Structures de données en R

Sophie Baillargeon, Université Laval

2018-01-15

Table des matières

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.

Différents 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 récursifs
1	**vecteur**	**liste**
2	**matrice**	**data frame**
⋮	⋮	
n	**array**	

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 « **récursifs** ». En fait, les listes sont parfois appelées « vecteurs récursifs » dans la documentation de R. Ici, le mot vecteur, non accompagné de l'adjectif récursif, 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.

Exemple de vecteur

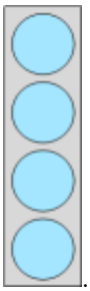
Le vecteur est l'objet le plus simple. Voici un exemple de vecteur :

```
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.

Exemple de matrice

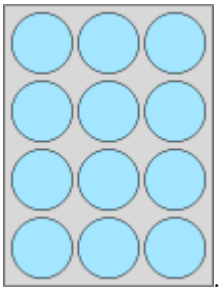
La matrice est une généralisation à 2 dimensions du vecteur. Voici un exemple de matrice :

```
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.

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 :

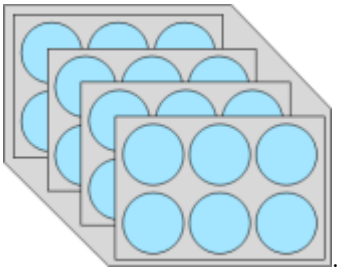
```
arr
```

```
## , , 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 :  .

Exemple de liste

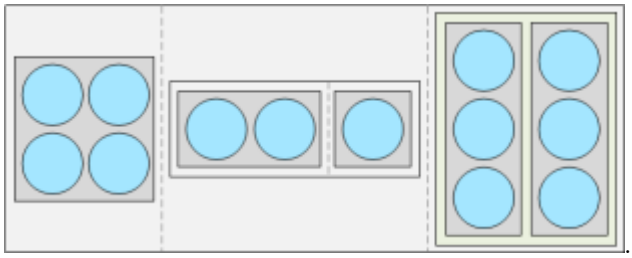
Une liste est pour sa part un objet récursif : elle contient d'autres objets, de types quelconques. Voici un exemple de liste :

```
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).

Exemple de data frame

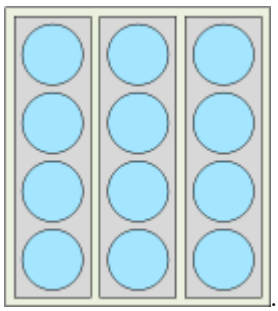
Le data frame est en quelque sorte un type particulier de liste, mais il partage aussi des caractéristiques avec les matrices, étant réputé posséder 2 dimensions (des lignes et des colonnes). Ses éléments sont des objets, mais ces objets doivent être tous de même longueur. Les éléments sont typiquement des vecteurs ou des facteurs, formant les colonnes du data frame. Les éléments d'un data frame peuvent aussi être des listes, mais cette possibilité n'est pas fréquemment exploitée. 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 première dimension d'un data frame est la longueur commune des objets qu'il contient (dans l'exemple

4) et 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. Ses dimensions sont vues comme un nombre de lignes ou d'observations et un nombre de colonnes ou de variables. Cependant, contrairement à une matrice, les données dans les différentes colonnes d'un data frame n'ont pas à être du même type.

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.

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`, `attr` et les fonctions raccourcies `dim`, `nrow`, `ncol`, `names`, `dimnames`, `rownames`, `colnames`.
- type des éléments : `typeof`, `mode`, `is.`(`numeric`/`character`/`logical`/...) ;
- nombre d'éléments : `length` ;

Ces fonctions seront illustrées plus loin. Notons que les fonction `typeof` et `mode` retournent, pour un objet atomique, le type des données en éléments, et retournent `"list"` pour un objet récursif.

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 la dimension de l'objet.

```
attributes(mat)
```

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

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

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

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

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

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

```
dim(mat)
```

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

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

```
nrow(mat)
```

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

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

```
ncol(mat)
```

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

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

```
attributes(vec)
```

```
## NULL
```

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

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

```
##  atomic [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 spécial `NULL`.

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

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

Différents 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.

Exemple de donnée réelle

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
```

Exemple de donnée entière

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 de fonctions en R traite par défaut les nombres comme des réels.

```
un <- 1
un
```

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

```
str(un)
```

```
##  num 1
```

```
typeof(un)
```

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

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

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

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

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

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

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

Exemple de donnée caractère

Une donnée caractère est une chaîne de caractères.

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

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

```r
str(ca)
```

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

```r
typeof(ca)
```

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

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

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

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

Exemple de donnée logique

Une donnée logique est simplement `TRUE` ou `FALSE`.

```r
lo <- TRUE
lo
```

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

```r
str(lo)
```

```
##  logi TRUE
```

```r
typeof(lo)
```

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

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

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

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

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.

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.

Opérateurs d'indiçage

Il y a trois opérateurs d'indiçage 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.

`[` 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
```

```
extrait1_liste <- liste[1]
str(extrait1_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

```
extrait2_liste <- liste[[1]]
str(extrait2_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
```

Extractions d'éléments dans un objet à plusieurs dimensions

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

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

```
## [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'avions 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"
```

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 $2^{\text{ième}}$ 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
```

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 positifs, dont les valeurs sont entre 1 et la longueur de la dimension concernée : *positions des éléments à conserver* ;
- un ou des nombres négatifs, dont les valeurs sont entre 1 et la longueur de la dimension concernée : `-` *positions des éléments à éliminer* ;
- un ou des chaînes de caractères (possible seulement si les éléments sont nommés) : *noms des éléments à conserver* ;

- vecteur de logiques (doit être de la même longueur que la dimension concernée) :
  `TRUE` *pour les éléments à conserver,* `FALSE` *pour ceux à éliminer* ;
- rien : utile par exemple pour extraire une ligne d'une matrice ou d'un data frame en conservant toutes les colonnes.

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 laisser tomber,

```
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'indiçage 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.

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 nom 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 la dimension de l'objet retourné. Ainsi, l'extraction suivante retourne un vecteur, alors que l'objet d'origine est un array à 3 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 la dimension 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"
```

Fonctions d'extraction

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

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.

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

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

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

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

Fonction `tail`

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

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

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

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

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

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.

```r
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.

```r
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`.

```r
subset(dat, subset = V1 > 5, select = - V2)
```

```
##    V1    V3
## 2 7.8  TRUE
## 3 9.9 FALSE
## 4 5.7  TRUE
```

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
```

Ça 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 court n'est pas un multiple de la longueur du vecteur le plus long, 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.

Le vecteur

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

Obtention d'informations sur un vecteur

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"
```

Fonctions de création d'un vecteur

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* (an 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"
```

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.

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), 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
```

La création de séquences avec l'opérateur ':' ou la fonction 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
```

Fonctions de concaténation de vecteurs

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
```

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
```

Ajout de métadonnées dans un vecteur

Supposons que l'objet de contient 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) <- paste("l", 1:10, sep = "")
```

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

```
de
```

```
##  l1  l2  l3  l4  l5  l6  l7  l8  l9 l10
##   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] "l1" "l2" "l3" "l4" ...
```

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

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

```
names(de)
```

```
##  [1] "l1"  "l2"  "l3"  "l4"  "l5"  "l6"  "l7"  "l8"  "l9"  "l10"
```

Cet attribut est un exemple de métadonnées. Il s'agit de données sur les 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 précéder 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) <- paste("l", 1:10, sep = "")
```

Extraction de données dans un vecteur

Un seul élément

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

```
de[1]
```

```
## l1
##  2
```

```
de[[1]]
```

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

Remarquons que l'opérateur `[` préserve l'attribut nom, 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["l1"]
```

```
## l1
##  2
```

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

```
de$l1
```

```
## Error in de$l1: $ operator is invalid for atomic vectors
```

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.

- 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)]
```

```
## l3 l6 l7
##  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]
```

```
## l1 l2 l3 l4 l5
##  2  3  4  1  2
```

- extraction des observations en positions paires.

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

```
##  l2  l4  l6  l8 l10
##   3   1   3   6   4
```

- Identifiants caractères

Dans ces exemples, des chaînes de caractères contenant les noms des éléments à extraire sont utilisées.

```
de[c("l3","l6","l7")]
```

```
## l3 l6 l7
##  4  3  5
```

```
de[paste("l", seq(2, 10, by = 2), sep = "")]
```

```
##  l2  l4  l6  l8 l10
##   3   1   3   6   4
```

- 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 omettre.

```
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]
```

```
##  l1 l10
##   2   4
```

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.

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)]
```

```
##  l3  l4  l5  l6  l7  l8  l9 l10
##   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
```

```
## l1 l2 l3 l4 l5
##  2  3  4  1  2
```

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

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

```
## l1 l2 l3 l4 l5
##  2  3  4  1  2 NA NA NA NA NA
```

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

Les 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" ...
```

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
```

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"
```

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

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")
tentative
```

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

```
str(tentative)
```

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

La fonction `paste` travaille, tout comme `nchar`, de façon vectorielle. Si elle reçoit en entrée des vecteurs, elle les combine élément par élément et retourne 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"
```

La fonction `paste` permet aussi de combiner dans une seule chaîne de caractères les chaînes de caractères comprises dans un vecteur grâce à son 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"
```

La matrice

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.

Obtention d'informations sur une matrice

Observons le vecteur `mat` présenté 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 en a 4.

```
ncol(mat)
```

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

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

```
length(mat)
```

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

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

Fonctions de création d'une matrice

Fonction `matrix`

La fonction `matrix` permet de créer un objet de type matrice comme suit.

```
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).

Les 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.

```
de1 <- c(2, 3, 4, 1, 2, 3, 5, 6, 5, 4)
de2 <- c(1, 4, 2, 3, 5, 4, 6, 2, 5, 3)
```

```
des_lignes <- rbind(de1, de2)
des_lignes
```

```
##     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## de1    2    3    4    1    2    3    5    6    5     4
## de2    1    4    2    3    5    4    6    2    5     3
```

Le `r` dans le nom `rbind` indique que la fonction combine des vecteurs par lignes (en anglais *rows*), 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`.

```
des <- cbind(de1, de2)
des
```

```
##       de1 de2
##  [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
```

Fonctions de concaténation de matrices

Les 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]
## de1    2    3    4    1    2    3    5    6    5     4
## de2    1    4    2    3    5    4    6    2    5     3
## de1    2    3    4    1    2    3    5    6    5     4
## de2    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.

Ajout de métadonnées dans une matrice

Les lignes et colonnes d'une matrice peuvent être nommées.

```
rownames(des) <- paste("l", 1:10, sep = "")
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"
```

Notons que la fonction `paste0` fait la même chose que la fonction `paste` avec l'argument `sep = ""`.

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

```
attributes(des)
```

```
## $dim
## [1] 10  2
##
## $dimnames
## $dimnames[[1]]
##  [1] "l1"  "l2"  "l3"  "l4"  "l5"  "l6"  "l7"  "l8"  "l9"  "l10"
##
```

```
## $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] "l1"  "l2"  "l3"  "l4"  "l5"  "l6"  "l7"  "l8"  "l9"  "l10"
```

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

Extraction de données dans une matrice

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.

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.

Plusieurs éléments

Il est possible d'extraire des lignes complètes,

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

```
##    de1 de2
## l2   3   4
## l3   4   2
## l4   1   3
```

ou encore des colonnes.

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

```
##  l1  l2  l3  l4  l5  l6  l7  l8  l9 l10
##   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
## l1    2
## l2    3
## l3    4
## l4    1
## l5    2
## l6    3
## l7    5
## l8    6
## l9    5
## l10   4
```

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

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

```
##  l1  l2  l3  l4  l5  l6  l7  l8  l9 l10
##   2   3   4   1   2   3   5   6   5   4
```

L'array

Une matrice est en fait un cas particulier d'array, à 2 dimensions.

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

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

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

Obtention d'informations sur un array

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
```

Fonction de création d'un array

Fonction `array`

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

```r
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.

Ajout de métadonnées dans un array

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

```r
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
```

```r
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"
```

Extraction de données dans un array

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
```

La liste

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.

Obtention d'informations sur une liste

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

```
length(liste)
```

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

Fonctions de création d'une liste

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.

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
```

Fonction de concaténation de listes

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 récursifs.

Ajout de métadonnées dans une liste

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
## l1    2   1
## l2    3   4
## l3    4   2
## l4    1   3
## l5    2   5
## l6    3   4
## l7    5   6
## l8    6   2
## l9    5   5
## l10   4   3
##
## $fct
## function (x, upper = FALSE)
## if (upper) toupper(x) else tolower(x)
## <bytecode: 0x00000000140a2888>
## <environment: namespace:base>
##
## $liste
## $liste$lignes
## [1] "ligne1" "ligne2"
##
## $liste$colonnes
## [1] "colonne1" "colonne2"
##
## $liste$matrices
## [1] "matrice1" "matrice2" "matrice3"
```

Extraction d'éléments dans une liste

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"
```

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"
```

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"
```

Le data frame

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.

Jeux de données

Afin de mieux comprendre le data frame, rappelons d'abord quelques définitions relatives à un jeu de données[1].

Définitions relatives à un jeu de données

- **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.

Représentation d'un jeu de données 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
```

[1] Ces définitions sont tirées de la référence suivante : Baillargeon, S., Rivest, L.-P., Simard, M., Ghazzali, N., Mérette, C., Belisle, C., Duchesne, T., Labbe, A. et Lakhal-Chaieb, L. (2013). Analyse de tableaux de fréquences : notes de cours, STT-4400/STT-6210. Université Laval, Département de mathématiques et de statistique.

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

Obtention d'informations sur un data frame

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, la dimension est un attribut pour les matrices et les arrays, alors qu'elle 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"
```

Fonction de création d'un data frame

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.

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

```
##     de1 de2 lanceur
## l1    2   1     Luc
## l2    3   4     Luc
## l3    4   2     Luc
## l4    1   3     Luc
## l5    2   5     Luc
## l6    3   4     Kim
## l7    5   6     Kim
## l8    6   2     Kim
## l9    5   5     Kim
## l10   4   3     Kim
```

Il faut donner en entrée à la fonction `data.frame` tous les vecteurs et/ou facteurs qu'il 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(desPlus)
```

```
## '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
```

Pourquoi la troisième colonne est-elle un facteur plutôt qu'un vecteur ? Parce que par défaut la fonction `data.frame`, soit la principale fonction de création d'un data frame, transforme les vecteurs contenant des données de type caractère en facteur. Il est possible d'empêcher la fonction `data.frame` d'avoir ce comportement grâce à l'argument `stringsAsFactors` comme suit.

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

```
## '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" ...
```

Notons que la valeur par défaut de l'argument `stringsAsFactors` peut aussi être changée avec la commande.

```
options(stringsAsFactors = FALSE)
```

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

Fonctions de concaténation de data frame

Fonction `data.frame`

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

```
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(desPlus, 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
```

Fonction `cbind`

Une concaténation de data frames peut aussi s'effectuer avec la fonction `cbind`.

```
cbind(desPlus, jour)
```

```
##     de1 de2 lanceur jour
## l1    2   1     Luc    1
## l2    3   4     Luc    1
## l3    4   2     Luc    1
## l4    1   3     Luc    1
## l5    2   5     Luc    1
## l6    3   4     Kim    1
## l7    5   6     Kim    1
## l8    6   2     Kim    1
## l9    5   5     Kim    1
## l10   4   3     Kim    1
```

Ajout de métadonnées dans un data frame

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

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")
```

Extraction d'éléments d'un data frame

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.

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'indiçage introduits plus tôt fonctionnent avec le data frame, comme le démontre ces exemples.

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

```
##     de1 de2
## l1    2   1
## l2    3   4
## l3    4   2
## l4    1   3
## l5    2   5
## l6    3   4
## l7    5   6
## l8    6   2
## l9    5   5
## l10   4   3
```

```
desPlus[[1]]
```

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

```
desPlus$de1
```

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

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'indiçage `[` ou `[[` : le premier pour identifier des lignes, le deuxième pour identifier des colonnes. Voici quelques exemples.

```
desPlus[, "de1"]
```

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

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

```
##    de2 lanceur
## l7   6     Kim
## l8   2     Kim
## l9   5     Kim
```

```
desPlus[[1, 1]]
```

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

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
## l1    2   1     Luc
## l2    3   4     Luc
## l3    4   2     Luc
## l4    1   3     Luc
## l5    2   5     Luc
## l6    3   4     Kim
## l7    5   6     Kim
## l8    6   2     Kim
## l9    5   5     Kim
## l10   4   3     Kim
```

Extensions du data frame

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.

Le tibble

Les packages du `tidyverse` utilisent des tibbles en remplacment 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)
desPlus_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))
desPlus_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(desPlus_tibble)
```

```
## Classes 'tbl_df', 'tbl' 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" ...
```

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

- les colonnes contenant des données de type caractère ne sont pas transformées en facteurs par défaut (pas d'argument `stringsAsFactors`).
- 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/

Le data table

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)
desPlus_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))
desPlus_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(desPlus_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>
```

Nous remarquons tout d'abord les différences suivantes avec le data frame :

- les colonnes contenant des données de type caractère ne sont pas transformées en facteurs par défaut (valeur par défaut de l'argument `stringsAsFactors` est `FALSE`) ;
- l'affichage des noms des lignes est un peu différent.

Au-delà de ces distinctions mineures, la vraie 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 `[`.

```
desPlus_data.table[,
                   .(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://github.com/Rdatatable/data.table/wiki
- https://stt4230.rbind.io/tutoriels_etudiants/hiver_2017/data.table/

Le facteur

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).

Obtention d'informations sur un facteur.

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.

Fonction de création d'un facteur

Fonction `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"), 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"), 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.

Fonction de modification d'un facteur

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
```

Ajout de métadonnées dans un facteur

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: pomme orange poire
```

```
str(reponses)
```

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

Extraction d'éléments d'un facteur

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: pomme orange poire
```

```
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`.

```
extrait2 <- droplevels(extrait)
extrait2
```

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

```
nlevels(extrait2)
```

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

Les 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 ordinale, 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
## [5] très satisfait satisfait      satisfait
## 4 Levels: très insatisfait < insatisfait < ... < 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 ente le facteur et le facteur ordonné est utilisée par certaines fonctions R, notamment des fonctions d'ajustement de modèles.

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.

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

```
vect1 <- 1:3
str(vect1)
```

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

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

```
vect2 <- c(1,2,3)
str(vect2)
```

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

```
vect3 <- as.integer(vect2)
str(vect3)
```

```
##  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

```
vect4 <- c(1L,2L,3L)
str(vect4)
```

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

Utilité des conversions implicites de type

Nous 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`).

```r
vect5 <- c("3.5", "4.6", "7")
as.numeric(vect5)
```

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

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

```r
vect6 <- c("3,5", "4,6", "-")
as.numeric(vect6)
```

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

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

```r
type.convert(vect6, na.strings = "-", dec = ",")
```

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

Références pertinentes

- R Core Team (2016). An Introduction to R.
  https://cran.r-project.org/doc/manuals/r-release/R-intro.html
- R Core Team (2016). The R Language Definition.
  https://cran.r-project.org/doc/manuals/r-release/R-lang.html
- Wickham, H. (2014). Advanced R. CRC Press.
  - http://adv-r.had.co.nz/Data-structures.html,
  - http://adv-r.had.co.nz/Subsetting.html.