut utiliser le signe **$**, suivi du nom de la composante à récupérer. Par exemple :

```
a$un_vecteur
```

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

Ceci est identique à utiliser les doubles crochets et la position de l'élément de la liste qu'on souhaite extraire :

```
a[[1]]
```

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

Si on utilise qu'un seul crochet, cela aura pour conséquence d'extraire l'élement de la liste, mais de conserver l'objet sous forme d'une liste. Dans l'exemple ci-dessous, l'objet retourné est une liste :

```
a[1]
```

```
## $un_vecteur
## [1] 1 2 3
```

Ceci permet de sélectionner un sous-ensemble de la liste, comme dans l'exemple suivant :

```
a[1:2]
```

```
## $un_vecteur
## [1] 1 2 3
##
## $un_character
## [1] "a string"
```

La différence entre l'utilisation d'un seul crochet ou de deux crochets est illustrée dans la figure suivante.

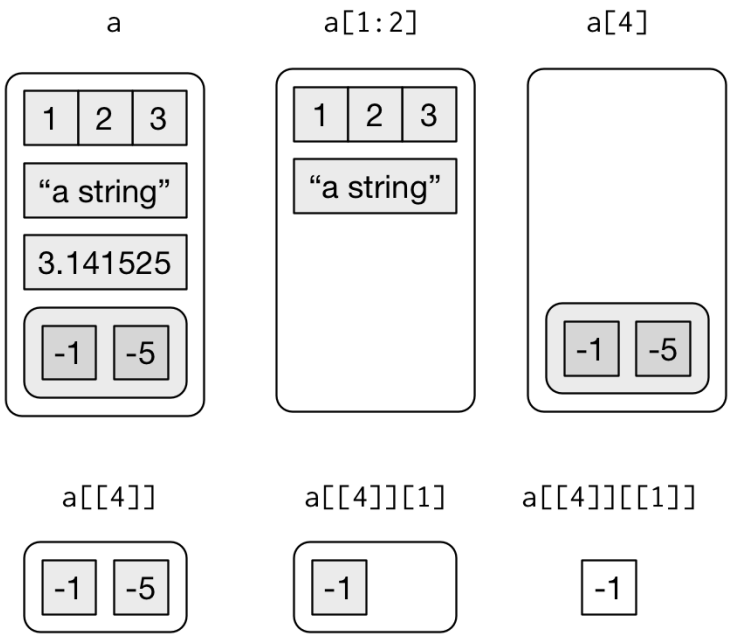

Figure 1: Exemple de manipulation d'une liste

3.2 Attributs

Les trois principaux attributs d'une **list** sont :

- sa longueur (obtenue avec la fonction *length()*),

```
length(a)
```

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

- le nom de ses composants (fonction *names()*),

```
names(a)
```

```
## [1] "un_vecteur"   "un_character" "un_scalaire"  "une_liste"
```

- le type des éléments qui composent la liste (fonction *str()*) :

```
str(a)
```

```
## List of 4
##  $ un_vecteur  : int [1:3] 1 2 3
##  $ un_character: chr "a string"
##  $ un_scalaire : num 3.14
##  $ une_liste   :List of 2
```

```
##   ..$ : num -1
##   ..$ : num -5
```

De nouveaux éléments peuvent être ajoutés à l'aide du symbole **$**. Dans notre exemple, on peut ajouter la composante **une_matrice**:

```
a$une_matrice <- matrix(c(1, 0, 1, 3, 2, 1),
                        ncol = 3, byrow = T)
```

De par sa nature, un objet de type **list** est donc idéal pour stocker des informations de différente nature sur un individu, un phénomène, ... Il sera notamment très utile pour stocker les résultats lors de la construction d'une fonction (dont nous verrons les grands principes lors du chapitre suivant).

3.3 Faire un calcul sur chaque élément d'une liste : fonction *lapply()*

Pour appliquer une fonction à chaque élément d'une liste, on utilise la fonction *lapply()* avec comme premier argument le nom de la liste et comme second argument, la fonction à appliquer à chaque élément de liste. Par exemple, pour connaître la taille de chaque élément de la liste, on fait :

```
lapply(a, length)
```

```
## $un_vecteur
## [1] 3
##
## $un_character
## [1] 1
##
## $un_scalaire
## [1] 1
##
## $une_liste
## [1] 2
##
## $une_matrice
## [1] 6
```

Remarque 1: quand on applique la fonction *length()* à une matrice, cela renvoie le nombre de cellules de la matrice.

Remarque 2: quand nous aurons vu comment créer ses propres fonctions, on pourra appliquer comme deuxième argument de cette fonction les fonctions que nous aurons créées.

Exercice 1.16

- Créer un objet **my_list** qui contient deux éléments : le vecteur **my_char** qu'on appelera **var_1** et le vecteur **my_num** qu'on appelera **var_2**
- Calculer la taille de chaque élément de la liste

4 Les data.frame

4.1 Présentation

Les **data.frame** sont les objets les plus importants dans **R**, du point de vue du statisticien. Il s'agit d'un tableau à deux dimensions, dont les colonnes, à l'inverse des matrices, peuvent être de différents types. Cela permet donc de pouvoir disposer, dans un seul objet, des observations sur de nombreux individus de variables quantitatives et qualitatives, formes habituelles d'un jeu de données. On partira du principe que la dimension des lignes correspond aux n observations et les colonnes correspondent aux p variables.

Un **data.frame** hérite à la fois des propriétés des **matrix** ainsi que des **list**.

La construction d'un **data.frame** peut se faire en contraignant un objet à devenir une structure de données (avec *as.data.frame()*) ou, ce qui est plus rare, en la construisant directement avec la fonction *data.frame()*. Mais la plupart du temps, les **data.frame** proviennent de l'importation de fichiers de données de type "txt", "csv", "xls", etc. (voir le paragraphe "Importation et Exportation de données" dans la deuxième partie de ce chapitre).

Si l'on considère les vecteurs suivants correspondant à des caractéristiques observées sur 6 étudiants :

```
age <- c(20, 21, 20, 25, 29, 22)
taille <- c(165, 155, 150, 170, 175, 180)
sexe <- c("F", "F", "F", "M", "M", "M")
```

On créé un **data.frame** avec l'instruction suivante :

```
don <- data.frame(age, taille, sexe)
don
```

```
##   age taille sexe
## 1  20    165    F
## 2  21    155    F
## 3  20    150    F
## 4  25    170    M
## 5  29    175    M
## 6  22    180    M
```

4.2 Attributs

Parmi les caractéristiques d'un **data.frame** :

- sa dimension (nombre de lignes $\times$ nombre de colonnes)

```
dim(don)
```

```
## [1] 6 3
```

équivalent à utiliser les fonctions *nrow()* et *ncol()* :

```
nrow(don)
```

```
## [1] 6
```

```
ncol(don)
```

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

- les noms des lignes et colonnes données par les fonctions *row.names()* et *colnames()* (ou *names()* comme pour les **list**) :

```
row.names(don)
```

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

```
colnames(don)
```

```
## [1] "age"    "taille" "sexe"
```

Remarque : par défaut, il est donné un nom à chaque ligne qui correspond aux numéros de lignes exprimés en chaînes de caractères. Pour changer le nom des lignes, on fait :

```
row.names(don) <- c("sonia", "maud", "iris", "mathieu", "amin", "gregory")
```

- le type des colonnes donné par la fonction *str()* :

```
str(don)
```

```
## 'data.frame':    6 obs. of  3 variables:
##  $ age   : num  20 21 20 25 29 22
##  $ taille: num  165 155 150 170 175 180
##  $ sexe  : chr  "F" "F" "F" "M" ...
```

Remarque : il est important de souligner que la fonction *data.frame()* a automatiquement converti la variable **sexe** en **factor** alors qu'elle est issue d'un vecteur de **character**. Il s'agit d'une particularité de la classe **data.frame** où les variables qualitatives seront assimilées par défaut à des **factor**.

4.3 Concaténation

4.3.1 Ajout d'un vecteur-colonne

On peut ajouter un vecteur-colonne à un **data.frame** de plusieurs façons. Voici la première façon où on utilise le même procédé que pour créer un nouvel élément dans une liste :

```
don$diplome <- c("DU", "M2", "M2", "DU", "DU", "M2")
```

qui est équivalente à la commande suivante (comme on pourrait faire avec les matrices) :

```
don[,"diplome"] <- c("DU", "M2", "M2", "DU", "DU", "M2")
```

Remarque : pour que l'ajout d'un vecteur-colonne à un **data.frame** soit possible, il faut que la taille du vecteur-colonne soit égale au nombre de lignes du **data.frame**, sinon un message d'erreur apparaîtra. Pour vous en convaincre, essayer d'exécuter l'instruction suivante :

```
don[,"diplome"] <- c("DU", "M2", "M2", "DU", "DU", "M2", "M2")
```

On peut également utiliser la fonction *cbind()* qu'on a vu pour les matrices :

```
don <- cbind(don, pays = c("FR", "FR", "SNG", "CAM", "HAI", "BF"))
```

Remarque : on peut voir avec la commande *str()* que **diplome** est restée un **character**. Avec la fonction *cbind()*, **pays** a été converti en **factor**.

```
str(don)
```

```
## 'data.frame':    6 obs. of  5 variables:
##  $ age    : num  20 21 20 25 29 22
##  $ taille : num  165 155 150 170 175 180
##  $ sexe   : chr  "F" "F" "F" "M" ...
##  $ diplome: chr  "DU" "M2" "M2" "DU" ...
##  $ pays   : chr  "FR" "FR" "SNG" "CAM" ...
```

4.3.2 Ajout d'un vecteur-ligne

Pour ajouter un vecteur-ligne à un **data.frame**, on peut utiliser la fonction *rbind()*. Comme précédemment, il faut que la taille du vecteur-ligne soit égale au nombre de colonnes du **data.frame**.

```
don <- rbind(don, c(21, 180, "F", "DU", "FR"))
don
```

```
##         age taille sexe diplome pays
## sonia    20    165    F      DU   FR
## maud     21    155    F      M2   FR
## iris     20    150    F      M2  SNG
## mathieu  25    170    M      DU  CAM
## amin     29    175    M      DU  HAI
```

```
## gregory  22    180    M      M2   BF
## 7        21    180    F      DU   FR
```

Remarque : on constate qu'un nom de ligne a été attribué par défaut à l'observation. Il s'agit du numéro de ligne exprimé en **character**.

```
row.names(don)
```

```
## [1] "sonia"   "maud"    "iris"    "mathieu" "amin"    "gregory" "7"
```

4.3.3 Les fonctions *rbind()* et *cbind()*

Le concaténation de **data.frame** entre eux peut se faire via les les fonctions *cbind()* et *rbind()* que nous avons déjà utilisées avec les matrices. Celle-ci est possible sous certaines contraintes. Parmi ces contraintes :

- respect des dimensions (nombre de colonnes égales si concaténation d'individus, nombre de lignes égales si concaténation de variables),
- respect du nom des colonnes si on fait une concaténation d'individus,

Par exemple, pour ajouter de nouveaux individus :

```
don2 <- data.frame(age = c(20, 21), taille = c(180, 175), sexe = c("M", "F"),
                   diplome = c("DU", "DU"), pays = c("FR", "ESP"))
row.names(don2) <- c("pierre", "sonia")
don <- rbind(don, don2)
don
```

```
##         age taille sexe diplome pays
## sonia    20    165    F      DU   FR
## maud     21    155    F      M2   FR
## iris     20    150    F      M2  SNG
## mathieu  25    170    M      DU  CAM
## amin     29    175    M      DU  HAI
## gregory  22    180    M      M2   BF
## 7        21    180    F      DU   FR
## pierre   20    180    M      DU   FR
## sonia1   21    175    F      DU  ESP
```

Remarque 1 : si des observations portent le même nom dans les deux tables, alors **R** va automatiquement modifier le nom d'un des deux individus (il ajoute "1" après le nom) afin de respecter l'unicité de chaque observation.

Remarque 2 : si on concatène des individus, il y a une vérification moins stricte sur le type des colonnes. Par exemple, pour les **factor**, il peut y avoir des **levels** différents dans les deux tables, la nouvelle table va créer un **factor** contenant l'union des **levels**. Dans l'exemple ci-dessus, c'est ce qui s'est passé avec la modalité **ESP**.

4.3.4 La fonction *merge()*

Pour concaténer des **data.frame** entre eux, on préconisera l'utilisation de la fonction *merge()* à la fonction *cbind()* car la fonction *merge()* fait davantage de vérifications pour s'assurer qu'on va relier les bonnes observations entre elles. Par exemple, supposons qu'on ait un **data.frame** contenant deux colonnes, sachant que le nom des observations est contenu cette fois-ci dans la colonne **nom**. Par ailleurs, on a modifié l'ordre des noms par rapport à la table précédente :

```
don3 <- data.frame(note_algebre = c(18, 10, 8, 15, 20, 5, 17, 12, 8),
                   nom = c("sonia1", "pierre", "7", "gregory", "amin",
                           "mathieu", "iris", "maud", "sonia"))
```

Si on utilise la commande

```
cbind(don, don3)
```

```
##         age taille sexe diplome pays note_algebre     nom
## sonia    20    165    F      DU   FR           18  sonia1
## maud     21    155    F      M2   FR           10  pierre
## iris     20    150    F      M2  SNG            8       7
## mathieu  25    170    M      DU  CAM           15 gregory
## amin     29    175    M      DU  HAI           20    amin
## gregory  22    180    M      M2   BF            5 mathieu
## 7        21    180    F      DU   FR           17    iris
## pierre   20    180    M      DU   FR           12    maud
## sonia1   21    175    F      DU  ESP            8   sonia
```

cela aura pour effet de concaténer les lignes sans se soucier de l'identifiant,ce qui va créer le désordre dans la nouvelle table.

La fonction *merge()* permet d'éviter ce problème. Pour ce faire, chacune des table doit avoir une clé commune, autrement dit un identifiant unique qui soit le même dans les deux tables. Cet identifiant peut se trouver dans les attributs **row.names** ou alors dans une colonne de la table, mais il faudra le préciser dans les arguments de la fonction. Les deux premiers arguments donnent le nom des tables à concaténer, les deux derniers précisent où se situe la clé commune dans chaque table :

```
merge(don, don3, by.x = "row.names", by.y = "nom")
```

```
##   Row.names age taille sexe diplome pays note_algebre
## 1         7  21    180    F      DU   FR            8
## 2      amin  29    175    M      DU  HAI           20
## 3   gregory  22    180    M      M2   BF           15
## 4      iris  20    150    F      M2  SNG           17
## 5   mathieu  25    170    M      DU  CAM            5
## 6      maud  21    155    F      M2   FR           12
## 7    pierre  20    180    M      DU   FR           10
## 8     sonia  20    165    F      DU   FR            8
## 9    sonia1  21    175    F      DU  ESP           18
```

Remarque : la fonction *merge()* ne garde par défaut que les observations qui sont présentes simultanément dans les deux tables. On peut si on le souhaite étendre la sélection aux autres observations en jouant avec les options **all.x=**, **all.y=** et **all=**.

4.4 Extraction

Un **data.frame** est une table à deux dimensions. Il est donc logique qu'on puisse faire de la sélection sur les deux dimensions à la fois, exactement comme on l'a fait avec les matrices, en utilisant les crochets. On fera la sélection d'abord sur les individus (avant la virgule), puis ensuite sur les colonnes (après la virgule). Si on ne précise rien avant la virgule, cela signifie qu'on sélectionne toutes les observations. Si on ne précise rien après la virgule, cela signifie qu'on sélectionne toutes les colonnes.

4.4.1 Extraction soit par le nom, soit par le numéro des indices

4.4.1.1 Sur les lignes Si on souhaite extraire des lignes d'un **data.frame**, on a plusieurs options. On peut appeler le nom de la ligne qui nous intéresse :

```
don["sonia",]
```

```
##       age taille sexe diplome pays
## sonia  20    165    F      DU   FR
```

ou bien appeler l'indice ou le vecteur des indices qui nous intéresse :

```
don[c(1, 3, 5),]
```

```
##       age taille sexe diplome pays
## sonia  20    165    F      DU   FR
## iris   20    150    F      M2  SNG
## amin   29    175    M      DU  HAI
```

4.4.1.2 Sur les colonnes De même, pour sélectionner une ou plusieurs colonnes, on le fait soit en utilisant le nom des variables, soit leurs indices dans l'espace des colonnes.

```
don[, c(2, 3)]
```

```
##         taille sexe
## sonia      165    F
## maud       155    F
## iris       150    F
## mathieu    170    M
## amin       175    M
## gregory    180    M
## 7          180    F
## pierre     180    M
## sonia1     175    F
```

est équivalent à :

```
don[, c("age", "sexe")]
```

```
##         age sexe
## sonia    20    F
## maud     21    F
## iris     20    F
## mathieu  25    M
## amin     29    M
## gregory  22    M
## 7        21    F
## pierre   20    M
## sonia1   21    F
```

4.4.1.3 Sur les lignes et les colonnes à la fois Enfin, on peut sélectionner sur certaines observations, certaines variables :

```
don[c("sonia", "sonia1"), c(3, 5)]
```

```
##        sexe pays
## sonia     F   FR
## sonia1    F  ESP
```

4.4.2 Extraction sur requête

Supposons qu'on souhaite sélectionner uniquement les filles. Pour cela, il y a deux façons pratiquement équivalentes de le faire. La première est de donner un vecteur de booléen (de même taille que le nombre d'observations) dans la première partie des crochets, comme le montre l'exemple ci-dessous :

```
don$sexe == "F"
```

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

```
don[don$sexe == "F",]
```

```
##        age taille sexe diplome pays
## sonia   20    165    F      DU   FR
## maud    21    155    F      M2   FR
## iris    20    150    F      M2  SNG
## 7       21    180    F      DU   FR
## sonia1  21    175    F      DU  ESP
```

Le deuxième consiste à mettre les numéros des observations qui satisfont la requête dans la première partie des crochets. C'est exactement ce que fait la fonction *which()* appliquée sur un vecteur de booléen :

```
which(don$sexe == "F")
```

```
## [1] 1 2 3 7 9
```

```
don[which(don$sexe == "F"),]
```

```
##        age taille sexe diplome pays
## sonia   20    165    F      DU   FR
## maud    21    155    F      M2   FR
## iris    20    150    F      M2  SNG
## 7       21    180    F      DU   FR
## sonia1  21    175    F      DU  ESP
```

4.5 Fonctions de base

Tout comme pour les matrices, un certain nombre de fonctions permettent de connaitre certaines propriétés d'un **data frame**. Nous allons illustrer ces fonctions sur le **data.frame diamants** que nous avons chargé au tout début de ce chapitre.

Pour afficher les premières, respectivement dernières lignes, on utilise les fonctions *head()* et *tail()* avec comme deuxième argument le nombre de lignes à afficher :

```
head(diamants, 3)
```

```
##   carat     cut color clarity depth table price    x    y    z
## 1  0.23   Ideal     E     SI2  61.5    55   326 3.95 3.98 2.43
## 2  0.21 Premium     E     SI1  59.8    61   326 3.89 3.84 2.31
## 3  0.23    Good     E     VS1  56.9    65   327 4.05 4.07 2.31
```

```
tail(diamants, 2)
```

```
##       carat     cut color clarity depth table price    x    y    z
## 53939  0.86 Premium     H     SI2  61.0    58  2757 6.15 6.12 3.74
## 53940  0.75   Ideal     D     SI2  62.2    55  2757 5.83 5.87 3.64
```

Il est également possible d'obtenir des informations statistiques sur les variables contenues dans la structure de données avec la fonction *summary()* qui donne la distribution des variables quantitatives et la répartition par modalité des variables qualitatives. Ainsi :

```
summary(diamants)
```

```
##      carat               cut        color        clarity          depth
##  Min.   :0.2000   Fair     : 1610   D: 6775   SI1    :13065   Min.   :43.00
##  1st Qu.:0.4000   Good     : 4906   E: 9797   VS2    :12258   1st Qu.:61.00
##  Median :0.7000   Very Good:12082   F: 9542   SI2    : 9194   Median :61.80
##  Mean   :0.7979   Premium  :13791   G:11292   VS1    : 8171   Mean   :61.75
##  3rd Qu.:1.0400   Ideal    :21551   H: 8304   VVS2   : 5066   3rd Qu.:62.50
```

```
##  Max.   :5.0100                    I: 5422   VVS1   : 3655   Max.   :79.00
##                                     J: 2808   (Other): 2531
##      table           price             x                y
##  Min.   :43.00   Min.   :  326   Min.   : 0.000   Min.   : 0.000
##  1st Qu.:56.00   1st Qu.:  950   1st Qu.: 4.710   1st Qu.: 4.720
##  Median :57.00   Median : 2401   Median : 5.700   Median : 5.710
##  Mean   :57.46   Mean   : 3933   Mean   : 5.731   Mean   : 5.735
##  3rd Qu.:59.00   3rd Qu.: 5324   3rd Qu.: 6.540   3rd Qu.: 6.540
##  Max.   :95.00   Max.   :18823   Max.   :10.740   Max.   :58.900
##
##        z
##  Min.   : 0.000
##  1st Qu.: 2.910
##  Median : 3.530
##  Mean   : 3.539
##  3rd Qu.: 4.040
##  Max.   :31.800
##
```

Exercice 1.17

- Créer un **data.frame** nommé **my__df** qui contient les deux variables **my__num** et **my__char** qu'on appelera **var__1** et **var__2**.
- Ajouter l'observation nommé "com_11" qui prend 20 pour **var__1** et "F" pour **var__2**
- Ajouter une colonne appelé **var__3** qui vaut 10, 10, 10, 14, 12, 12, 12, 11, 10, 18, 19