Chapitre 1

Structures de données dans R

Dans R, les éléments de base sont les objets. Tout objet en R est caractérisé par une classe et par un ou plusieurs modes. Le (ou les) mode(s) décri(ven)t le contenu d'un objet et la classe décrit sa structure. Il existe deux types possibles d'objets. Les objets simples (ou atomiques) sont des objets qui ne comportent que des éléments du même mode. Quant aux objets composés (ou hétérogènes), ce sont des objets qui comportent des éléments de mode différent (on parle aussi d'objets récursifs). On peut connaître la classe d'un objet en utilisant la fonction `class`. De plus, il est possible de tester si un objet est atomique ou récursif en utilisant respectivement les fonctions `is.atomic` et `is.recursive`. Le mode fait partie des attributs d'un objet. Un autre attribut est sa longueur qui peut être connue en appliquant la fonction `length`. Les principaux modes sont les suivants :

null (objet vide)	`NULL`
logical (booléen)	`TRUE`, `FALSE`
numeric (valeur numérique)	`1`, `2.3123`, `pi`
character (chaîne de caractères)	`'bonjour'`, `"oui"`

Il existe d'autres modes qui seront abordés dans d'autres chapitres. Les fonctions `mode` et `typeof` permettent d'afficher respectivement le mode et le type d'un objet. La fonction `attributes` permet de connaître les attributs d'un objet. Il est possible de tester si un objet est d'un mode donné ou pas. Pour cela, il faut utiliser l'une des fonctions suivantes.

`is.character`	teste si l'objet est de type chaîne de caractères
`is.integer`	teste si l'objet est de type entier
`is.logical`	teste si l'objet est de type logique
`is.null`	teste si l'objet est vide
`is.numeric`	teste si l'objet est de type flottant

Dans ce chapitre, nous allons étudier différentes structures de données : de l'élément de base que sont les vecteurs aux structures complexes que sont les tableaux de données et les listes. Grâce à des fonctions spécifiques, il est possible de faire des conversions entre

les différentes structures de données. La figure 1.1 résume les conversions possibles.

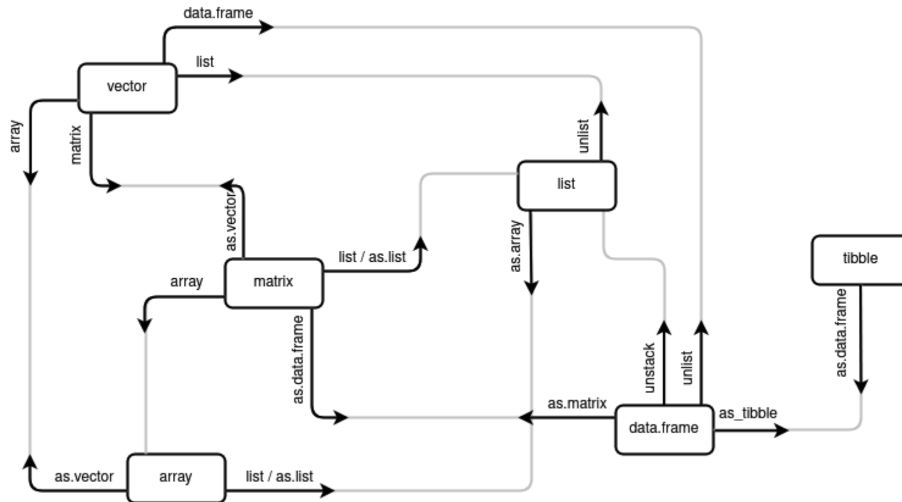

FIGURE 1.1 – Conversion entre les différentes structures de données

1.1 Vecteurs

Comme pour tout langage de programmation, le vecteur est l'élément de base pour stocker des données. Les vecteurs sont des objets homogènes, donc tous leurs éléments sont nécessairement du même type. La fonction `vector` permet de créer un vecteur d'un type donné et d'une longueur donnée. Dans l'exemple ci-dessous, on créé un vecteur de type `numeric` et de longueur 9 :

```
MonPremierObjet <- vector(mode = "numeric", length = 9)
```

Dans ce code, on voit comment stocker le résultat d'une fonction dans un objet afin de pouvoir l'utiliser plus tard. Ici, on a utilisé l'opérateur `<-`. Il existe cependant deux autres manières de faire cette affectation comme on peut le voir ci-dessous :

```
MonPremierObjet = vector(mode = "numeric", length = 9)
vector(mode = "numeric", length = 9) -> MonPremierObjet
```

Bien qu'il n'existe pas de norme pour écrire un code en R, les conventions préfèrent la première solution. Il faut noter que le symbole `=` peut avoir plusieurs sens différents : il peut être utilisé comme opérateur d'affectation, comme nous venons de le voir, mais aussi pour spécifier les options lors de l'appel à une fonction. Il n'est pas obligatoire de préciser le nom des options, mais dans ce cas il faut respecter l'ordre des options telles que définies dans la fonction :

```
MonPremierObjet <- vector("numeric", 9)
MonPremierObjet <- vector(9, "numeric")
```

Préciser le nom des options contribue à avoir un code plus lisible. De plus, on peut aussi s'affranchir de l'ordre des options dans l'appel à une fonction :

```
MonPremierObjet <- vector(length = 9, mode = "numeric")
```

Pour afficher le contenue d'un objet, on peut utiliser la fonction `print` :

```
print(MonPremierObjet)
```

La fonction `ls` permet d'afficher la liste des objets d'une session :

```
ls()
```

Cette fonction renvoie un vecteur de chaînes de caractères constitué des noms des différents objets (quelle que soit leur nature).

La fonction `rm` permet de supprimer un ou plusieurs objets d'une session :

```
rm()
```

On peut supprimer tous les objets d'une session en combinant les fonctions `rm` et `ls` de la manière suivante :

```
rm(list = ls())
```

Il est important de préciser ici que `R` fait la différence entre minuscules et majuscules (quel que soit le système d'exploitation utilisé) : par exemple, les objets `x` et `X` ne sont pas les mêmes.

Il existe deux fonctions très utiles pour générer de manière automatique des vecteurs. Ce sont les fonctions `seq` et `rep`. La fonction `seq` permet de construire un vecteur en spécifiant au moins un point de départ (paramètre `from`) et un point d'arrivée (paramètre `to`). Dans ce cas, le pas d'incrémentation sera de 1 si la valeur donnée à `from` est inférieure à celle donnée à `to` ou de -1 si la valeur donnée à `from` est inférieure à celle donnée à `to`. Si on souhaite un pas différent, il faut alors spécifier une valeur au paramètre optionnel `by`. Voici deux exemples :

```
x <- seq(from = -6, by = 2, to = 12)
x <- seq(from = -16, by = -2, to = 2)
```

De manière alternative, on peut spécifier, à l'aide de l'option `by`, la longueur de l'objet qu'on souhaite créer en utilisant le paramètre `length` :

```
x <- seq(from = -6, by = 2, length = 4)
```

Pour générer un vecteur avec un pas de 1, on peut utiliser plus simplement l'expression suivante :

```
x <- -3:2
```

Nous allons maintenant étudier la fonction `rep` qui permet de répéter plusieurs fois un vecteur pour en construire un autre. Voici un premier exemple :

```
Vecteur1 <- seq(from = -6, by = 2, length = 4)
rep(x = Vecteur1, times = 3)
```

On obtient la sortie suivante :

```
 [1] -6 -4 -2  0 -6 -4 -2  0 -6 -4 -2  0
```

Avec l'option `times` fixée à 3, on a créé un nouveau vecteur en répétant trois fois le vecteur d'entrée en les disposant à la suite. Si on utilise l'option `each` à la place de l'option `times`, alors le nouveau vecteur est construit en faisant les répétitions élément par élément. Dans l'exemple ci-dessous, on va d'abord répéter trois fois le premier, puis trois fois le second et ainsi de suite :

```
rep(x = Vecteur1, each = 3)
```

On obtient la sortie suivante :

```
 [1] -6 -6 -6 -4 -4 -4 -2 -2 -2  0  0  0
```

Il est possible de concaténer des vecteurs en utilisant la fonction `c`. On peut insérer des éléments où l'on veut : à gauche, au milieu ou à droite.

```
Vecteur1 <- -1:-3
Vecteur2 <- 1:3
c(Vecteur1, Vecteur2)
c(Vecteur2, 4, 5, 6)
c(Vecteur2, 4:6)
c(Vecteur1, 0, Vecteur2)
```

Pour manipuler un objet, il est important de pouvoir accéder à un ou plusieurs éléments d'un vecteur. Voici différentes façons de le faire pour un objet `x` :

`x[2]`	le deuxième élément d'un vecteur
`x[3:5]`	les éléments entre les positions 3 et 5
`x[c(1,2,4)]`	les éléments aux positions 1, 2 et 4
`x[-3]`	tous les éléments sauf le troisième
`x[x>3]`	les éléments supérieurs à 3
`x[x<=1|x>3]`	les éléments inférieurs à 1 ou strictement supérieurs à 3
`x[(x>=2)&(x<4)]`	les éléments supérieurs à 2 et strictement inférieurs à 4

Les trois dernières expressions dans les crochets sont des expressions logiques (nous reverrons cela plus loin). Pour les trois derniers cas, on peut obtenir les indices des éléments correspondants en utilisant la fonction `which`.

```
print(x)
index <- which((x>=2) & (x<4))
print(index)
x[index]
```

On peut s'en servir pour afficher une partie d'un vecteur, mais également pour modifier des valeurs d'un vecteur. Par exemple, dans l'exemple ci-dessous, on seuille le vecteur à zéro (c'est-à-dire que tous les éléments négatifs du vecteur sont mis à zéro) :

```
x <- seq(from = -6, by = 2, to = 12)
x[x<0] <- 0
```

Après avoir considéré le cas d'un vecteur constitué d'éléments numériques, nous allons voir le cas d'un vecteur de chaînes de caractères. La création d'un tel vecteur se fait également à l'aide de la fonction `vector` en spécifiant `mode` à `"character"` :

```
x <- vector(mode = "character", length = 3)
print(x)
```

Lorsqu'on applique la fonction `vector`, l'objet est initialisé avec des éléments qui sont des chaînes vides (avec le type `"numeric"`, tous les éléments sont à `0`). On peut

changer les éléments en y accédant (avec l'indice entre crochets) un à un. De manière plus rapide, on peut créer un vecteur à l'aide de la fonction `c` (pour combiner) :

```
x <- c("Oui","Non","Oui","Oui","Non")
```

Il existe de nombreuses fonctions permettant de manipuler les chaînes de caractères et les vecteurs de chaînes de caractères. Pour plus de détails, le lecteur pourra consulter mon précédent ouvrage[1].

1.2 Matrices et tableaux

Pour stocker des jeux de données, un vecteur (un objet de dimension 1) est très souvent insuffisant. Il est donc nécessaire de considérer des tableaux en dimension 2, 3 ou plus. Commençons par les matrices qui sont des tableaux en dimension 2. On peut créer une matrice à partir d'un vecteur à l'aide de la fonction `matrix` de la manière suivante :

```
v1 <- sample(x = 0:9, size = 20, replace = TRUE)
M <- matrix(data = v1, nrow = 4, ncol = 5)
```

La première ligne de ce code génère un vecteur de taille 20 où chacun des éléments est un nombre tiré au hasard entre 0 et 9, les tirages étant effectuées avec remise. La seconde ligne transforme ce vecteur en une matrice à 4 lignes et 5 colonnes. Par défaut, le remplissage se fait par colonne par colonne :

L'option `byrow` permet de remplir la matrice ligne par ligne :

```
matrix(data = v1, nrow = 4, ncol = 5, byrow = TRUE)
```

On obtient donc la sortie suivante :

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

Quand on effectue ce genre d'opérations, il faut être très prudent. En effet, dans les exemples ci-dessus, tout se passe bien car le produit entre le nombre de lignes et le nombre de colonnes est égale à la longueur du vecteur. Maintenant que se passe-t-il si ce n'est pas le cas ? R va produire un avertissement tout en créant un objet. Ainsi, ne produisant pas une erreur, le script va continuer à s'exécuter comme on peut le voir dans l'exemple ci-dessous :

1. Christian Paroissin. *Programmation et analyse statistique avec R*, Ellipses, juin 2015.

```
matrix (data = v1 , nrow = 4, ncol = 6)
```

L'accès aux éléments se fait de manière tout à fait similaire aux vecteurs. Voici quelques exemples :

M1[1,2]	l'élément situé sur la 1ère ligne et 2ème colonne
M1[1,3:5]	les éléments sur la 1ère ligne et entre les colonnes 3 et 5
M1[,2]	les éléments de la 2ème colonne
M1[2,]	les éléments de la 2ème ligne
M1[2,-3]	les éléments de la 2ème ligne sauf celui sur la 3ème colonne

Il est possible de concaténer des matrices que ce soit en colonne ou en ligne. Pour disposer deux matrices côte à côte, il faut utiliser la fonction `cbind` :

```
v2 <- sample(x = 0:9, size = 12, replace = TRUE)
M2 <- matrix(data = v2, nrow = 4, ncol = 3)
cbind(M,M2)
```

On obtient le résultat suivant :

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

Pour empiler deux matrices l'une au-dessous de l'autre, il faut utiliser la fonction `rbind` :

```
v3 <- sample(x = 0:9, size = 15, replace = TRUE)
M3 <- matrix(data = v3, nrow = 3, ncol = 5)
rbind(M,M3)
```

On obtient le résultat suivant :

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

Il peut être utile de considérer des tableaux tri-dimensionnels voire plus. La structure de données correspondante dans `R` est `array`. On peut créer une matrice à partir d'un

vecteur à l'aide de la fonction `array` de la manière suivante :

```
A <- array(data = v1, dim = c(2,2,5))
```

Dans le premier argument de cette fonction, on spécifie le vecteur alors que le second argument indique les différentes dimensions. On peut accéder à et extraire des éléments d'un tableau de la même manière que pour une matrice. Voici quelques exemples :

```
A[1,2,2]
A[1,1:2,2]
A[,,2]
A[,2,]
A[2,]
A[2,-2,2]
```

1.3 Listes

Les vecteurs, les matrices et les tableaux sont tous des objets homogènes, c'est-à-dire que leurs éléments sont tous de même nature. Or, en statistique, on considère souvent à la fois des variables quantitatives et des variables qualitatives. On a donc des éléments de nature différente et il est donc très utile de pouvoir disposer d'objets hétérogènes. Un tel objet dans `R` est appelé une liste et peut être construit à l'aide de la fonction `list`.

```
v1 <- sample(x = 0:9, size = 20, replace = TRUE)
v2 <- sample(x = 0:9, size = 20, replace = TRUE)
v3 <- sample(x = c(TRUE,FALSE), size = 20, replace = TRUE)
v4 <- sample(x = c("Blue","Red"), size = 20, replace = TRUE)

v <- sample(x = 0:9, size = 20, replace = TRUE)
M <- matrix(data = v, nrow = 4, ncol = 5)

L <- list(v1, v2, v3, v4, M)
print(L)
```

Une liste est donc une collection d'objets homogènes (donc de vecteurs, de matrices ou de tableaux), mais aussi éventuellement d'une liste. Pour accéder aux composantes d'une liste, on utilise des doubles crochets :

```
L[[1]]
L[[1]][1:3]
```