

# Structures de données en R

Sophie Baillargeon, Université Laval

2021-01-30

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Types d'objets R servant de structure de données . . . . .	3
1.1.1	Exemple de vecteur . . . . .	4
1.1.2	Exemple de matrice . . . . .	4
1.1.3	Exemple d'array . . . . .	5
1.1.4	Exemple de liste . . . . .	5
1.1.5	Exemple de data frame . . . . .	6
1.1.6	Exemple de facteur . . . . .	7
1.2	Types de données . . . . .	7
1.2.1	Données réelles . . . . .	7
1.2.2	Données entières . . . . .	8
1.2.3	Données caractères . . . . .	9
1.2.4	Données logiques . . . . .	9
1.2.5	Données manquantes . . . . .	9
1.3	Obtention d'informations sur les objets . . . . .	10
1.3.1	Attributs des objets . . . . .	10
1.4	Extraction d'éléments . . . . .	12
1.4.1	Opérateurs d'indigage . . . . .	12
1.4.2	Fonctions d'extraction . . . . .	16
1.5	Remplacement d'éléments . . . . .	17
<b>2</b>	<b>Vecteurs</b>	<b>18</b>
2.1	Obtention d'informations . . . . .	19
2.2	Création . . . . .	19
2.2.1	Fonction <code>c</code> . . . . .	19
2.2.2	Fonction <code>vector</code> . . . . .	19
2.2.3	Fonction <code>as.vector</code> . . . . .	20
2.2.4	Fonction <code>rep</code> . . . . .	20
2.2.5	Création de séquences avec <code>:</code> ou <code>seq</code> . . . . .	20
2.2.6	Vecteur de longueur 1 . . . . .	21
2.3	Concaténation de vecteurs . . . . .	21
2.3.1	Fonction <code>c</code> . . . . .	21
2.3.2	Fonction <code>append</code> . . . . .	21
2.4	Métadonnées . . . . .	22
2.5	Extraction d'éléments . . . . .	22
2.5.1	Un seul élément . . . . .	22
2.5.2	Plusieurs éléments . . . . .	23
2.5.3	Extraction par exclusion . . . . .	24
2.6	Vecteurs de chaînes de caractères . . . . .	24
2.6.1	Fonctions <code>paste</code> et <code>paste0</code> . . . . .	25

<b>3</b>	<b>Matrices</b>	<b>26</b>
3.1	Obtention d'informations	26
3.2	Création	27
3.2.1	Fonction <code>matrix</code>	27
3.2.2	Fonction <code>as.matrix</code>	27
3.2.3	Fonctions <code>rbind</code> et <code>cbind</code>	27
3.3	Concaténation de matrices	28
3.3.1	Fonctions <code>rbind</code> et <code>cbind</code>	28
3.4	Métadonnées	28
3.5	Extraction d'éléments	29
3.5.1	Un seul élément	29
3.5.2	Plusieurs éléments	29
<b>4</b>	<b>Arrays</b>	<b>30</b>
4.1	Obtention d'informations	30
4.2	Création	31
4.2.1	Fonction <code>array</code>	31
4.2.2	Fonction <code>as.array</code>	31
4.3	Métadonnées	31
4.4	Extraction d'éléments	32
<b>5</b>	<b>Listes</b>	<b>33</b>
5.1	Obtention d'informations	33
5.2	Création	33
5.2.1	Fonction <code>list</code>	33
5.2.2	Fonction <code>as.list</code>	34
5.2.3	Fonction <code>vector</code>	34
5.3	Concaténation de listes	35
5.3.1	Fonction <code>c</code>	35
5.4	Métadonnées	35
5.5	Extraction d'éléments	36
5.5.1	Un seul élément	36
5.5.2	Plusieurs éléments	37
5.5.3	Extraction par exclusion	37
5.6	Conversion en vecteur atomique	37
<b>6</b>	<b>Data frames</b>	<b>38</b>
6.1	Jeux de données	38
6.1.1	Définitions	38
6.1.2	Représentation en R	38
6.2	Obtention d'informations	39
6.3	Création	40
6.3.1	Fonction <code>data.frame</code>	40
6.3.2	Fonction <code>as.data.frame</code>	41
6.4	Concaténation de data frames	41
6.4.1	Fonction <code>data.frame</code>	41
6.4.2	Fonctions <code>rbind</code> et <code>cbind</code>	42
6.5	Métadonnées	42
6.6	Extraction d'éléments	43
6.6.1	En traitant le data frame comme une liste	43
6.6.2	En traitant le data frame comme une matrice	44
6.6.3	Extraction par exclusion	44
6.7	Extensions	45
6.7.1	Tibbles	45

6.7.2	Data tables	46
<b>7</b>	<b>Facteurs</b>	<b>47</b>
7.1	Obtention d'informations	47
7.2	Création	48
7.2.1	Fonctions <code>factor</code> et <code>as.factor</code>	48
7.3	Modification	48
7.3.1	Fonction <code>levels</code>	48
7.3.2	Réordonnement des niveaux	48
7.4	Métadonnées	49
7.5	Extraction d'éléments	49
7.6	Facteurs ordonnés	49
<b>8</b>	<b>Conversions de type de données</b>	<b>50</b>
8.1	Conversions explicites	51
<b>9</b>	<b>Résumé</b>	<b>52</b>
	<b>Références</b>	<b>56</b>

---

*Note préliminaire : Lors de leur dernière mise à jour, ces notes ont été révisées en utilisant R version 4.0.3, le package `tibble` version 3.0.5 et le package `data.table` version 1.13.6. Pour d'autres versions, les informations peuvent différer.*

---

## 1 Introduction

Le point de départ d'une analyse de données en R est d'avoir accès aux données. Ces données doivent être stockées dans un ou des objets R. Pour utiliser R, il faut donc d'abord connaître les différents types d'objets pouvant servir de « contenant à données » et savoir travailler avec ces objets.

### 1.1 Types d'objets R servant de structure de données

Le tableau suivant présente les différents types d'objets R servant de structure de données offerts dans le R de base, selon leur nombre de dimensions.

Nombre de dimensions	Objets atomiques	Objets rékursifs
1	<b>vecteur</b>	<b>liste</b>
2	<b>matrice</b>	<b>data frame</b>
⋮	⋮	
n	<b>array</b>	-

Aux 5 types d'objets de ce tableau, il faut ajouter les **facteurs**. Ils sont une généralisation des vecteurs, utiles pour stocker des données catégoriques.

D'autres types d'objets existent dans le R de base, mais ne servent pas de structure de données, par exemple les fonctions, les expressions (comme les formules), etc.

Les objets dont les éléments sont contraints d'être des données toutes du même type sont qualifiés de « **atomiques** », alors que les autres sont qualifiés de « **rékursifs** ». En fait, les listes sont parfois appelées « vecteurs rékursifs » dans la documentation de R. Ici, le mot vecteur, non accompagné de l'adjectif rékursif, fera toujours référence à un vecteur atomique.

Les éléments des objets récursifs sont d'autres objets. Pour une liste, ils peuvent être n'importe quoi : des vecteurs, des facteurs, des matrices, des arrays, des listes, des data frames, des fonctions, des expressions, etc. Pour un data frame, ils sont typiquement des vecteurs ou des facteurs, tous de même longueur.

Illustrons maintenant ces différents types d'objets servant de structure de données. De l'information détaillée sur chaque type d'objet est présentée plus loin.

### 1.1.1 Exemple de vecteur

Le vecteur est l'objet le plus simple. Voici un exemple de vecteur :

```
vec
```

```
## [1] 3.5 7.8 9.9 5.7
```

```
str(vec)
```

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



Il pourrait être représenté comme suit :  ou encore à la verticale : 

Il n'a qu'une dimension et ses éléments (représentés par les cercles bleus) sont des données toutes du même type.

### 1.1.2 Exemple de matrice

La matrice est une généralisation à 2 dimensions du vecteur. Voici un exemple de matrice :

```
mat
```

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

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

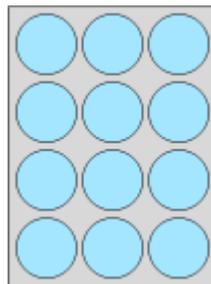
```
## [2,] TRUE  TRUE FALSE
```

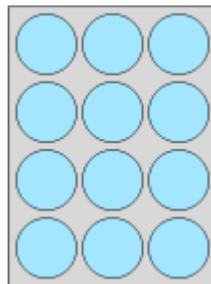
```
## [3,] FALSE TRUE  TRUE
```

```
## [4,] TRUE  TRUE  TRUE
```

```
str(mat)
```

```
## logi [1:4, 1:3] TRUE TRUE FALSE TRUE FALSE TRUE ...
```



Elle pourrait être représentée comme suit : 

Comme pour un vecteur, tous ses éléments doivent être du même type.

### 1.1.3 Exemple d'array

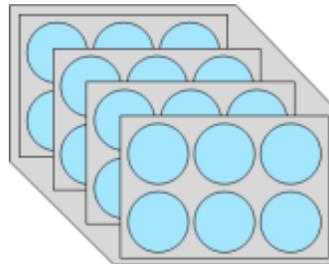
Le dernier objet atomique est l'array, qui généralise le vecteur et la matrice à un nombre quelconque de dimensions. Voici un exemple d'array à 3 dimensions :

```
arr
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    2    3    6
## [2,]    4    1    3
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    9    6    8
## [2,]    2    3    0
##
## , , 3
##
##      [,1] [,2] [,3]
## [1,]    5    2    7
## [2,]    3    8    1
##
## , , 4
##
##      [,1] [,2] [,3]
## [1,]    2    5    8
## [2,]    6    3    5
```

```
str(arr)
```

```
## num [1:2, 1:3, 1:4] 2 4 3 1 6 3 9 2 6 3 ...
```



Il pourrait être représenté comme suit :

### 1.1.4 Exemple de liste

Une liste est pour sa part un objet récursif : elle contient d'autres objets, de types quelconques. Voici un exemple de liste :

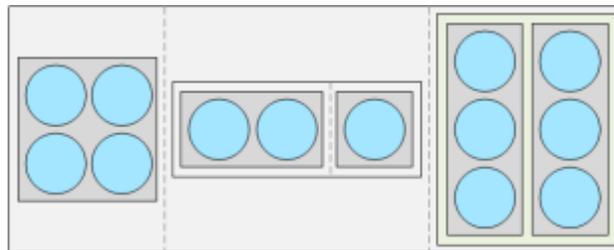
```
liste
```

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

```
## [1] 2 3
##
## [[2]][[2]]
## [1] 8
##
##
## [[3]]
##   V1 V2
## 1  2  1
## 2  4  5
## 3  3  3
```

```
str(liste)
```

```
## List of 3
## $ : num [1:2, 1:2] 2 4 3 1
## $ :List of 2
## ..$ : num [1:2] 2 3
## ..$ : num 8
## $ :'data.frame':   3 obs. of  2 variables:
## ..$ V1: num [1:3] 2 4 3
## ..$ V2: num [1:3] 1 5 3
```



Elle pourrait être représentée comme suit :

Les éléments d'une liste (séparés par une ligne pointillée dans la figure) ne sont pas de simples données comme dans un vecteur. Il s'agit d'autres objets. Dans cet exemple, la liste contient même une autre liste (deuxième élément).

### 1.1.5 Exemple de data frame

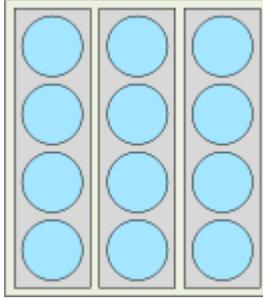
Le data frame est en quelque sorte un type particulier de liste, mais il partage aussi des caractéristiques avec les matrices, étant réputé posséder 2 dimensions (des lignes et des colonnes). Ses éléments sont des objets contraints d'être de dimensions concordantes. Dans la grande majorité des cas, les éléments sont des vecteurs ou des facteurs de même longueur, formant les colonnes du data frame. Voici un exemple de data frame.

```
dat
```

```
##   V1      V2  V3
## 1 3.5   matin TRUE
## 2 7.8 après-midi TRUE
## 3 9.9    soir FALSE
## 4 5.7    nuit  TRUE
```

```
str(dat)
```

```
## 'data.frame':   4 obs. of  3 variables:
## $ V1: num  3.5 7.8 9.9 5.7
## $ V2: chr  "matin" "après-midi" "soir" "nuit"
## $ V3: logi  TRUE TRUE FALSE TRUE
```



Il pourrait être illustré comme suit :

La taille de la première dimension d'un data frame typique est la longueur commune des objets qu'il contient (4 dans l'exemple) et la taille de sa deuxième dimension est le nombre d'objets qu'il contient (dans l'exemple, le data frame contient 3 vecteurs).

Les data frames ont été créés spécialement pour l'analyse statistique de données. Ils représentent des tableaux de données dans lesquels les lignes sont des observations et les colonnes des variables (des variables au sens statistique et non au sens informatique). Le data frame fait donc penser à une matrice. Cependant, contrairement à une matrice, les données dans les différentes colonnes d'un data frame n'ont pas à être du même type.

### 1.1.6 Exemple de facteur

Un autre type d'objet propre à la statistique a été créé en R : le facteur. Ce type d'objet est utilisé pour stocker les valeurs observées d'une variable catégorique.

```
fac
```

```
## [1] 5 2 5 5  
## Levels: 2 5
```

```
str(fac)
```

```
## Factor w/ 2 levels "2","5": 2 1 2 2
```

Il s'agit d'une généralisation du vecteur.

## 1.2 Types de données

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

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

Voici un exemple de chacun des types.

### 1.2.1 Données réelles

Une donnée réelle est un nombre réel, par exemple

```
re <- 5.8  
re
```

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

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

```
typeof(re)
```

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

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

```
str(re)
```

```
## num 5.8
```

### 1.2.2 Données entières

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

```
en <- 1L  
en
```

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

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

```
str(en)
```

```
## int 1
```

```
typeof(en)
```

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

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

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

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

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

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

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

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

```
un <- 1  
un
```

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

```
str(un)
```

```
## num 1
```

```
typeof(un)
```

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

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

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

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

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

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

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

### 1.2.3 Données caractères

Une donnée caractère est une chaîne de caractères.

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

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

```
str(ca)
```

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

```
typeof(ca)
```

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

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

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

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

### 1.2.4 Données logiques

Une donnée logique est simplement TRUE ou FALSE.

```
lo <- TRUE  
lo
```

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

```
str(lo)
```

```
## logi TRUE
```

```
typeof(lo)
```

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

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

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

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

### 1.2.5 Données manquantes

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

```
NA
```

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

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

```
0/0
```

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

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

### 1.3 Obtention d'informations sur les objets

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

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

Ces fonctions seront illustrées plus loin.

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

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

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

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

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

#### 1.3.1 Attributs des objets

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

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

```
attributes(mat)
```

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

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

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

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

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

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

```
dim(mat)
```

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

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

```
nrow(mat)
```

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

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

```
ncol(mat)
```

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

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

```
attributes(vec)
```

```
## NULL
```

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

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

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

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

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

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

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

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

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

Légende :

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

## 1.4 Extraction d'éléments

L'extraction d'éléments dans un jeu de données est une opération très usuelle au cours d'une analyse de données. Par exemple, il arrive souvent de vouloir faire un certain calcul seulement sur une partie des données plutôt que sur le jeu de données entier. Il faut alors prélever les données concernées.

### 1.4.1 Opérateurs d'indigage

Il y a trois opérateurs d'indigage d'éléments d'un objet `R` : `[`, `[[` et `$`. Ces opérateurs permettent d'extraire des éléments s'ils sont utilisés seuls, et ils permettent de remplacer des éléments s'ils sont utilisés en combinaison avec une assignation.

#### 1.4.1.1 `[` versus `[[` et `$`

L'opérateur `[` effectue de l'extraction **d'éléments** (potentiellement plus d'un) en **préservant presque toujours la structure** de l'objet (l'objet produit en résultat de l'extraction est, sauf exception, du même type que l'objet d'origine et conserve la majorité de ses attributs).

Les opérateurs `[[` et `$` permettent de référer à **un seul élément** et ils **simplifient l'objet** lors d'une extraction (retirent des attributs pour les objets atomiques et sortent de l'objet principal pour un objet récursif).

Voici des exemples.

- extraction de deux éléments d'un objet atomique avec `[`

```
vec
## [1] 3.5 7.8 9.9 5.7
vec[c(2, 4)]
## [1] 7.8 5.7
```

Dans cet exemple, la fonction `c` sert à créer un vecteur contenant les positions des éléments à extraire.

- extraction d'un élément d'un objet récursif avec `[`, en préservant la structure de l'objet d'origine

```
str(liste)

## List of 3
## $ : num [1:2, 1:2] 2 4 3 1
## $ :List of 2
## ..$ : num [1:2] 2 3
## ..$ : num 8
## $ :'data.frame': 3 obs. of 2 variables:
## ..$ V1: num [1:3] 2 4 3
## ..$ V2: num [1:3] 1 5 3

extrait_1_liste <- liste[1]
str(extrait_1_liste)

## List of 1
## $ : num [1:2, 1:2] 2 4 3 1
```

Ici, le résultat est encore une liste, mais contenant un seul élément.

- extraction d'un élément d'un objet récursif avec `[[`, avec perte de la structure de l'objet d'origine

```
extrait_2_liste <- liste[[1]]
str(extrait_2_liste)
```

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

Ici, le résultat est un vecteur, celui qui était stocké en position 1 dans l'objet `liste`.

- extraction d'un élément d'un objet atomique avec perte d'un attribut

```
mat
```

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

```
attributes(mat)
```

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

```
extrait_mat <- mat[[1, 3]]
extrait_mat
```

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

```
attributes(extrait_mat)
```

```
## NULL
```

#### 1.4.1.2 Extractions d'éléments dans un objet à plusieurs dimensions

Les opérateurs `[` et `[[` prennent en entrée un argument pour chacune des dimensions d'un objet, comme illustré dans l'exemple précédent. Voici un autre exemple, avec l'array à 3 dimensions `arr`. La commande suivante extrait un seul élément de `arr`, dont la position est identifiée par les valeurs fournies en entrée aux arguments de `[`.

```
arr[2, 1, 3]
```

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

#### 1.4.1.3 Spécificité de l'opérateur `$`

L'opérateur `$` fonctionne seulement avec les objets récursifs (une liste ou un data frame) dont les éléments sont nommés. Il doit être suivi du nom de l'élément à extraire. Ce nom n'a pas besoin d'être encadré de guillemets s'il ne contient pas d'espaces.

Prenons par exemple le data frame `dat` observé précédemment. Nous ne l'avons peut-être pas remarqué, mais les éléments de `dat` sont nommés.

```
str(dat)
```

```
## 'data.frame': 4 obs. of 3 variables:
## $ V1: num 3.5 7.8 9.9 5.7
## $ V2: chr "matin" "après-midi" "soir" "nuit"
## $ V3: logi TRUE TRUE FALSE TRUE
```

```
attr(dat, "names")
```

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

Alors il est possible d'extraire disons la deuxième colonne du data frame par la commande suivante.

```
dat$V2
```

```
## [1] "matin" "après-midi" "soir" "nuit"
```

#### 1.4.1.4 Identification de l'élément à extraire avec l'opérateur `[[`

Les arguments fournis à `[[` pour identifier l'unique élément à extraire peuvent recevoir des valeurs de type :

- numérique : un nombre spécifiant la position de l'élément selon la dimension, ou
- caractère (dans le cas d'éléments nommés) : le nom de l'élément selon la dimension.

Par exemple, le 2<sup>ème</sup> élément de la colonne nommée "V2" du data frame `dat` peut être extrait ainsi.

```
dat[[2, "V2"]]
```

```
## [1] "après-midi"
```

Si une valeur numérique non entière est fournie en argument, elle est tronquée vers 0, comme dans cet exemple.

```
vec
```

```
## [1] 3.5 7.8 9.9 5.7
```

```
vec[[2.6]]
```

```
## [1] 7.8
```

#### 1.4.1.5 Identification du ou des éléments à extraire avec l'opérateur `[`

Étant donné que l'opérateur `[` permet l'extraction de plusieurs éléments, il accepte une plus grande variété de types de valeurs. Voici les valeurs qu'il accepte en argument pour identifier les éléments à extraire :

- un ou des nombres entiers positifs, dont les valeurs sont entre 1 et la taille de la dimension concernée : *positions des éléments à sélectionner* ;
- un ou des nombres entiers négatifs, dont les valeurs sont entre 1 et la taille de la dimension concernée :  $-1 \times$  *positions des éléments à exclure* ;
- une ou des chaînes de caractères (possible seulement si les éléments sont nommés) : *noms des éléments à sélectionner* ;
- vecteur de logiques (doit être de la même longueur que la dimension concernée) : *TRUE pour les éléments à sélectionner, FALSE pour ceux à exclure* ;
- rien : utile par exemple pour extraire tous les éléments selon une dimension d'un objet à plusieurs dimensions.

Voici quelques exemples :

- extraction d'éléments identifiés par leur position,

```
vec[c(1, 3)]
```

```
## [1] 3.5 9.9
```

- extraction d'éléments par identification d'éléments à exclure,

```
vec[-c(2, 4)]
```

```
## [1] 3.5 9.9
```

- extraction d'éléments identifiés par leurs noms,

```
dat[c("V1", "V2")]
```

```
##      V1      V2
## 1 3.5    matin
## 2 7.8 après-midi
## 3 9.9     soir
## 4 5.7     nuit
```

- extraction d'éléments identifiés par un vecteur logique,

```
vec[c(TRUE, FALSE, TRUE, FALSE)]
```

```
## [1] 3.5 9.9
```

- extraction de tous les éléments selon une des dimensions.

```
dat[2, ]
```

```
##      V1      V2  V3
## 2 7.8 après-midi TRUE
```

Lors de l'indigage à l'aide de valeurs numériques positives ou de chaînes de caractères, les valeurs peuvent se répéter. Dans ce cas, les éléments sont extraits autant de fois que la fréquence de leur identifiant, comme dans ces exemples.

```
vec[c(1, 1, 1, 2, 2, 3, 3, 3, 3)]
```

```
## [1] 3.5 3.5 3.5 7.8 7.8 9.9 9.9 9.9 9.9
```

```
dat[, c("V2", "V2")]
```

```
##      V2      V2.1
## 1    matin    matin
## 2 après-midi après-midi
## 3     soir     soir
## 4     nuit     nuit
```

Les sections suivantes proposent plusieurs autres exemples d'extraction de données.

#### 1.4.1.6 Argument drop de l'opérateur [

L'opérateur [ préserve la structure des objets, sauf dans quelques cas particuliers. Un de ces cas est l'extraction d'une colonne, complète ou partielle, d'un data frame. Le résultat de l'extraction est alors un vecteur ou un facteur, et non un autre data frame, comme dans cet exemple.

```
str(dat)
```

```
## 'data.frame':  4 obs. of  3 variables:
## $ V1: num  3.5 7.8 9.9 5.7
## $ V2: chr  "matin" "après-midi" "soir" "nuit"
## $ V3: logi  TRUE TRUE FALSE TRUE
```

```
str(dat[c(1, 3), 2])
```

```
## chr [1:2] "matin" "soir"
```

Il n'a pas ce comportement lors de l'extraction d'une ligne d'un data frame, car les éléments sur une même ligne ne sont pas nécessairement du même type.

```
str(dat[2, ])
```

```
## 'data.frame':  1 obs. of  3 variables:
## $ V1: num 7.8
## $ V2: chr "après-midi"
## $ V3: logi TRUE
```

Cependant, avec les objets atomiques, l'opérateur [ cherche toujours par défaut à réduire le plus possible le nombre de dimensions de l'objet retourné. Ainsi, l'extraction suivante retourne un vecteur, alors que l'objet d'origine est un array à trois dimensions.

```
str(arr[, 2, 4])
```

```
## num [1:2] 5 3
```

Il est possible de contrôler ce comportement grâce à l'argument `drop`. Donner la valeur `FALSE` à cet argument empêche [ de réduire le nombre de dimensions du résultat retourné. Ainsi, la commande suivante retourne un array,

```
str(arr[, 2, 4, drop = FALSE])
```

```
## num [1:2, 1, 1] 5 3
```

et la suivante un data frame.

```
str(dat[c(1, 3), 2, drop = FALSE])
```

```
## 'data.frame':  2 obs. of  1 variable:
## $ V2: chr "matin" "soir"
```

## 1.4.2 Fonctions d'extraction

En plus des opérateurs d'indigage, certaines fonctions permettent aussi d'extraire des éléments, notamment les fonctions `head`, `tail` et `subset`.

### 1.4.2.1 Fonction head

La fonction `head` extrait les premiers éléments d'un vecteur, d'une liste ou d'un facteur et les premières lignes d'une matrice ou d'un data frame. L'argument `n` permet de spécifier combien d'éléments ou de lignes extraire. Voici un exemple.

```
head(vec, n = 2)
```

```
## [1] 3.5 7.8
```

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

```
##   V1      V2  V3
## 1 3.5   matin TRUE
## 2 7.8 après-midi TRUE
## 3 9.9     soir FALSE
```

### 1.4.2.2 Fonction tail

À l'inverse, la fonction `tail` extrait les derniers éléments ou les dernières lignes.

```
tail(vec, n = 3)
```

```
## [1] 7.8 9.9 5.7
```

```
tail(dat, n = 1)
```

```
##      V1  V2  V3  
## 4 5.7 nuit TRUE
```

### 1.4.2.3 Fonction subset

La fonction `subset` est quant à elle une option de rechange à l'opérateur `[` utilisé avec des identifiants logiques. Elle fonctionne avec les vecteurs.

```
subset(vec, subset = c(FALSE, TRUE, TRUE, TRUE))
```

```
## [1] 7.8 9.9 5.7
```

Cependant, elle est surtout utile avec des matrices et des data frames. Avec un objet à deux dimensions, l'argument `subset` de la fonction du même nom sert à identifier les lignes à extraire et l'argument `select` à identifier les colonnes.

```
subset(mat, subset = c(FALSE, TRUE, FALSE, TRUE), select = 1)
```

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

Lorsque le premier argument fourni en entrée à la fonction `subset` est un data frame, il est possible de référer aux colonnes de celui-ci directement par leurs noms dans les arguments `subset` et `select`. Il est aussi possible de retirer une colonne en utilisant l'opérateur `-` avant le nom de la colonne à retirer dans la valeur fournie à l'argument `select`.

```
subset(dat, subset = c(TRUE, FALSE, FALSE, TRUE), select = -V3)
```

```
##      V1  V2  
## 1 3.5 matin  
## 4 5.7  nuit
```

**Note :** Typiquement, le vecteur de `TRUE` et `FALSE` servant à identifier les lignes à sélectionner est créé par une condition logique, comme dans l'exemple suivant.

```
subset(dat, subset = V1 < 6, select = -V3)
```

```
##      V1  V2  
## 1 3.5 matin  
## 4 5.7  nuit
```

Nous verrons plus tard comment utiliser des opérateurs logiques pour créer des vecteurs d'identifiants logiques. Avec ces opérateurs, il sera facile d'extraire par exemple toutes les valeurs tombant dans un certain ensemble de valeurs acceptées.

## 1.5 Remplacement d'éléments

En combinant une extraction à un opérateur d'assignation (`<-`) suivi de nouvelles valeurs, des éléments d'un objet peuvent être remplacés. Voici un exemple.

```
vec
```

```
## [1] 3.5 7.8 9.9 5.7
```

```
vec[3] <- 6.1
```

```
vec
```

```
## [1] 3.5 7.8 6.1 5.7
```

Il est tout à fait possible de remplacer plus d'un élément à la fois, comme dans l'exemple suivant.

```
dat
##   V1      V2   V3
## 1 3.5   matin TRUE
## 2 7.8 après-midi TRUE
## 3 9.9    soir FALSE
## 4 5.7    nuit  TRUE
dat[c(1, 2), 3] <- c(NA, FALSE)
dat
```

```
##   V1      V2   V3
## 1 3.5   matin  NA
## 2 7.8 après-midi FALSE
## 3 9.9    soir FALSE
## 4 5.7    nuit  TRUE
```

Il est parfois utile d'exploiter la **règle de recyclage** lors du remplacement d'éléments dans un objet R. Cette règle permet des opérations sur des vecteurs qui ne sont pas de même longueur. Les éléments du vecteur le plus court sont répétés de façon à ce que ce vecteur devienne de la même longueur que le vecteur le plus long.

Par exemple, si une seule valeur de remplacement est fournie, mais que plusieurs éléments sont identifiés, alors ceux-ci seront tous remplacés par l'unique valeur fournie.

```
vec[1:3] <- 0.5
vec
```

```
## [1] 0.5 0.5 0.5 5.7
```

Cela fonctionne aussi lorsque le vecteur le plus court est de longueur supérieure à 1. Dans l'exemple suivant, 4 valeurs sont identifiées à remplacer, mais 2 valeurs de remplacement sont fournies. Le vecteur de valeurs de remplacement est donc dupliqué.

```
vec[1:4] <- c(0.1, 0.2)
vec
```

```
## [1] 0.1 0.2 0.1 0.2
```

Si la longueur du vecteur le plus long n'est pas un multiple de la longueur du vecteur le plus court, ça fonctionne encore, mais un avertissement est émis, comme dans cet exemple.

```
vec[1:3] <- c(2.0, 4.9)
```

```
## Warning in vec[1:3] <- c(2, 4.9): number of items to replace is not a multiple of
## replacement length
vec
```

```
## [1] 2.0 4.9 2.0 0.2
```

Mentionnons que la fonction `edit` permet également de modifier des éléments dans une matrice ou un data frame à l'intérieur d'un chiffrier ouvert dans une fenêtre externe. Aussi, la fonction `replace` permet de remplacer des éléments dans un vecteur. Ces fonctions ne sont pas illustrées ici.

---

## 2 Vecteurs

Un vecteur est un simple objet atomique à une dimension. Toutes les données qu'il contient doivent être du même type.

## 2.1 Obtention d'informations

Observons le vecteur `vec` illustré précédemment en utilisant des fonctions déjà vues jusqu'à maintenant.

Il est possible d'afficher les données que `vec` contient,

```
vec
```

```
## [1] 2.0 4.9 2.0 0.2
```

ou encore un résumé de ces données avec la fonction `summary`.

```
summary(vec)
```

```
##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.200  1.550   2.000   2.275   2.725   4.900
```

Des informations sur la structure de l'objet sont affichées avec la fonction `str`.

```
str(vec)
```

```
##  num [1:4] 2 4.9 2 0.2
```

L'objet possède une certaine longueur, retournée par la fonction `length`.

```
length(vec)
```

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

Son contenu est d'un certain type, retourné par la fonction `typeof`.

```
typeof(vec)
```

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

## 2.2 Création

### 2.2.1 Fonction `c`

La fonction `c` permet de créer un objet de type vecteur comme suit.

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

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

La lettre `c` a été choisie comme nom pour cette fonction, car celle-ci sert à *combiner* (en anglais *combine*) ou concaténer des éléments dans un vecteur.

Bien que les nombres dans ce vecteur soient des entiers, ils ont été stockés sous le format réel. C'est ce que fait par défaut la fonction `c`.

```
typeof(de)
```

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

### 2.2.2 Fonction `vector`

Une autre fonction permet de créer des vecteurs : la fonction `vector`. Elle initialise des vecteurs, qui sont remplis par la suite. Avec `vector`, il faut identifier la longueur du vecteur créé et le type des données qu'il doit contenir, comme dans cet exemple.

```
ve <- vector(mode = "numeric", length = 3)
ve
```

```
## [1] 0 0 0
```

L'utilisateur ne contrôle pas les données initiales placées dans un vecteur créé avec `vector`. Il doit plutôt les modifier a posteriori.

```
ve[1] <- 5
ve
```

```
## [1] 5 0 0
```

Cette fonction sera utile pour initialiser des vecteurs servant à contenir des valeurs créées itérativement dans une boucle.

### 2.2.3 Fonction `as.vector`

La fonction `as.vector` sert quant à elle à transformer un objet en vecteur. Tous les attributs de l'objet sont retirés lors de l'opération. Voici un exemple.

```
as.vector(mat)
```

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

### 2.2.4 Fonction `rep`

Si un vecteur doit contenir des données qui se répètent, la fonction `rep` est utile. Ainsi, plutôt que de créer, par exemple, un vecteur contenant cinq 1 suivis de cinq 2 avec la fonction `c` comme suit

```
c(1, 1, 1, 1, 1, 2, 2, 2, 2, 2)
```

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

il est plus succinct de le créer en utilisant conjointement les fonctions `rep` et `c` comme suit

```
rep(c(1, 2), each = 5)
```

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

L'utilisation de l'argument `each` de `rep` produit des répétitions consécutives élément par élément. Si la fonction `rep` est appelée sans attribuer explicitement les valeurs fournies en entrée à des arguments, alors la deuxième valeur fournie sera assignée à l'argument `times` et non `each`. Dans ce cas, un unique entier positif fourni en deuxième position provoque la répétition du vecteur entier fourni en première position un certain nombre de fois, comme dans cet exemple.

```
rep(c(1, 2), 5)
```

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

Le nombre de répétitions pourrait aussi varier d'un élément à l'autre en fournissant à l'argument `times` un vecteur de nombre de répétitions aussi long que le vecteur fourni comme premier argument.

```
rep(c(1, 2), times = c(4, 3))
```

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

Il est même possible d'exploiter simultanément les arguments `each` et `times`.

```
rep(c(1, 2), each = 3, times = 2)
```

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

### 2.2.5 Création de séquences avec `:` ou `seq`

Pour créer un vecteur contenant une séquence régulière de nombres, l'opérateur `:` ou la fonction `seq` sont très utiles.

Avec l'opérateur `:`, les séquences créées comportent des nombres séparés par des bonds de 1, par exemple

```
6:-2
```

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

```
0.2:4.2
```

```
## [1] 0.2 1.2 2.2 3.2 4.2
```

Le nombre placé devant l'opérateur indique le début de la séquence et le nombre placé après indique la fin.

La fonction `seq` généralise `:` en permettant des bonds de longueur autre que 1. Voici des exemples.

```
seq(from = 0, to = 9, by = 3)
```

```
## [1] 0 3 6 9
```

```
seq(from = 0, to = 9, length.out = 4)
```

```
## [1] 0 3 6 9
```

### 2.2.6 Vecteur de longueur 1

Si un objet R est créé en lui assignant une seule donnée, cet objet est considéré comme un vecteur de longueur 1.

```
une_donnee <- "a"  
str(une_donnee)
```

```
## chr "a"
```

```
is.vector(une_donnee)
```

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

```
length(une_donnee)
```

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

Ainsi, l'appel à la fonction `c` n'est pas nécessaire pour créer un vecteur contenant un seul élément.

## 2.3 Concaténation de vecteurs

### 2.3.1 Fonction `c`

En plus de servir à créer de nouveaux vecteurs, `c` permet de concaténer des vecteurs (autant que souhaité), comme dans cet exemple.

```
c(de, ve, 6:-2)
```

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

### 2.3.2 Fonction `append`

La fonction `append` fait aussi de la concaténation de vecteurs. Contrairement à `c`, elle est limitée à la fusion de deux vecteurs. Cependant, elle permet d'insérer le deuxième vecteur n'importe où dans le premier vecteur, pas nécessairement à la fin. Voici un exemple.

```
append(de, ve, after = 3)
```

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

## 2.4 Métadonnées

Supposons que l'objet `de` contienne les résultats de 10 lancers d'un dé : 2, 3, 4, 1, 2, 3, 5, 6, 5, 4. Dans cet objet, nous pourrions identifier à quel lancer chaque résultat fait référence en ajoutant des noms aux éléments du vecteur.

```
names(de) <- c("11", "12", "13", "14", "15", "16", "17", "18", "19", "110")
# nous apprendrons bientôt à créer plus succinctement ce vecteur avec l'instruction
# paste0("l", 1:10)
```

Voyons de quoi a l'air l'objet `de` suite à cet ajout.

```
de
##  11  12  13  14  15  16  17  18  19 110
##   2   3   4   1   2   3   5   6   5   4
str(de)
## Named num [1:10] 2 3 4 1 2 3 5 6 5 4
## - attr(*, "names")= chr [1:10] "11" "12" "13" "14" ...
```

Il possède maintenant un attribut nommé `"names"`.

```
attr(de, which = "names")
# ou
```

```
names(de)
## [1] "11" "12" "13" "14" "15" "16" "17" "18" "19" "110"
```

Cet attribut est un exemple de métadonnées. Il s'agit de données à propos des données.

Nous avons déjà vu que tout attribut peut être retiré en lui assignant la valeur spéciale `NULL`. Pour retirer les noms que nous venons d'ajouter, nous pourrions donc faire comme ceci.

```
names(de) <- NULL
de
## [1] 2 3 4 1 2 3 5 6 5 4
```

Pour la tâche spécifique d'effacer l'attribut `"names"` ou `"dimnames"`, une autre façon de procéder est d'appeler la fonction `unname` comme suit.

```
de <- unname(de)
```

Dans le cas d'un vecteur, il est également possible d'effacer d'un coup tous les attributs avec `as.vector` comme suit.

```
de <- as.vector(de)
```

Redonnons maintenant des noms aux éléments de `de`, car ils seront utiles dans les prochains exemples.

```
names(de) <- c("11", "12", "13", "14", "15", "16", "17", "18", "19", "110")
```

## 2.5 Extraction d'éléments

### 2.5.1 Un seul élément

Un élément d'un vecteur peut être extrait en l'identifiant par sa position comme suit.

```
de[1]
```

```
## 11
## 2
de[[1]]
```

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

Remarquons que l'opérateur `[` préserve le nom de l'élément (l'attribut `names`), alors que l'opérateur `[[` ne le conserve pas, ce qui concorde avec les propriétés de ces opérateurs vues précédemment.

Un élément peut aussi être extrait en l'identifiant par son nom.

```
de["11"]
```

```
## 11
## 2
```

Mais l'opérateur d'extraction `$` ne fonctionne pas avec un objet atomique tel qu'un vecteur.

```
de$11
```

```
## Error in de$11: $ operator is invalid for atomic vectors
```

## 2.5.2 Plusieurs éléments

Pour extraire plusieurs éléments simultanément, il faut utiliser un autre vecteur afin d'identifier les éléments à extraire. Voici quelques exemples selon le type du vecteur d'identifiants.

### 2.5.2.1 Identifiants numériques

Dans cet exemple, les éléments à extraire sont identifiés par des nombres représentant leurs positions.

```
posObs <- c(3, 6, 7)
de[posObs]
# ou directement
```

```
de[c(3, 6, 7)]
```

```
## 13 16 17
## 4 3 5
```

Note : L'opérateur `[[` sert à extraire uniquement un élément. Donc la commande suivante ne fonctionne pas.

```
de[[c(3, 6, 7)]]
```

```
## Error in de[[c(3, 6, 7)]]: attempt to select more than one element in vectorIndex
```

Exemples de moyens rapides de créer des vecteurs de positions avec l'opérateur `:` et la fonction `seq` :

- extraction des 5 premières observations (équivalent à `head(de, n = 5)`) :

```
de[1:5]
```

```
## 11 12 13 14 15
## 2 3 4 1 2
```

- extraction des observations en positions paires.

```
de[seq(from = 2, to = 10, by = 2)]
```

```
## 12 14 16 18 110
## 3 1 3 6 4
```

### 2.5.2.2 Identifiants caractères

Dans cet exemple, des chaînes de caractères contenant les noms des éléments à extraire sont utilisées.

```
de[c("13", "16", "17")]
```

```
## 13 16 17
## 4 3 5
```

### 2.5.2.3 Identifiants logiques

Voici un exemple de vecteur logique permettant d'identifier des éléments à extraire. Ce vecteur servira à extraire des éléments du vecteur `de`. Il doit donc être de longueur 10, tout comme `de`. Le vecteur contient la valeur `TRUE` aux positions des éléments à extraire et `FALSE` aux positions des éléments à exclure.

```
index.logic <- c(TRUE, rep(FALSE, 8), TRUE)
index.logic
```

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

Ce vecteur est utile pour extraire le premier et le dernier élément de `de`

```
de[index.logic]
```

```
## 11 110
## 2 4
```

### 2.5.3 Extraction par exclusion

Une autre méthode d'extraction d'éléments d'un vecteur consiste à identifier des éléments à exclure. Pour ce faire, il faut utiliser un indicateur numérique de position négatif. Voici un exemple qui permet d'extraire tous les éléments de `de` à l'exception des deux premiers.

```
de[-(1:2)]
```

```
## 13 14 15 16 17 18 19 110
## 4 1 2 3 5 6 5 4
```

Il est aussi possible de tronquer les derniers éléments en modifiant la longueur du vecteur.

```
length(de) <- 5
de
```

```
## 11 12 13 14 15
## 2 3 4 1 2
```

Si sa longueur initiale est réassignée au vecteur, que se passe-t-il ?

```
length(de) <- 10
de
```

```
## 11 12 13 14 15
## 2 3 4 1 2 NA NA NA NA NA
```

R ne sachant pas quelles données mettre dans les positions supplémentaires, il introduit sa constante pour les données manquantes `NA`.

## 2.6 Vecteurs de chaînes de caractères

Les vecteurs de chaînes de caractères sont un peu particuliers. Nous verrons dans un prochain cours des fonctions propres à la manipulation de vecteurs contenant ce type de données. Contentons-nous ici de regarder quelques exemples pour présenter ce genre de vecteurs.

```
letters
```

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

letters est une constante R qui contient l'alphabet en caractères minuscules (LETTERS est l'équivalent en majuscules). Il s'agit d'un vecteur de chaînes de caractères.

```
str(letters)
```

```
## chr [1:26] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" ...
```

Une autre constante R, month.name, contient les noms des mois de l'année en anglais. Utilisons cette constante pour illustrer la fonction nchar qui indique combien de caractères contiennent des chaînes de caractères.

```
month.name
```

```
## [1] "January" "February" "March" "April" "May" "June"  
## [7] "July" "August" "September" "October" "November" "December"
```

```
nchar(month.name)
```

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

Comme bien des fonctions R, la fonction nchar travaille de façon vectorielle. Si elle reçoit en entrée un vecteur, elle fait un calcul pour chaque élément du vecteur.

Pour savoir combien de chaînes de caractères sont contenues dans month.name, il faut utiliser la fonction length.

```
length(month.name)
```

```
## [1] 12
```

La fonction length ne retourne donc pas le longueur d'une chaîne de caractères, mais bien la longueur d'un objet (soit le nombre d'éléments dans cet objet).

```
length("January")
```

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

```
nchar("January")
```

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

### 2.6.1 Fonctions paste et paste0

Une fonction très utile pour créer des vecteurs de chaînes de caractères est la fonction paste. Comme son nom l'indique, elle permet de coller ensemble des chaînes de caractères. Par exemple, l'appel à la fonction paste suivant combine "Hello" et "world" dans une seule chaîne de caractères.

```
paste("Hello", "world", sep = " ")
```

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

L'argument sep = " " indique à paste de séparer les chaînes de caractères par un espace lors de la combinaison.

Il existe une fonction raccourcie pour l'utilisation de la fonction paste avec sep = "" (aucun caractère pour séparer les chaînes de caractères) : la fonction paste0.

```
paste0("Hello", "world")
```

```
## [1] "Helloworld"
```

En comparaison, la combinaison de ces deux chaînes de caractères avec la fonction `c` aurait créé un vecteur de longueur 2.

```
tentative <- c("Hello", "world")
str(tentative)
```

```
## chr [1:2] "Hello" "world"
```

Les fonctions `paste` et `paste0` travaillent, tout comme `nchar`, de façon vectorielle. Si elles reçoivent en entrée des vecteurs, elles les combinent élément par élément et retournent un vecteur de même longueur que ceux fournis.

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

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

Les fonctions `paste` et `paste0` permettent aussi de combiner dans une seule chaîne de caractères les chaînes de caractères comprises dans un vecteur grâce à leur argument `collapse`.

```
paste(LETTERS, collapse = ",")
```

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

---

## 3 Matrices

Une matrice est un objet atomique à deux dimensions. Elle possède des lignes et des colonnes. Toutes les données qu'elle contient sont du même type.

### 3.1 Obtention d'informations

Observons la matrice `mat` présentée précédemment.

```
mat
```

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

Quelles sont les informations relatives à cet objet ?

```
str(mat)
```

```
## logi [1:4, 1:3] TRUE TRUE FALSE TRUE FALSE TRUE ...
```

Il contient des éléments de type logique.

```
dim(mat)
```

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

Il possède deux dimensions.

```
nrow(mat)
```

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

La taille de sa première dimension est vue comme un nombre de lignes. Il y en a 4.

```
ncol(mat)
```

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

La taille de sa deuxième dimension est vue comme un nombre de colonnes. Il y en a 3.

```
length(mat)
```

```
## [1] 12
```

La longueur d'une matrice est son nombre total d'éléments, soit 12 dans cet exemple.

## 3.2 Création

### 3.2.1 Fonction `matrix`

La fonction `matrix` permet de créer un objet de type matrice à partir d'un vecteur contenant les éléments à mettre dans la matrice.

```
mat <- matrix(  
  c(TRUE, TRUE, FALSE, TRUE, FALSE, TRUE, TRUE, TRUE, FALSE, FALSE, TRUE, TRUE),  
  ncol = 3, nrow = 4, byrow = FALSE  
)
```

L'objet obtenu est celui que nous venons d'observer.

La fonction `matrix` attend comme premier argument un vecteur d'éléments, ensuite un nombre de colonnes (argument `ncol`) et/ou un nombre de lignes (argument `nrow`). L'argument `byrow` permet de spécifier si les éléments du vecteur fourni comme premier argument doivent être placés dans la matrice ligne par ligne ou colonne par colonne (ce qui est fait par défaut).

### 3.2.2 Fonction `as.matrix`

Pour tenter de convertir en matrice un objet d'un autre type, la fonction `as.matrix` est utile. Par exemple, convertissons en matrice la data frame `dat` créé précédemment.

```
as.matrix(dat)
```

```
##      V1    V2      V3  
## [1,] "3.5" "matin"   NA  
## [2,] "7.8" "après-midi" "FALSE"  
## [3,] "9.9" "soir"    "FALSE"  
## [4,] "5.7" "nuit"    "TRUE"
```

Étant donné qu'une matrice est un objet atomique, tous ses éléments doivent être des données du même type. Ici, toutes les données ont donc été converties en chaînes de caractères. Nous reviendrons plus loin sur les [conversions de types de données](#).

### 3.2.3 Fonctions `rbind` et `cbind`

Nous allons maintenant étendre notre exemple de données provenant de 10 lancers d'un dé à des données provenant de 10 lancers de 2 dés. Supposons donc qu'en plus des résultats du dé 1 (2, 3, 4, 1, 2, 3, 5, 6, 5, 4), nous avons les résultats du dé 2 : 1, 4, 2, 3, 5, 4, 6, 2, 5, 3.

Les fonctions `rbind` et `cbind` permettent de créer des matrices à partir d'une série de vecteurs de même longueur, comme dans cet exemple.

```
de_1 <- c(2, 3, 4, 1, 2, 3, 5, 6, 5, 4)  
de_2 <- c(1, 4, 2, 3, 5, 4, 6, 2, 5, 3)
```

```
des_lignes <- rbind(de_1, de_2)
des_lignes
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## de_1  2   3   4   1   2   3   5   6   5   4
## de_2  1   4   2   3   5   4   6   2   5   3
```

Le `r` dans le nom `rbind` indique que la fonction combine des vecteurs par *rows* en anglais ou lignes en français, dans le sens que chacun des vecteurs devient une ligne de la matrice produite.

Pour les données de lancers de dés, il serait plus approprié de combiner les vecteurs par colonnes, afin que les lignes de la matrice obtenue représentent des observations et les colonnes des variables. Nous utiliserons donc plutôt `cbind` (*c* pour *columns* en anglais ou colonnes en français)

```
des <- cbind(de_1, de_2)
des
```

```
##      de_1 de_2
## [1,]    2    1
## [2,]    3    4
## [3,]    4    2
## [4,]    1    3
## [5,]    2    5
## [6,]    3    4
## [7,]    5    6
## [8,]    6    2
## [9,]    5    5
## [10,]   4    3
```

### 3.3 Concaténation de matrices

#### 3.3.1 Fonctions `rbind` et `cbind`

Les fonctions `rbind` et `cbind` acceptent aussi en entrée des matrices, qu'elles concatènent, comme dans cet exemple.

```
rbind(des_lignes, des_lignes)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## de_1  2   3   4   1   2   3   5   6   5   4
## de_2  1   4   2   3   5   4   6   2   5   3
## de_1  2   3   4   1   2   3   5   6   5   4
## de_2  1   4   2   3   5   4   6   2   5   3
```

Des matrices concaténées en lignes doivent avoir le même nombre de colonnes et des matrices concaténées en colonnes doivent avoir le même nombre de lignes.

### 3.4 Métadonnées

Les lignes et colonnes d'une matrice peuvent être nommées (ou renommés).

```
rownames(des) <- paste0("l", 1:10)
colnames(des) <- paste0("de", 1:2)
str(des)
```

```
## num [1:10, 1:2] 2 3 4 1 2 3 5 6 5 4 ...
## - attr(*, "dimnames")=List of 2
## ..$ : chr [1:10] "l1" "l2" "l3" "l4" ...
```

```
## ..$ : chr [1:2] "de1" "de2"
```

Les noms des lignes et colonnes d'une matrice R sont des attributs de l'objet, tout comme les tailles de ses dimensions.

```
attributes(des)
```

```
## $dim
## [1] 10 2
##
## $dimnames
## $dimnames[[1]]
## [1] "11" "12" "13" "14" "15" "16" "17" "18" "19" "110"
##
## $dimnames[[2]]
## [1] "de1" "de2"
```

Les noms des lignes peuvent être extraits des plusieurs façons :

```
rownames(des)
```

```
#ou
```

```
dimnames(des)[[1]]
```

```
#ou
```

```
attr(des, which = "dimnames")[[1]]
```

```
## [1] "11" "12" "13" "14" "15" "16" "17" "18" "19" "110"
```

Les noms des colonnes peuvent être extraits de façons similaires avec la fonction `colnames` et l'élément 2 de l'attribut `dimnames`.

### 3.5 Extraction d'éléments

Tout ce qui a été mentionné concernant l'extraction de données dans un vecteur fonctionne aussi pour une matrice. Cependant, il y a maintenant deux dimensions à considérer.

#### 3.5.1 Un seul élément

Un élément est identifié par sa position en ligne et en colonne.

```
des[9, 2]
```

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

En réalité, cette extraction est équivalente à

```
des[19]
```

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

parce qu'en mémoire, une matrice est stockée comme un vecteur, en mettant bout à bout ses colonnes.

#### 3.5.2 Plusieurs éléments

Il est possible d'extraire des lignes complètes,

```
des[2:4, ]
```

```
##   de1 de2
## 12  3  4
## 13  4  2
## 14  1  3
```

ou encore des colonnes.

```
des[, -2]
```

```
## 11 12 13 14 15 16 17 18 19 110
##  2  3  4  1  2  3  5  6  5  4
```

Étant donné qu'il ne restait qu'une seule colonne, R a par défaut transformé la matrice en vecteur. Rappelons que pour empêcher cette simplification d'être effectuée, il faut utiliser l'argument `drop = FALSE`.

```
des[, -2, drop = FALSE]
```

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

Les lignes ou les colonnes peuvent aussi être identifiées par leur nom dans l'extraction.

```
des[, "de1"]
```

```
## 11 12 13 14 15 16 17 18 19 110
##  2  3  4  1  2  3  5  6  5  4
```

---

## 4 Arrays

Une matrice est en fait un cas particulier d'array, à deux dimensions.

```
is.array(mat)
```

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

L'array est un objet atomique pouvant posséder un nombre quelconque de dimensions.

### 4.1 Obtention d'informations

Reprenons l'array `arr` illustré précédemment.

```
str(arr)
```

```
## num [1:2, 1:3, 1:4] 2 4 3 1 6 3 9 2 6 3 ...
```

Comme pour une matrice, la fonction `dim` retourne les tailles de ses dimensions.

```
dim(arr)
```

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

Sa longueur est aussi le nombre total d'éléments qu'il contient.

```
length(arr)
```

```
## [1] 24
```

## 4.2 Création

### 4.2.1 Fonction array

La fonction `array` sert à créer des arrays. Il faut lui fournir le vecteur des éléments à stocker dans l'array et les tailles des dimensions de celui-ci, comme dans cet exemple.

```
collection <- array(1:12, dim = c(2, 2, 3))
collection
```

```
## , , 1
##
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## , , 2
##
##      [,1] [,2]
## [1,]    5    7
## [2,]    6    8
##
## , , 3
##
##      [,1] [,2]
## [1,]    9   11
## [2,]   10   12
```

Les arrays sont utiles, notamment, pour ranger dans une même structure plusieurs matrices de même taille.

### 4.2.2 Fonction as.array

Il est aussi possible de tenter de créer un array à partir d'un objet R quelconque avec `as.array`. Par exemple, les vecteurs ne sont pas considérés comme des types particuliers d'arrays.

```
vec
```

```
## [1] 2.0 4.9 2.0 0.2
```

```
is.array(vec)
```

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

Cependant, un vecteur peut être transformé en array comme suit.

```
array_vec <- as.array(vec)
array_vec
```

```
## [1] 2.0 4.9 2.0 0.2
```

```
is.array(array_vec)
```

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

Même si les affichages des objets `vec` et `array_vec` sont identiques, les deux objets ne sont pas du même type.

## 4.3 Métadonnées

Pour donner des noms aux éléments selon les dimensions d'un array, il faut utiliser une liste comme suit.

```
dimnames(collection) <- list(
  paste0("ligne", 1:2),
  paste0("colonne", 1:2),
  paste0("matrice", 1:3)
)
collection
```

```
## , , matrice1
##
##      colonne1 colonne2
## ligne1      1      3
## ligne2      2      4
##
## , , matrice2
##
##      colonne1 colonne2
## ligne1      5      7
## ligne2      6      8
##
## , , matrice3
##
##      colonne1 colonne2
## ligne1      9     11
## ligne2     10     12
```

```
str(collection)
```

```
## int [1:2, 1:2, 1:3] 1 2 3 4 5 6 7 8 9 10 ...
## - attr(*, "dimnames")=List of 3
## ..$ : chr [1:2] "ligne1" "ligne2"
## ..$ : chr [1:2] "colonne1" "colonne2"
## ..$ : chr [1:3] "matrice1" "matrice2" "matrice3"
```

## 4.4 Extraction d'éléments

Quelques exemples :

- extraction de la première matrice,

```
collection[, , 1]
```

```
##      colonne1 colonne2
## ligne1      1      3
## ligne2      2      4
```

- extraction de l'élément en position (1,2) de toutes les matrices,

```
collection[1, 2, ]
```

```
## matrice1 matrice2 matrice3
##      3      7     11
```

- extraction de la deuxième ligne des matrices 2 et 3

```
collection[2, , 2:3]
```

```
##      matrice2 matrice3
## colonne1      6     10
## colonne2      8     12
```

---

## 5 Listes

La liste est semblable au vecteur dans le sens qu'elle combine des éléments. Cependant, les éléments d'un vecteur sont des données, toutes de même type. Les éléments d'une liste sont plutôt des objets, de types quelconques. Il s'agit d'un objet récursif.

Les fonctions qui ont plus d'un objet à retourner en sortie (ex. la fonction `lm` pour faire de la régression) retournent une liste de tous ces objets.

### 5.1 Obtention d'informations

Jetons un coup d'oeil à la liste `liste` tirée de l'introduction.

```
str(liste)

## List of 3
## $ : num [1:2, 1:2] 2 4 3 1
## $ :List of 2
## ..$ : num [1:2] 2 3
## ..$: num 8
## $ :'data.frame': 3 obs. of 2 variables:
## ..$ V1: num [1:3] 2 4 3
## ..$ V2: num [1:3] 1 5 3
```

Elle contient une matrice, une autre liste et un data frame. La liste est de longueur 3 car elle comporte trois sous-objets.

```
length(liste)
```

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

### 5.2 Création

#### 5.2.1 Fonction `list`

Nous venons tout juste d'utiliser la fonction de création de listes nommée `list` pour créer la valeur à assigner à l'attribut `dimnames` de l'array `collection`.

```
noms <- list(
  paste0("ligne", 1:2),
  paste0("colonne", 1:2),
  paste0("matrice", 1:3)
)
noms

## [[1]]
## [1] "ligne1" "ligne2"
##
## [[2]]
## [1] "colonne1" "colonne2"
##
## [[3]]
## [1] "matrice1" "matrice2" "matrice3"
```

Pour utiliser cette fonction, il suffit de lui donner en entrée tous les objets à inclure dans la liste, dans l'ordre souhaité.

### 5.2.2 Fonction `as.list`

La fonction `as.list` permet quant à elle de transformer n'importe quel objet R en liste. Par exemple, en appliquant la fonction `as.list` sur un vecteur, elle crée une liste pour laquelle chaque élément du vecteur est devenu un vecteur de longueur 1 en élément de la liste.

```
as.list(vec)
```

```
## [[1]]
## [1] 2
##
## [[2]]
## [1] 4.9
##
## [[3]]
## [1] 2
##
## [[4]]
## [1] 0.2
```

Pour obtenir plutôt une liste de longueur 1 comportant le vecteur comme seul élément, il aurait plutôt fallu utiliser la fonction `list` comme suit.

```
list(vec)
```

```
## [[1]]
## [1] 2.0 4.9 2.0 0.2
```

### 5.2.3 Fonction `vector`

La fonction `vector` permet aussi de créer des listes (rappelons qu'une liste est considérée être un vecteur récursif).

```
ex <- vector(mode = "list", length = 2)
ex
```

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

Une liste créée avec la fonction `vector`, comme dans l'exemple ci-dessus, est initialement vide. Elle est remplie en assignant des objets à ses éléments.

```
ex[[1]] <- matrix(1:8, nrow = 2, ncol = 4, byrow = TRUE)
ex
```

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

## 5.3 Concaténation de listes

### 5.3.1 Fonction c

Les éléments de plusieurs listes peuvent être concaténés avec la fonction `c` comme suit.

```
combine <- c(noms, ex)
combine

## [[1]]
## [1] "ligne1" "ligne2"
##
## [[2]]
## [1] "colonne1" "colonne2"
##
## [[3]]
## [1] "matrice1" "matrice2" "matrice3"
##
## [[4]]
##      [,1] [,2] [,3] [,4]
## [1,]   1   2   3   4
## [2,]   5   6   7   8
##
## [[5]]
## NULL
```

Ainsi, la fonction `c` n'est pas seulement utile avec les vecteurs atomiques, elle l'est aussi avec les vecteurs récurrents.

## 5.4 Métadonnées

Les éléments d'une liste peuvent être nommés en utilisant la fonction `names` sur une liste déjà existante comme suit.

```
names(noms) <- c("lignes", "colonnes", "matrices")
noms

## $lignes
## [1] "ligne1" "ligne2"
##
## $colonnes
## [1] "colonne1" "colonne2"
##
## $matrices
## [1] "matrice1" "matrice2" "matrice3"

str(noms)

## List of 3
## $ lignes : chr [1:2] "ligne1" "ligne2"
## $ colonnes: chr [1:2] "colonne1" "colonne2"
## $ matrices: chr [1:3] "matrice1" "matrice2" "matrice3"
```

Ils peuvent aussi être nommés directement dans une commande de création avec la fonction `list`, en précédant la spécification de chaque objet par un nom et l'opérateur `=`.

```
exemple <- list(mat = des, fct = casefold, liste = noms)
exemple
```

```

## $mat
##      de1 de2
## 11    2  1
## 12    3  4
## 13    4  2
## 14    1  3
## 15    2  5
## 16    3  4
## 17    5  6
## 18    6  2
## 19    5  5
## 110   4  3
##
## $fct
## function (x, upper = FALSE)
## if (upper) toupper(x) else tolower(x)
## <bytecode: 0x000002c34dfa7768>
## <environment: namespace:base>
##
## $liste
## $liste$lignes
## [1] "ligne1" "ligne2"
##
## $liste$colonnes
## [1] "colonne1" "colonne2"
##
## $liste$matrices
## [1] "matrice1" "matrice2" "matrice3"

```

## 5.5 Extraction d'éléments

### 5.5.1 Un seul élément

Voici quelques exemples :

```
noms[1]
```

```
## $lignes
## [1] "ligne1" "ligne2"
```

```
noms[[1]]
```

```
## [1] "ligne1" "ligne2"
```

Nous avons déjà vu un exemple similaire. Il est important de remarquer ici que `noms[1]` est une liste de longueur 1, alors que `noms[[1]]` est l'objet en position 1 dans la liste `noms`, soit ici un vecteur. L'opérateur `[[` sort l'élément extrait de la structure de l'objet d'origine si celui-ci est récursif.

D'ailleurs, puisqu'une liste est un objet récursif, l'opérateur `$` peut être utilisé pour extraire un de ses éléments. Cependant, l'opérateur `$` est seulement utilisable lorsque les éléments de la liste sont nommés. Il faut inscrire le nom de l'élément à extraire après l'opérateur, comme dans cet exemple.

```
noms$lignes
```

```
## [1] "ligne1" "ligne2"
```

Cette dernière opération est équivalente à la suivante.

```
noms[["lignes"]]
```

```
## [1] "ligne1" "ligne2"
```

Contrairement à l'opérateur `[`, l'opérateur `$` n'exige pas que le nom de l'élément à extraire soit encadré de guillemets, sauf s'il comporte un ou des espaces.

Il est parfois utile d'utiliser des opérateurs d'extraction à la chaîne. Par exemple, la commande suivante extrait l'élément `lignes` dans le sous-objet `liste` de la liste `exemple`.

```
exemple$liste$lignes
```

```
## [1] "ligne1" "ligne2"
```

### 5.5.2 Plusieurs éléments

Pour extraire d'un coup plusieurs éléments d'une liste, il faut employer l'opérateur `[`. Voici un exemple.

```
noms[1:2]
```

```
## $lignes
## [1] "ligne1" "ligne2"
##
## $colonnes
## [1] "colonne1" "colonne2"
```

### 5.5.3 Extraction par exclusion

La façon la plus simple de retirer un élément d'une liste est d'assigner la valeur `NULL` à l'élément en question.

```
noms$lignes <- NULL
noms
```

```
## $colonnes
## [1] "colonne1" "colonne2"
##
## $matrices
## [1] "matrice1" "matrice2" "matrice3"
```

## 5.6 Conversion en vecteur atomique

Si les éléments de plus bas niveau d'une liste sont tous des données de même type, on pourrait vouloir les mettre bout à bout dans un vecteur. La fonction `unlist` permet de réaliser facilement cette tâche, comme dans cet exemple.

```
liste
```

```
## [[1]]
##      [,1] [,2]
## [1,]    2    3
## [2,]    4    1
##
## [[2]]
## [[2]][[1]]
## [1] 2 3
##
## [[2]][[2]]
## [1] 8
##
```

```
##
## [[3]]
##   V1 V2
## 1  2  1
## 2  4  5
## 3  3  3

unlist(liste)

##
##           V11 V12 V13 V21 V22 V23
##    2    4    3    1    2    3    8    2    4    3    1    5    3
```

---

## 6 Data frames

Peu de langages de programmation possèdent une structure de données équivalente au data frame de R. Il s'agit d'une structure spécifiquement conçue pour stocker des jeux de données.

### 6.1 Jeux de données

Afin de mieux comprendre le data frame, rappelons d'abord quelques définitions relatives à un jeu de données (définitions sont tirées de [2]).

#### 6.1.1 Définitions

- **Données** : En statistique, des données sont des valeurs numériques (des nombres) ou alphanumériques (des chaînes de caractères) représentant les observations de certaines variables sur certains individus. Elles se présentent souvent sous la forme de jeux de données, c'est-à-dire de tableaux de valeurs, stockées dans un fichier informatique.
- **Population** : La population est l'ensemble de référence sur lequel porte l'étude dans le cadre de laquelle les données ont été recueillies.
- **Individu ou unité statistique** : Un individu est un élément de la population. L'ensemble des individus constitue la population. Chaque observation est associée à un individu.
- **Échantillon** : L'échantillon est un sous-groupe de la population, composé des individus pour lesquels des observations ont été recueillies. Si des mesures ont été prises pour tous les individus de la population, il s'agit d'un recensement.
- **Variable** : Le terme variable désigne la représentation d'une caractéristique des individus. Ainsi, une variable n'est pas la caractéristique elle-même, mais plutôt une mesure de cette caractéristique.
- **Observation** : Une observation est l'ensemble des valeurs obtenues en mesurant des variables sur un individu de la population.

#### 6.1.2 Représentation en R

Dans un logiciel informatique, une façon courante de stocker un jeu de données est d'utiliser une structure de données à deux dimensions et d'y placer les observations en lignes et les variables en colonnes. En R, si toutes les variables sont du même type, il est possible d'utiliser une matrice. Cependant, lorsque les valeurs observées des différentes variables ne sont pas du même type, par exemple si certaines sont numériques et d'autres caractères, il est impossible de regrouper ces variables au sein d'une seule matrice. Le data frame a été créé pour stocker un tel jeu de données.

Observons de nouveau la structure du data frame illustré en introduction.

```
dat
```

```
##   V1      V2   V3
## 1 3.5   matin  NA
## 2 7.8 après-midi FALSE
## 3 9.9     soir  FALSE
## 4 5.7     nuit  TRUE
```

Il s'agit d'un exemple de jeu de données sous la forme « observations en lignes, variables en colonnes ». La sortie de la fonction `str` appliquée à un data frame énonce clairement cette interprétation du data frame.

```
str(dat)
```

```
## 'data.frame':   4 obs. of  3 variables:
## $ V1: num  3.5 7.8 9.9 5.7
## $ V2: chr  "matin" "après-midi" "soir" "nuit"
## $ V3: logi  NA FALSE FALSE TRUE
```

Ici, la mention « 4 obs. of 3 variables » signifie que les lignes doivent être vues comme des observations et les colonnes comme des variables.

## 6.2 Obtention d'informations

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

```
attributes(dat)
```

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

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

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

```
dim(dat)
```

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

```
nrow(dat)
```

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

```
ncol(dat)
```

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

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

```
length(dat)
```

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

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

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

```
typeof(dat)
```

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

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

```
names(dat)
```

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

```
colnames(dat)
```

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

## 6.3 Création

### 6.3.1 Fonction `data.frame`

Ajoutons à la matrice `des` créée précédemment, qui contient des données fictives de lancers de deux dés, une colonne contenant le nom de la personne qui a lancé le dé. Le résultat ne peut pas être une matrice si nous voulons conserver des données de type numérique dans les deux premières colonnes et caractère dans la troisième colonne. Il faut donc créer un data frame, ce que réalise la commande suivante.

```
des_plus <- data.frame(des, lanceur = rep(c("Luc", "Kim"), each = 5))
des_plus
```

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

La principale fonction de création d'un data frame se nomme `data.frame`. Il faut lui donner en entrée tous les vecteurs et/ou facteurs que le data frame doit contenir, dans l'ordre désiré. La fonction `data.frame` accepte aussi en entrée des matrices, comme dans l'exemple précédent, auquel cas les colonnes des matrices en entrée deviennent des colonnes dans le data frame en sortie.

Regardons la structure interne du data frame que nous venons de créer.

```
str(des_plus)
```

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

Les trois colonnes de `des_plus` sont des vecteurs.

### Remarque : changement depuis la version 4.0.0 de R

Avec une version de R antérieure à 4.0.0, la troisième colonne de `des_plus` serait un facteur plutôt qu'un vecteur, parce que la valeur par défaut de l'argument `stringsAsFactors` de la fonction `data.frame` était

auparavant TRUE.

```
des_plus_fac <- data.frame(
  des,
  lanceur = rep(c("Luc", "Kim"), each = 5),
  stringsAsFactors = TRUE
)
str(des_plus_fac)
```

```
## 'data.frame':  10 obs. of  3 variables:
## $ de1      : num  2 3 4 1 2 3 5 6 5 4
## $ de2      : num  1 4 2 3 5 4 6 2 5 3
## $ lanceur: Factor w/ 2 levels "Kim","Luc": 2 2 2 2 2 1 1 1 1 1
```

La valeur TRUE assignée à l'argument `stringsAsFactors` a pour conséquence que `data.frame` transforme les vecteurs contenant des données de type caractère en facteur.

Notons que si vous aimeriez revenir à l'ancien comportement par défaut de R, vous pouvez modifier globalement la valeur par défaut de l'argument `stringsAsFactors` comme suit.

```
options(stringsAsFactors = TRUE)
```

Cette commande provoque un changement qui affecte toutes les fonctions ayant un argument `stringsAsFactors` (la fonction `data.frame` n'est pas la seule à avoir cet argument).

### 6.3.2 Fonction `as.data.frame`

Pour transformer, si possible, un objet R en data frame, il faut utiliser la fonction `as.data.frame`. Par exemple, transformons la matrice `mat` en data frame.

```
df_mat <- as.data.frame(mat)
df_mat
```

```
##      V1    V2    V3
## 1  TRUE FALSE FALSE
## 2  TRUE  TRUE FALSE
## 3 FALSE  TRUE  TRUE
## 4  TRUE  TRUE  TRUE
```

## 6.4 Concaténation de data frames

### 6.4.1 Fonction `data.frame`

En plus de servir à créer des data frames, la fonction `data.frame` permet d'en combiner par concaténation de colonnes.

```
jour <- data.frame(jour = rep(1, 10))
str(jour)
```

```
## 'data.frame':  10 obs. of  1 variable:
## $ jour: num  1 1 1 1 1 1 1 1 1 1
```

```
autre <- data.frame(des_plus, jour)
str(autre)
```

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

### 6.4.2 Fonctions rbind et cbind

Une concaténation de data frames peut aussi s'effectuer avec les fonctions `rbind` et `cbind`. Comme avec une matrice, `rbind` effectue une concaténation en lignes et `cbind` en colonnes.

La fonction `rbind` permet donc de mettre en commun les observations de plusieurs data frames, comme dans cet exemple.

```
des_plus_exp_2 <- data.frame(de1 = c(3, 5), de2 = c(4, 2), lanceur = "Paul")
des_plus_exp_3 <- data.frame(de1 = c(6, 1), de2 = c(2, 5), lanceur = "Paul")
rbind(des_plus, des_plus_exp_2, des_plus_exp_3)
```

```
##      de1 de2 lanceur
## 11     2  1      Luc
## 12     3  4      Luc
## 13     4  2      Luc
## 14     1  3      Luc
## 15     2  5      Luc
## 16     3  4      Kim
## 17     5  6      Kim
## 18     6  2      Kim
## 19     5  5      Kim
## 110    4  3      Kim
## 1      3  4      Paul
## 2      5  2      Paul
## 11     6  2      Paul
## 21     1  5      Paul
```

Pour que la concaténation fonctionne, les noms des variables (colonnes) dans les data frames doivent concorder.

Une concaténation de data frames avec `cbind` est quant à elle similaire à une concaténation avec `data.frame` : des variables sont rassemblées dans un seul data frame.

```
cbind(des_plus, jour)
```

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

Cette fois, les noms des observations (lignes) n'ont pas besoin de concorder entre les data frames, mais les nombres d'observations doivent être égaux.

## 6.5 Métadonnées

La fonction `data.frame` attribue toujours par défaut des noms aux lignes et colonnes des data frames qu'elle crée. Voyons un exemple.

```
essai <- data.frame(1:5, letters[1:5])
essai
```

```
##      X1.5 letters.1.5.
```

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

```
attributes(essai)
```

```
## $names
## [1] "X1.5" "letters.1.5."
##
## $class
## [1] "data.frame"
##
## $row.names
## [1] 1 2 3 4 5
```

Les noms par défaut des colonnes ne sont pas toujours pertinents. Il vaut donc mieux nommer les colonnes lors de la création, directement dans l'appel à la fonction `data.frame`. Pour ce faire, il suffit de précéder les éléments à combiner d'un nom et de l'opérateur `=`, comme dans cet exemple.

```
data.frame(chiffre = 1:5, lettre = letters[1:5])
```

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

Avec cette technique, les noms ont besoin d'être encadrés de guillemets uniquement s'ils comprennent un ou des espaces.

Les noms des colonnes d'un data frame peuvent aussi être remplacés a posteriori avec la fonction `names` ou `colnames` accompagnée d'une assignation.

```
names(essai) <- c("chiffre", "lettre")
```

## 6.6 Extraction d'éléments

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

### 6.6.1 En traitant le data frame comme une liste

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

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

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

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

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

```
des_plus[[1]]
```

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

```
des_plus$de1
```

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

### 6.6.2 En traitant le data frame comme une matrice

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

```
des_plus[, "de1"]
```

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

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

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

```
des_plus[[1, 1]]
```

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

### 6.6.3 Extraction par exclusion

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

```
autre$jour <- NULL
autre
```

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

## 6.7 Extensions

Certains packages R offrent des structures de données alternatives au data frame. Deux de ces structures sont mentionnées ici, car elles sont de plus en plus utilisées dans la communauté R. Elles ne sont cependant pas approfondies.

### 6.7.1 Tibbles

Les packages du [tidyverse](#) utilisent des [tibbles](#) en remplacement des data frames. Le [tidyverse](#) est une collection de packages dit être spécialisés en « science des données ». Plusieurs des développeurs de ces packages sont affiliés à [RStudio](#). Cette collection comprend notamment le très populaire package [ggplot2](#) et d'autres packages développés par Hadley Wickham.

Voyons un exemple de tibble.

```
library(tibble)
des_plus_tibble <- tibble(
  de1 = c(2, 3, 4, 1, 2, 3, 5, 6, 5, 4),
  de2 = c(1, 4, 2, 3, 5, 4, 6, 2, 5, 3),
  lanceur = rep(c("Luc", "Kim"), each = 5)
)
des_plus_tibble
```

```
## # A tibble: 10 x 3
##   de1   de2 lanceur
##   <dbl> <dbl> <chr>
## 1     2     1 Luc
## 2     3     4 Luc
## 3     4     2 Luc
## 4     1     3 Luc
## 5     2     5 Luc
## 6     3     4 Kim
## 7     5     6 Kim
## 8     6     2 Kim
## 9     5     5 Kim
## 10    4     3 Kim
```

```
str(des_plus_tibble)
```

```
## tibble[,3] [10 x 3] (S3: tbl_df/tbl/data.frame)
## $ de1      : num [1:10] 2 3 4 1 2 3 5 6 5 4
## $ de2      : num [1:10] 1 4 2 3 5 4 6 2 5 3
## $ lanceur: chr [1:10] "Luc" "Luc" "Luc" "Luc" ...
```

Un tibble se manipule comme un data frame, mais il possède les distinctions suivantes :

- l'affichage (*print*) a été repensé : pas plus de 10 lignes affichées par défaut, abréviations des types des colonnes affichées ;
- l'extraction d'éléments d'un tibble avec l'opérateur [ retourne toujours un autre tibble, sa sortie n'est jamais simplifiée (pas d'argument `drop`) ;
- la règle de recyclage lors de la création d'un tibble s'applique seulement pour des vecteurs de longueur unitaire ;
- certaines opérations d'extraction sont plus rapides.

**Pour en apprendre davantage :**

Voici quelques références pour ceux intéressés à en apprendre davantage à propos des tibbles :

- <http://tibble.tidyverse.org/articles/tibble.html>

- <http://r4ds.had.co.nz/tibbles.html>
- [http://blog.jumpingrivers.com/posts/2018/trouble\\_with\\_tibbles/](http://blog.jumpingrivers.com/posts/2018/trouble_with_tibbles/)

### 6.7.2 Data tables

Le `data table` est une extension du data frame développée dans le but de rendre plus rapides et conviviales les manipulations de jeu de données en R.

Voyons un exemple de data table.

```
library(data.table)
des_plus_data_table <- data.table(
  de1 = c(2, 3, 4, 1, 2, 3, 5, 6, 5, 4),
  de2 = c(1, 4, 2, 3, 5, 4, 6, 2, 5, 3),
  lanceur = rep(c("Luc", "Kim"), each = 5)
)
des_plus_data_table
```

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

```
str(des_plus_data_table)
```

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

La force du data table est la rapidité d'exécution des manipulations et la possibilité d'en faire beaucoup plus avec l'opérateur `[` comparativement au data frame. Pour illustrer ce dernier point, voici un exemple de calcul de statistiques descriptives avec l'opérateur `[`.

```
des_plus_data_table[
  j = .(moy_de1 = mean(de1), moy_de2 = mean(de2)),
  by = lanceur
]
```

```
##      lanceur moy_de1 moy_de2
## 1:      Luc      2.4      3
## 2:      Kim      4.6      4
```

Les deux premiers arguments (`i` et `j`) de l'opérateur `[` acceptent plus de valeurs pour un data table que pour un data frame. Dans le cas d'un data frame, ces arguments servent uniquement à identifier des lignes et des colonnes. Pour un data table, ils ont plus d'utilités et servent notamment à demander le calcul de statistiques sur une partie des données. Dans l'exemple ci-dessus, le calcul de moyennes sur les variables `de1` et `de2` est demandé. La syntaxe pour demander de tels calculs est simple.

De plus, l'opérateur `[` pour le data table possède plusieurs autres arguments, dont l'argument `by`. Dans la commande précédente, l'argument `by` a permis de demander un calcul distinct selon la valeur d'une variable.

Dans les valeurs fournies aux arguments de l'opérateur `[` pour le data table, il est possible de référer aux colonnes du data table par leurs noms sans encadrer ces noms de guillemets.

Nous verrons dans un prochain cours comment produire avec les fonctionnalités de base de R les mêmes calculs que ceux de l'exemple exploité ici.

### Pour en apprendre davantage :

Voici quelques références pour ceux intéressés à en apprendre davantage à propos des data tables :

- <https://cran.r-project.org/web/packages/data.table/vignettes/datatable-intro.html>
  - <https://rdatatable.gitlab.io/data.table/>
  - [https://stt4230.rbind.io/tutoriels\\_etudiants/hiver\\_2017/data.table/](https://stt4230.rbind.io/tutoriels_etudiants/hiver_2017/data.table/)
- 

## 7 Facteurs

Le facteur possède plusieurs propriétés du vecteur, mais il est conçu pour stocker les valeurs observées d'une variable catégorique. Son nom vient de la terminologie utilisée en analyse de la variance (ANOVA).

### 7.1 Obtention d'informations

Reprenons l'exemple en introduction.

```
fac
```

```
## [1] 5 2 5 5
## Levels: 2 5
```

Le facteur possède des attributs de plus qu'un vecteur, soit un identifiant de ses niveaux (modalités possibles) ainsi qu'une classe pour spécifier qu'il s'agit d'un facteur.

```
attributes(fac)
```

```
## $levels
## [1] "2" "5"
##
## $class
## [1] "factor"
```

La fonction `levels` retourne les niveaux d'un facteur, et la fonction `nlevels` retourne le nombre de niveaux.

```
levels(fac)
```

```
## [1] "2" "5"
```

```
nlevels(fac)
```

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

Pour prendre le moins de place possible en mémoire, les données dans un facteur sont toujours stockées sous la forme d'entiers : les chiffres 1 à disons  $l$  où  $l$  est le nombre de niveaux du facteur. La sortie de `str` avec un facteur affiche ces entiers.

```
str(fac)
```

```
## Factor w/ 2 levels "2","5": 2 1 2 2
```

Cependant, les niveaux, ou modalités, du facteur sont stockés sous forme de caractère, peu importe leur nature d'origine.

## 7.2 Création

### 7.2.1 Fonctions `factor` et `as.factor`

Pour créer un facteur de toutes pièces, il faut faire appel à la fonction `factor` ou `as.factor`.

```
reponses <- factor(rep(c("roche", "papier", "ciseau"), times = 3))
reponses

## [1] roche papier ciseau roche papier ciseau roche papier ciseau
## Levels: ciseau papier roche

str(reponses)

## Factor w/ 3 levels "ciseau","papier",...: 3 2 1 3 2 1 3 2 1
```

Les niveaux du facteur sont extraits automatiquement par la fonction `factor` et placés en ordre alphanumérique. Pour contrôler l'ordre des facteurs, il faut spécifier cet ordre dans la commande de création du facteur avec l'argument `levels`.

```
reponses <- factor(
  rep(c("roche", "papier", "ciseau"), times = 3),
  levels = c("roche", "papier", "ciseau")
)
reponses

## [1] roche papier ciseau roche papier ciseau roche papier ciseau
## Levels: roche papier ciseau

str(reponses)

## Factor w/ 3 levels "roche","papier",...: 1 2 3 1 2 3 1 2 3
```

Il est utile de contrôler l'ordre des niveaux d'un facteur, car dans des sorties ou graphiques produits avec un facteur, l'ordre de ses niveaux est respecté.

La fonction `as.factor` agit comme la fonction `factor`, mais elle ne possède pas d'arguments supplémentaires au vecteur à transformer.

## 7.3 Modification

### 7.3.1 Fonction `levels`

Pour modifier les libellés des niveaux, il suffit de combiner l'utilisation de la fonction `levels` à une assignation de données comme suit.

```
levels(reponses) <- c("pomme", "orange", "poire")
reponses

## [1] pomme orange poire pomme orange poire pomme orange poire
## Levels: pomme orange poire

str(reponses)

## Factor w/ 3 levels "pomme","orange",...: 1 2 3 1 2 3 1 2 3
```

### 7.3.2 Réordonnement des niveaux

Il est toujours possible de modifier l'ordre des niveaux d'un facteur après sa création. Pour ce faire, il faut de nouveau faire appel à la fonction `factor`, en lui donnant en entrée le facteur et les niveaux dans l'ordre désiré. En réassignant le résultat au nom du facteur, il en ressort modifié, comme dans l'exemple suivant.

```
reponses <- factor(reponses, levels = c("poire", "pomme", "orange"))
reponses
```

```
## [1] pomme orange poire pomme orange poire pomme orange poire
## Levels: poire pomme orange
```

```
str(reponses)
```

```
## Factor w/ 3 levels "poire","pomme",...: 2 3 1 2 3 1 2 3 1
```

## 7.4 Métadonnées

Les éléments d'un facteur peuvent être nommés, comme pour un vecteur.

```
names(reponses) <- letters[1:9]
reponses
```

```
##      a      b      c      d      e      f      g      h      i
## pomme orange poire pomme orange poire pomme orange poire
## Levels: poire pomme orange
```

```
str(reponses)
```

```
## Factor w/ 3 levels "poire","pomme",...: 2 3 1 2 3 1 2 3 1
## - attr(*, "names")= chr [1:9] "a" "b" "c" "d" ...
```

## 7.5 Extraction d'éléments

L'extraction d'éléments d'un facteur se réalise comme avec un vecteur. Par exemple :

```
extrait <- reponses[1:2]
extrait
```

```
##      a      b
## pomme orange
## Levels: poire pomme orange
```

```
nlevels(extrait)
```

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

Ici, aucun élément contenant la modalité `poire` n'est conservé. Pourtant, dans les métadonnées du facteur, il est encore inscrit que le facteur possède 3 niveaux. Pour effacer des métadonnées d'un facteur les niveaux non observés, il faut utiliser la fonction `droplevels`.

```
extrait_2 <- droplevels(extrait)
extrait_2
```

```
##      a      b
## pomme orange
## Levels: pomme orange
```

```
nlevels(extrait_2)
```

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

## 7.6 Facteurs ordonnés

Il est possible de spécifier un ordre dans les valeurs des niveaux d'un facteur. Il ne s'agit pas ici simplement de l'ordre dans lequel sont placés les niveaux qui a été mentionné précédemment. Il s'agit plutôt des valeurs

relatives des niveaux les uns par rapport aux autres. Pour une variable catégorique ordinaire, les modalités ou niveaux de la variable peuvent être ordonnés. Il existe une structure de données R pour stocker une telle variable. Il s'agit des facteurs ordonnés.

Créons par exemple un facteur ordonné contenant des niveaux de satisfaction de clients.

```
satisfaction <- factor(
  c("très satisfait", "satisfait", "très satisfait", "insatisfait",
    "très satisfait", "satisfait", "satisfait"),
  levels = c("très insatisfait", "insatisfait", "satisfait", "très satisfait"),
  ordered = TRUE
)
satisfaction

## [1] très satisfait satisfait      très satisfait insatisfait      très satisfait
## [6] satisfait      satisfait
## Levels: très insatisfait < insatisfait < satisfait < très satisfait

str(satisfaction)
```

```
## Ord.factor w/ 4 levels "très insatisfait"<..: 4 3 4 2 4 3 3
```

Grâce à l'argument `ordered = TRUE`, l'objet obtenu est plus qu'un facteur. Il s'agit d'un facteur ordonné. La distinction entre le facteur et le facteur ordonné est utilisée par certaines fonctions R, notamment des fonctions d'ajustement de modèles.

---

## 8 Conversions de type de données

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

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

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

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

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

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

Voici un autre exemple.

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

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

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

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

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

caractère > réel > entier > logique.

## 8.1 Conversions explicites

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

**Exemple :** Comment créer un vecteur d’entiers ?

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
typeof(x)
```

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

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

### Conversions explicites de chaînes de caractères vers des nombres

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

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

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

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

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

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

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

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

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

---

## 9 Résumé

### Contenant versus contenu

Objet R servant de structure de données = CONTENANT

Éléments stockés dans cet objet = CONTENU :

- **données** : valeurs observées de variables aléatoires, valeurs de résultats de calculs (ex. : statistiques descriptives, coefficients d'un modèle), etc. ;
- **métadonnées = attributs** (souvent facultatifs) :  
nom des éléments (par dimension), taille des dimensions pour les matrices et arrays, niveaux d'un facteur, etc.

### Caractéristiques des structures de données en R

- Type de l'objet : `is.(atomic/recursive)`, `is.(vector/matrix/array/list/data.frame/factor)`
- Type des données en éléments : `typeof`, `is.(numeric/character/logical/...)`
  - valeurs numériques (`numeric`) :  
entières (`integer`) ou réelles (`double`)
  - chaînes de caractères (`character`)
  - valeurs logiques (`logical`)
  - types moins communs de valeurs :  
nombres complexes (`complex`), bits (`raw`), etc.
  - si éléments = sous-objets (par exemple pour une liste ou un data frame), `typeof` retourne la valeur `"list"`
- Nombre total d'éléments : `length`

### Types de structures de données en R

Nombre de dimensions	Objets atomiques	Objets récurifs
1	<b>vecteur</b>	<b>liste</b>
2	<b>matrice</b>	<b>data frame</b>
⋮	⋮	
n	<b>array</b>	-

- Objets atomiques : Éléments = données toutes du même type
- Objets récurifs : Éléments = objets de types quelconques

**facteur** = une généralisation du vecteur pour stocker les observations d'une variable catégorique

### Attributs selon le type d'objet

Fonctions pour extraire des attributs : **attributes** (tous les attributs), **attr** (un seul attribut), **names**, **dimnames**, **colnames**, **rownames**, **dim**, **nrow**, **ncol**, **class**, **levels**, etc.

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

Légende :

« - » = attribut non existant pour ce type d'objet

« ✓ » = attribut existe (non obligatoire si encadré de parenthèses)

« • » = métadonnée existante, mais pas dans les attributs obtenus avec la fonction **attributes**

### Extraction d'éléments

- Opérateurs d'indilage :
  - [ : extrait un ou plusieurs éléments, préserve la structure de l'objet, sauf pour un objet à plus d'une dimension si une simplification est possible et **drop = TRUE** (TRUE est la valeur par défaut de l'argument) ;
  - [[ et \$ : extrait un seul élément, laisse tomber des attributs d'un objet atomique, sort de la structure d'un objet récurif (\$ s'utilise seulement avec des objets récurifs) ;
- Fonctions : **head**, **tail**, **subset**.

Différentes façons d'identifier les éléments à extraire avec [ :

- vecteur de nombre(s) positif(s) (entre 1 et la taille de la dimension concernée) : *positions des éléments à sélectionner* ;
- vecteur de nombre(s) négatif(s) (entre 1 et la taille de la dimension concernée) :  $-1 \times$  *positions des éléments à mettre de côté* ;
- vecteur de chaîne(s) de caractères (possible seulement si les éléments sont nommés) : *noms des éléments à sélectionner* ;
- vecteur de logiques (de même longueur que la dimension concernée) : **TRUE** pour les éléments à sélectionner, **FALSE** pour ceux à mettre de côté ;
- rien : utile pour extraire tous les éléments selon une dimension.

### Remplacement d'éléments dans un objet

Afin de modifier des éléments dans un objet, il faut soumettre une commande contenant les 3 composantes suivantes :

1. une extraction avec l'opérateur [, [[ ou \$ pour identifier les éléments à remplacer (ou simplement le nom de l'objet s'il doit être remplacé dans son entièreté),
2. un opérateur d'assignation (<-),
3. les valeurs de remplacement, dans une structure correspondant à la section à remplacer dans l'objet, *par exemple* :

```
matriceA[1:2,3] <- c(3,5)
```

## Règle de recyclage

Lors de toute opération faisant intervenir 2 vecteurs de longueurs différentes, les éléments du vecteur le plus court sont répétés de façon à ce que ce vecteur devienne de la même longueur que le vecteur le plus long.

## Conversions de types de données

Les types de données communs, du moins contraignant au plus contraignant :

- caractères > réels > entiers > logiques

R fait automatiquement des **conversions implicites** en respectant cet ordonnancement des types de données.

Fonctions R pour effectuer des **conversions explicites** :

```
as.(character/double/integer/logical), type.convert
```

## Données manquantes

Peu importe le type, une **donnée manquante** est toujours représentée en R par NA.

## Vecteurs

Simple objet atomique à 1 dimension

- fonctions de création :
  - c, vector, as.vector;
  - valeurs répétées : rep;
  - séquences régulières : :, seq;
  - chaînes de caractères : paste, paste0, constantes (letters, LETTERS, month.name, etc.).
- fonctions d'ajout de valeurs :
  - par concaténation d'un autre vecteur : c;
  - append;
- comment réaliser une extraction avec un opérateur :
  - [ ou [[ avec un seul argument.

## Matrice

Généralisation du vecteur à 2 dimensions

- fonctions de création :
  - matrix, as.matrix;
  - en combinant des vecteurs par lignes : rbind;
  - en combinant des vecteurs par colonnes : cbind;
- fonctions d'ajout de valeurs :
  - par concaténation d'une autre matrice ou d'un vecteur : rbind, cbind;
- comment réaliser une extraction avec un opérateur :

- [ ou [[ avec deux arguments, soit un par dimension (ne pas fournir de valeur pour les dimensions à conserver complètes).

## Arrays

Généralisation du vecteur ou de la matrice à un nombre quelconque de dimensions

- fonctions de création :
  - `array`, `as.array`;
- fonctions d'ajout de valeurs :
  - par concaténation d'un autre array :  
`abind` du package `abind` (pas vu dans le cours);
- comment réaliser une extraction avec un opérateur :
  - [ ou [[ avec autant d'arguments qu'il y a de dimensions (ne pas fournir de valeur pour les dimensions à conserver complètes).

## Listes

Objet récursif à 1 dimension

- fonctions de création :
  - `list`, `as.list`, `vector`;
- fonctions d'ajout de valeurs :
  - par concaténation avec `c`;
- comment réaliser une extraction avec un opérateur :
  - [ ou [[ avec un seul argument ;
  - ou encore `$` pour extraire un élément nommée.

## Data frames

Objet récursif à 2 dimensions, conçu pour stocker :

- des jeux de données (ligne = observation, colonne = variable)
  - $\approx$  liste de vecteurs ou facteurs de même longueur (un élément = une colonne);
  - $\approx$  matrice avec données dont le type peut varier entre les colonnes;
- fonctions de création :
  - `data.frame`, `as.data.frame`;
- fonctions d'ajout de valeurs :
  - par concaténation avec : `data.frame`, `cbind`, `rbind`;
- comment réaliser une extraction avec un opérateur :
  - [ ou [[ avec 1 argument identifiant une ou des colonnes;
  - [ ou [[ avec 2 arguments, soit un par dimension;
  - ou encore `$` pour extraire une colonne nommée.

## Facteurs

Généralisation du vecteur conçu pour stocker les valeurs observées d'une variable catégorique.

- fonctions de création : `factor`, `as.factor`;
- fonctions en lien avec les niveaux du facteur :
  - pour extraire (ou modifier lorsqu'accompagné d'une assignation de valeurs) les niveaux : `levels`;
  - retourne le nombre de niveaux : `nlevels`;
  - pour éliminer des niveaux non présents : `droplevels`;
- comment réaliser une extraction avec un opérateur :
  - [ ou [[ avec un seul argument.

# Références

## Références citées dans le texte :

[1] Bryan, J. (2020). *Data wrangling, exploration, and analysis with R > Table of atomic R object flavors*. University of British Columbia et RStudio. URL <https://stat545.com/r-objects.html#table-of-atomic-r-object-flavors>

[2] Baillargeon, S., Rivest, L.-P., Simard, M., Ghazzali, N., Mérette, C., Belisle, C., Duchesne, T., Labbe, A. & Lakhali-Chaieb, L. (2013). *Analyse de tableaux de fréquences*. Notes de cours, STT-4400/STT-6210. Université Laval, Département de mathématiques et de statistique.

## Références supplémentaires :

- R Core Team (2020). *An Introduction to R*. R version 4.0.3. URL <https://cran.r-project.org/doc/manuals/r-release/R-intro.html>
- R Core Team (2020). *The R Language Definition*. R version 4.0.3. URL <https://cran.r-project.org/doc/manuals/r-release/R-lang.html>
- Wickham, H. (2019). *Advanced R*. 2e édition. Chapman and Hall/CRC.
  - <https://adv-r.hadley.nz/vectors-chap.html>
  - <https://adv-r.hadley.nz/subsetting.html>

Rappel des références pour en apprendre davantage à propos des tibbles :

- <http://tibble.tidyverse.org/articles/tibble.html>
- <http://r4ds.had.co.nz/tibbles.html>
- [http://blog.jumpingrivers.com/posts/2018/trouble\\_with\\_tibbles/](http://blog.jumpingrivers.com/posts/2018/trouble_with_tibbles/)

Rappel des références pour en apprendre davantage à propos des data tables :

- <https://cran.r-project.org/web/packages/data.table/vignettes/datatable-intro.html>
- <https://rdatatable.gitlab.io/data.table/>
- [https://stt4230.rbind.io/tutoriels\\_etudiants/hiver\\_2017/data.table/](https://stt4230.rbind.io/tutoriels_etudiants/hiver_2017/data.table/)