Introduction à R

Chapitre 1: Les concepts de base, l'organisation des données

Une première session

R est une calculatrice

R remplace aisément les fonctionnalités d'une calculatrice. Il permet aussi de faire des calculs sur des vecteurs. Voici quelques exemples très simples.

```
> 5*(-3.2) # Attention, le separateur décimal doit # être un point (.)
```

```
## [1] -16
```

```
> 5*(-3,2) # sinon, l'erreur suivante est generée:
```

```
## Error: <text>:1:6: unexpected ','
## 1: 5*(-3,
##          ^
```

```
> 5^2 # Identique a 5**2
```

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

```
> sin(2*pi/3)
```

```
## [1] 0.8660254
```

```
> sqrt (4) # Racine carree de 4.
```

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

```
> log(1) # Logarithme neperien de 1.
```

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

```
> c(1,2,3,4,5) # Crée un vecteur contenant les cinq premiers entiers
```

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

```
> c(1,2,3,4,5)*2 # Calcul des cinq premiers nombres pairs
```

```
## [1]  2  4  6  8 10
```

Tout code **R** qui suit le caractere «#» est considéré par **R** comme un commentaire. En fait, il n'est pas interprété par **R**.

Pour quitter le logiciel R, il faut taper la commande `q()`.

Il vous est ensuite proposé de sauver une image de la session. En repondant oui, les commandes tapées précédemment seront de nouveau accessibles lors d'une prochaine réouverture de **R**, au moyen des flèches «haut» et «bas» du clavier.

Stratégie de travail

- Prenez l'habitude de stocker vos fichiers dans un dossier réservé à cet usage (nomme par exemple `TravauxR`). En outre, il est conseillé de taper toutes ses instructions **R** dans une fenêtre de script appelée *script* ou *R Editor*, accessible depuis le menu «Fichier/Nouveau script». Ouvrez une nouvelle fenêtre de script puis copiez le script suivant :

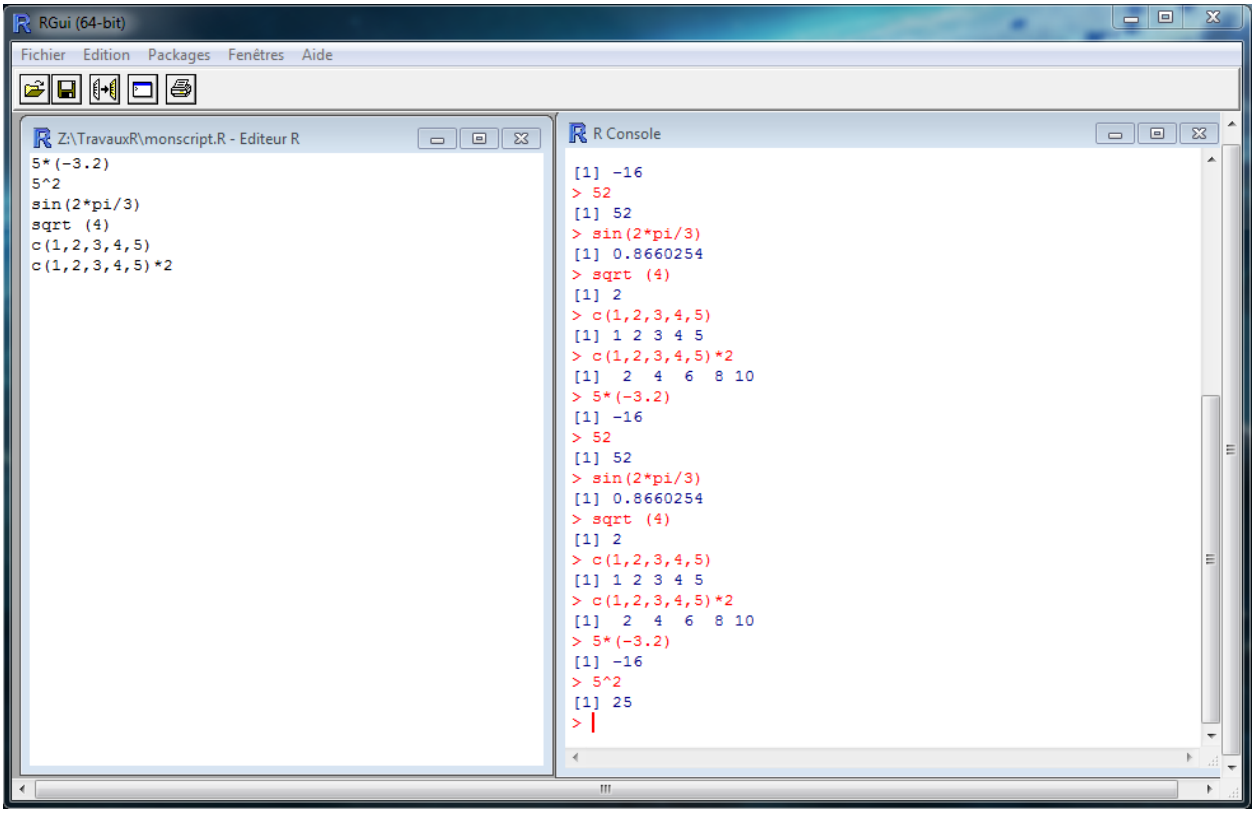

Figure 1: Vue de la fenêtre de script et de la consolde de commandes.

```
5*(-3.2)
5^2
sin(2*pi/3)
sqrt (4)
c(1,2,3,4,5)
c(1,2,3,4,5)*2
```

A la fin de la session, il est possible de sauvegarder ce script dans le dossier `TravauxR` par exemple, sous le nom `monscript.R`, et le rouvrir lors d'une session ulterieure depuis le menu «Fichier/Ouvrir un script».

Ensuite, vous pouvez taper successivement les combinaisons de touches `CTRL+A` pour sélectionner l'ensemble de ces instructions, puis `CTRL+R` pour les coller et les exécuter en une seule étape dans la console de **R**. Il est aussi possible d'exécuter une seule ligne d'instructions **R** du script en tapant `CTRL+R` lorsque le curseur clignotant se trouve sur la ligne en question dans la fenê tre de script.

Il est possible d'utiliser la fonction `source ()` depuis la console de R pour aller lire et exécuter le contenu du fichier script. Pour cela, il faut proceder ainsi :

1. cliquez une fois dans la fenetre *R Console*;
2. allez dans le menu «Fichier/Changer le répertoire courant» ;
3. explorez votre système de fichiers pour sélectionner le dossier `TravauxR`;
4. tapez dans la console: `source("monscript.R")` .

- Il faut avoir l'habitude d'utiliser le systerne d'aide en ligne de **R**. Cette aide est accessible au moyen de la fonction `help ()`. La commande `help (source)` permet par exemple d'obtenir de l'aide sur la fonction `source ()`.

Affichage des résultats et redirection dans des variables

R répond aux requêtes en affichant le résultat obtenu apres évaluation. **Ce résultat est affiché puis perdu**. Il apparaît plus intéressant de rediriger la sortie **R** d'une requête en la stockant dans une variable: cette opération s'appelle aussi **affectation du resultat dans une variable**. Une affectation évalue une expression, mais n'affiche pas le résultat qui est stocké dans un objet. Pour afficher ce résultat, il suffira de taper le nom de cet objet, suivi de la touche `ENTREE`.

Pour réaliser cette opération, on utilise la **flèche d'affectation** `<-`. La flèche `<-` s'obtient en tapant le signe inferieur (`<`) suivi du signe moins (`-`).

Pour créer un objet dans **R**, on utilise donc la syntaxe suivante : `nom.objet.a.creer <- instructions`

Par exemple

```
> x <- 1 # Affectation.
> x
```

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

On dit que `x` vaut `1`, ou que l'on affecte `1` à `x`, ou encore que l'on met dans `x` la valeur `1`. Notez que l'on peut aussi utiliser l'operation d'affectation dans l'autre sens -> de la facon suivante :

Si une commande n'est pas complète à la fin d'une ligne, **R** affichera un signe d'invite different, par defaut le signe plus (`+`), sur la deuxième ligne ainsi que sur les lignes subséquentes. **R** continuera d'attendre des instructions jusqu'à ce que la commande soit syntaxiquement complète.

Utilisation des fonctions

En plus des fonctions qu'on a déjà vu (`sin()`, `sqrt()`, `exp()` et `log()`), **R** contient de nombreuses autres fonctions dans sa version de base, et on peut en ajouter des milliers d'autres (en installant des packages, ou même en les realisant soi-même).

Une fonction dans **R** est définie par son nom et par la liste de ses paramètres. La majorité des fonctions retournent une valeur, qui peut être un nombre, un vecteur, une matrice...

L'**utilsiation** d'une fonction (on dit aussi **appeler** ou **exécuter**) se fait en tapant le nom de celle-ci, suivi, entre une paire de parenthèses, de la liste des paramètres que l'on veut utiliser. Les paramètres sont séparés par des virgules. Chacun des paramètres peut être suivi du signe `=` et de la valeur que l'on veut donner au paramètre. Cette valeur du paramètre sera appelée paramètre effectif, paramètre d'appel ou parfois paramètre d'entrée.

Exemple d'un appel d'une fonction:

`nomfonction(par1=valeur1,par2=valeur2,par3=valeur3)`

où `par1`, `par2`, ... sont appelés les paramètres de la fonction tandis que `valeur1` est la valeur que l'on donne au parametre `par1`, etc. On peut toutefois noter qu'il n'est pas forcément nécessaire d'indiquer les paramètres mais seulement leurs valeurs, pour autant que l'on respecte leur ordre. Pour toute fonction présente dans **R**, certains paramètres sont obligatoires et d'autres sont facultatifs (car une valeur par defaut est déjà fournie dans le code de la fonction).

Exemple d'utilsiation des paramètres d'une fonction:

La fonction `log(x ,base=exp(1))` admet deux paramètres : `x` et `base`.

- `x` est obligatoire, c'est le nombre dont on veut calculer le logarithme.
- `base` est un paramètre optionnel puisqu'il est suivi du signe `= et de la valeur par défaut`exp(1)‘.

Dans l'appel suivant, le calcul effectué sera celui du logarithme népérien du nombre 1 puisque la valeur de base n'est pas renseignée :

```
> log(1)
```

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

On peut appeler une fonction en jouant sur les paramètres de plusieurs façons différentes. Ainsi, pour calculer le logarithme népérien de 3, on peut utiliser n'importe laquelle des expressions suivantes.

- `log(3)`
- `log(3,base=exp(1))`
- `log(x=3)`
- `log(3 ,exp(1)))`
- `log(x=3,base=exp(1))`
- `log(base=exp(1),3)`
- `log(x=3, exp(1))`
- `log(base=exp(1),x=3)`

Les données dans R

Nature (ou type, ou mode) des données

Les fonctions `mode()` et `typeof()` permettent de gérer le type des données.

Enumérons maintenant les divers types de données (aussi appelés modes).

Type numérique (`numeric`)

Il y a deux types numériques : les entiers (`integer) et les reels` (realoudouble‘). Lorsque vous saisissez :

```
> a <- 1
> b <- 3.4
> c <- as.integer(a)
> typeof(c)
```

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

Les variables `a` et `b` sont du type `"double"` et la variable `c` a la même valeur que `a` excepté qu'elle a été forcée à être du type `"integer"`. L'intérêt est que son stockage prend moins de place en mémoire. Les instructions commençant par `as.` sont très courantes en **R** pour convertir une donnée en un type différent. Nous verrons après comment vérifier que le type d'un objet est numérique.

Type complexe (`complex`)

Type booléen ou logique (`logical`)

Données manquantes (`NA`)

Type chaînes de caractères (`character`)

Données brutes (‘raw)

Vecteurs

Un vecteur est un groupe de valeurs primitives du même type. Il peut s'agir d'un groupe de nombres, de valeurs vraies / fausses, de textes et de valeurs d'un autre type. C'est l'un des éléments constitutifs de tous les objets R. Il existe plusieurs types de vecteurs dans **R**. Ils sont distincts les uns des autres par le type d'éléments qu'ils stockent. Dans les sections suivantes, nous verrons les types de vecteurs les plus couramment utilisés, notamment les vecteurs numériques, les vecteurs logiques et les vecteurs de caractères.

Vecteur numérique

Un vecteur numérique est un vecteur de valeurs numériques. Un nombre scalaire est le vecteur numérique le plus simple. Voici un exemple de vecteur numérique:

```
> 1.5
```

```
## [1] 1.5
```

Le vecteur numérique est le type de données le plus fréquemment utilisé et constitue la base de presque tous les types d'analyse de données. Dans d'autres langages de programmation courants, il existe certains types scalaires tels que entier, double et chaîne, et ces types scalaires constituent les blocs de construction des types de conteneur tels que les vecteurs. En **R**, cependant, il n'existe pas de définition formelle des types scalaires. Un nombre scalaire n'est qu'un cas particulier de vecteur numérique, et il n'est spécial que parce que sa longueur est 1.

Lorsque nous créons une valeur, il est naturel de penser à la manière de la stocker pour une utilisation future. Pour stocker la valeur, nous pouvons utiliser `<-` pour affecter la valeur à un symbole. En d'autres termes, nous créons une variable nommée `x` de la valeur `1.5`:

```
> x <- 1.5
```

Ensuite, la valeur est assignée au symbole `x`, et nous pouvons utiliser `x` pour représenter cette valeur:

```
> x
```

```
## [1] 1.5
```

Il existe plusieurs façons de créer un vecteur numérique. Nous pouvons appeler la fonction 'numeric() pour créer un vecteur nul d'une longueur donnée:

```
> numeric(10)
```

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

Nous pouvons également utiliser `c()` pour combiner plusieurs vecteurs afin de créer un seul vecteur. Le cas le plus simple consiste, par exemple, à associer plusieurs vecteurs à un seul élément pour former un vecteur à plusieurs éléments:

```
> c(1, 2, 3, 4, 5)
```

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

Nous pouvons également combiner un mélange de vecteurs à un élément et à plusieurs éléments et obtenir un vecteur avec les mêmes éléments que ceux précédemment créés:

```
> c(1, 2, c(3, 4, 5))
```

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

Pour créer une série d'entiers consécutifs, l'opérateur `:` fera facilement l'affaire.

```
> 1:5
```

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

Une méthode plus générale pour produire une séquence numérique est `seq()`. Par exemple, le code suivant produit un vecteur numérique d'une séquence de `1` à `10` par incrément de `2`:

```
> seq(1, 10, 2)
```

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

Les fonctions comme `seq()` ont de nombreux arguments. Nous pouvons appeler une telle fonction en fournissant tous les arguments, mais ce n'est pas nécessaire dans la plupart des cas. La plupart des fonctions fournissent des valeurs par défaut raisonnables pour certains arguments, ce qui nous permet de les appeler

plus facilement. Dans ce cas, il suffit de spécifier l'argument que nous souhaitons modifier à partir de sa valeur par défaut.

Par exemple, nous pouvons créer un autre vecteur numérique qui commence par `3` avec la longueur `10` en spécifiant l'argument `length.out`:

```
> seq(3, length.out = 10)
```

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

Un appel de fonction comme ci-dessus utilise un argument nommé `length.out` afin que les autres arguments soient conservés par défaut et que seul cet argument soit modifié.

Il existe plusieurs façons de définir des vecteurs numériques, mais nous devons toujours faire attention lorsque nous utilisons `:`, un exemple:

```
> 1 + 1:5
```

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

Comme le résultat le montre, `1 + 1:5` ne signifie pas une séquence de `2` à `5`, mais de `2` à `6`. C'est parce que `:` a une priorité supérieure à `+`, ce qui conduit à évaluer d'abord `1:5` et à ajouter `1` à chaque entrée, ce qui donne la séquence que vous voyez dans le résultat.

Vecteur logique

Contrairement aux vecteurs numériques, un vecteur logique stocke un groupe de valeurs `TRUE` ou `FALSE`. Ils sont fondamentalement oui ou non pour désigner les réponses à un groupe de questions logiques.

Les vecteurs logiques les plus simples sont `TRUE` et `FALSE` eux-mêmes:

```
> TRUE
```

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

Un moyen plus habituel d'obtenir un vecteur logique consiste à poser des questions logiques sur les objets **R**. Par exemple, nous pouvons demander à **R** si `1` est supérieur à `2`:

```
> 1 > 2
```

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

La réponse est non, représentée par `FALSE`. Parfois, il est verbeux d'écrire `TRUE` et `FALSE`; nous pouvons donc utiliser `T` comme abréviation de `TRUE` et `F` pour `FALSE`. Si nous voulons effectuer plusieurs comparaisons en même temps, nous pouvons directement utiliser des vecteurs numériques dans la question:

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

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

interprète cette expression comme la comparaison élémentaire entre `c(1, 2)` et `2`. En d'autres termes, elle est équivalente à `c(1 > 2, 2 > 2)`.

Nous pouvons comparer deux vecteurs numériques multi-éléments tant que la longueur du vecteur le plus long est un multiple de celle du vecteur le plus court:

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

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

Le code précédent est équivalent à `c(1> 2, 2> 1)`. Pour montrer comment comparer deux vecteurs de longueurs différentes, voir l'exemple suivant:

```
> c(2, 3) > c(1, 2, -1, 3)
```

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

Le mécanisme informatique **recycle** le vecteur le plus court et fonctionne comme `c(2 > 1, 3 > 2, 2 > -1, 3 > 3)`. Plus spécifiquement, le vecteur le plus court sera **recyclé** pour terminer toutes les comparaisons de chaque élément du vecteur le plus long.

Dans **R**, plusieurs opérateurs logiques binaires sont définies, tels que `==` pour indiquer l'égalité, `>` pour supérieur à, `>=` pour supérieur ou égal à, `<` pour inférieur et `<=` pour inférieur ou égal à. De plus, **R** fournit d'autres opérateurs logiques supplémentaires tels que `%in%` pour indiquer si chaque élément du vecteur de gauche est contenu par le vecteur de droite:

```
>  1 %in% c(1, 2, 3)
```

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

```
> c(1, 4) %in% c(1, 2, 3)
```

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

On remarque que tous les opérateurs d'égalité effectuent le recyclage, mais que `%in%` ne le fait pas. Au lieu de cela, cela fonctionne toujours en itérant sur le vecteur à gauche et fonctionne comme `c(1 %in% c(1, 2, 3), 4 %in% c(1, 2, 3))` dans l'exemple précédent.

Vecteur de caractère

Un vecteur de caractère est un groupe de chaînes de caractères. Ici, un caractère ne signifie pas littéralement une lettre ou un symbole dans une langue, mais une chaîne comme `elle-ci est une chaîne`. Les guillemets doubles et les guillemets simples peuvent être utilisés pour créer un vecteur de caractère, comme suit:

```
> "hello, world!"
```

```
## [1] "hello, world!"
```

```
> 'hello, world!'
```

```
## [1] "hello, world!"
```

Nous pouvons également utiliser la fonction de combinaison `c()` pour construire un vecteur de caractères multi-éléments:

```
> c("Hello", "World")
```

```
## [1] "Hello" "World"
```

Nous pouvons utiliser `==` pour dire si deux vecteurs ont des valeurs égales dans les positions correspondantes; cela s'applique aussi aux vecteurs de caractères:

```
> c("Hello", "World") == c('Hello', 'World')
```

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

Les vecteurs de caractères sont égaux, car `"` et `'` fonctionnent ensemble pour créer une chaîne et n'affectent pas sa valeur:

```
> c("Hello", "World") == "Hello, World"
```

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

L'expression précédente donne `FALSE` car ni `Hello` ni `World` ne valent `Hello, World`. La seule différence entre les deux guillemets est le comportement lorsque vous créez une chaîne contenant des guillemets.

Si vous utilisez `"` pour créer une chaîne (un vecteur de caractères à un seul élément) contenant `"` lui-même, vous devez taper `\"` pour échapper `"` dans la chaîne pour empêcher l'interpréteur de considérer `"` dans la chaîne comme le guillemet de la chaîne.

Les exemples suivants illustrent l'échappement des guillemets. Le code utilise `cat()` pour imprimer le texte donné:

```
> cat("Bonjour \"Salah\" comment tu vas?")
```

```
## Bonjour "Salah" comment tu vas?
```

Si vous pensez que ce n'est pas facile à lire, vous pouvez utiliser ' pour créer la chaîne, ce qui peut être plus facile:

```
> cat('Bonjour "Salah" comment tu vas??')
```

```
## Bonjour "Salah" comment tu vas??
```

En d'autres termes, " permet " dans la chaîne sans échapper et " permet " dans la chaîne sans échapper.

Nous connaissons maintenant les bases de la création de vecteurs numériques, de vecteurs logiques et de vecteurs de caractères. En fait, nous avons aussi des vecteurs complexes et des vecteurs bruts dans **R**. Les vecteurs complexes sont des vecteurs de valeurs complexes, tels que `c(1 + 2i, 2 + 3i)`. Les vecteurs bruts stockent essentiellement des données binaires brutes représentées sous la forme hexadécimale. Ces deux types de vecteurs sont beaucoup moins utilisés, mais ils partagent de nombreux comportements avec les trois types de vecteurs que nous avons abordés.

Dans la section suivante, vous apprendrez plusieurs manières d'accéder à une partie d'un vecteur.

Sous-vecteurs

Si nous voulons accéder à des entrées spécifiques ou à un sous-ensemble d'un vecteur, sous-définir un vecteur signifie accéder à des entrées spécifiques ou à un sous-ensemble du vecteur. Dans cette section, nous allons montrer différentes manières de sous-définir un vecteur.

Tout d'abord, nous créons un vecteur numérique simple et l'attribuons à 'v1:

```
> v1 <- c(1, 2, 3, 4)
```

Chacune des lignes suivantes obtient un sous-ensemble spécifique de `v1`. Par exemple, nous pouvons obtenir le deuxième élément:

```
> v1[2]
```

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

Nous pouvons obtenir les deuxième à quatrième éléments:

```
> v1[2:4]
```

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

Nous pouvons obtenir tous les éléments sauf le troisième:

```
> v1[-3]
```

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

Les modèles sont clairs: nous pouvons placer n'importe quel vecteur numérique entre les crochets après le vecteur pour extraire un sous-ensemble correspondant:

```
> a <- c(1, 3)
> v1[a]
```

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

Tous les exemples précédents effectuent des sous-ensembles par position, c'est-à-dire que nous obtenons un sous-ensemble d'un vecteur en spécifiant les positions des éléments. L'utilisation de nombres négatifs exclura

ces éléments. Une chose à noter est que vous ne pouvez pas utiliser des nombres positifs et des nombres négatifs ensemble:

```
> v1[c(1, 2, -3)]
```

```
## Error in v1[c(1, 2, -3)]: only 0's may be mixed with negative subscripts
```

Que se passe-t-il si nous subdivisons le vecteur en utilisant des positions situées au-delà de la plage du vecteur? L'exemple suivant tente d'obtenir un sous-ensemble de v1 du troisième élément au sixième élément non existant:

```
> v1[3: 6]
```

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

Comme on peut le constater, les positions non existantes se retrouvent avec les valeurs manquantes représentées par NA. Dans les données du monde réel, les valeurs manquantes sont courantes. La bonne partie est que tous les calculs arithmétiques avec NA aboutissent également à NA pour la cohérence. D'autre part, toutefois, le traitement des données nécessite un effort supplémentaire, car il peut être dangereux de supposer que les données ne contiennent aucune valeur manquante.

Un autre moyen de créer un vecteur consiste à utiliser des vecteurs logiques. Nous pouvons fournir un vecteur logique de longueur égale pour déterminer si chaque entrée doit être extraite:

```
> v1[c(TRUE, FALSE, TRUE, FALSE)]
```

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

Plus qu'un sous-ensemble, nous pouvons écraser un sous-ensemble spécifique d'un vecteur comme ceci:

```
> v1[2] <- 0
```

Dans ce cas, `v1` devient le suivant:

```
> v1
```

```
## [1] 1 0 3 4
```

Nous pouvons également écraser plusieurs éléments à différentes positions en même temps:

```
> v1[2: 4] <- c(0, 1, 3)
```

Maintenant, `v1` devient le suivant:

```
> v1
```

```
## [1] 1 0 1 3
```

Comme les sous-ensembles, les sélecteurs logiques sont également acceptés pour la réécriture:

```
> v1[c(TRUE, FALSE, TRUE, FALSE)] <- c(3, 2)
```

Comme vous pouvez vous en douter, `v1` devient le suivant:

```
> v1
```

```
## [1] 3 0 2 3
```

Une implication utile de cette opération est la sélection d'entrées par critère logique. Par exemple, le code suivant sélectionne tous les éléments qui ne sont pas supérieurs à `2` dans `v1`:

```
> v1[v1 <= 2]
```

```
## [1] 0 2
```

Un critère de sélection plus complexe fonctionne également. L'exemple suivant sélectionne tous les éléments de `v1` qui satisfont `x^2 - x + 1 > 0`:

```
> v1[v1 ^ 2 - v1 + 1 >= 0]
```

```
## [1] 3 0 2 3
```

Pour remplacer toutes les entrées qui satisfont `x <= 2` par `0`, nous pouvons appeler:

```
> v1[v1 <= 2] <- 0
```

Comme vous vous en doutez, v1 devient le suivant:

```
> v1
```

```
## [1] 3 0 0 3
```

Si nous écrasons le vecteur sur une entrée inexistante, le vecteur se développera automatiquement avec la valeur non attribuée étant NA comme valeurs manquantes:

```
> v1[10] <- 8
> v1
```

```
##  [1]  3  0  0  3 NA NA NA NA NA  8
```

Chapitre 2: Importation-exportation et production de données

Importer des données

Importer les données depuis un fichier texte ASCII

Il existe trois fonctions **R** principales à utiliser pour importer des données depuis un fichier texte: * `read.table()`: pour des jeux de données organisés sous la forme de tableaux, comme cela est souvent le cas en statistique. * `read.ftable()`: pour lire des tableaux de contingence. * `scan()`: beaucoup plus flexible et puissante à utiliser dans tous les autres cas.

Lecture de données avec `read.table()`

L'instruction **R** suivante va lire les données presentes dans un fichier (à sélectionner dans une fenetre de dialogue) et les rapatrier dans **R** sous la forme d'un `data.frame` que nous avons choisi de nommer `donnees`.

```
donnees <- read.table(file=file.choose(),header=T,sep="\t", dec=".",row.names=1)
```

Les paramètres principaux de `read.table()`

Lecture de données avec `read.ftable()`

Lecture de données avec la fonction `scan()`

Importer des données depuis Excel au le tableur d'Open Office

Utiliser le copier-coller

Passer par un fichier ASCII intermédiaire

Utiliser des packages spécifiques

Importer des données depuis SPSS, Minitab, SAS ou Matlab

Les gros fichiers de données