

Programmation statistique avec R

Les bases du langage

J. R. Lobry

Université Claude Bernard Lyon I – France

Biologie & Modélisation 2007-2008 (saison 2)

Table des matières

- 1 Objets
- 2 Fonctions
- 3 Vecteurs
- 4 Matrices et tableaux
- 5 Facteurs
- 6 Listes
- 7 Data.frame
- 8 Structures de contrôle

Table des matières

- 1 Objets
- 2 Fonctions
- 3 Vecteurs
- 4 Matrices et tableaux
- 5 Facteurs
- 6 Listes
- 7 Data.frame
- 8 Structures de contrôle

Plan détaillé

1 Objets

- Exemples avec des valeurs numériques
- Classes d'objets
- Classes et fonctions génériques

Objet \leftarrow Expression

 fonctionne en évaluant les **expressions** entrées dans la fenêtre de commande :

```
2 + 2
```

```
[1] 4
```

On peut ranger la valeur d'une expression dans un **objet** pour son utilisation ultérieure :

```
x <- 2 + 2  
10 * x
```

```
[1] 40
```

Division par zéro

La division d'un nombre non nul par zéro donne Inf ou -Inf en fonction du signe du dénominateur :

1/0

[1] Inf

-1/0

[1] -Inf


La division de zéro par zéro n'est pas définie :

0/0

[1] NaN

NaN pour Not a Number.

NaN

 R sait gérer les indéterminations :

```
Inf + 1
```

```
[1] Inf
```

```
1/Inf
```

```
[1] 0
```

```
Inf/0
```

```
[1] Inf
```

```
Inf - Inf
```

```
[1] NaN
```

```
Inf/Inf
```

```
[1] NaN
```

NaN

Les indéterminations se propagent :

```
log(-1)
```

```
[1] NaN
```


```
log(-1) + 2
```

```
[1] NaN
```

```
5 * log(-1)
```

```
[1] NaN
```


NA

 sait gérer les données manquantes :

```
NA + 3
```

```
[1] NA
```

```
4 * NA
```

```
[1] NA
```

NA pour **N**ot **A**vailable.

Fonctions de test associées

```
is.finite(3.1415)
```

```
[1] TRUE
```

```
is.infinite(Inf)
```

```
[1] TRUE
```

```
is.nan(NaN)
```

```
[1] TRUE
```

```
is.na(NA)
```

```
[1] TRUE
```

Plan détaillé

1 Objets


- Exemples avec des valeurs numériques
- Classes d'objets
- Classes et fonctions génériques

Classes

- Nous avons vu des exemples d'objets contenant des valeurs numériques. Ces objets sont des instances de classe `numeric` :


```
x <- 2 + 2  
class(x)
```

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

-  a plusieurs classes d'objets importants que nous allons détailler ci-après. Citons par exemple : `functions`, `vectors` (numeric, character, logical), `matrices`, `lists` et `data.frames`.

Que peut on mettre dans un objet ?

Les objets peuvent contenir de nombreuses choses différentes, par exemple :

- Des constantes : 2, 13.005, "January"
- Des symboles spéciaux : NA, TRUE, FALSE, NULL, NaN
- Des choses déjà définies dans  : seq, c (function), month.name, letters (character), pi (numeric)
- De nouveaux objets créés en utilisant des objets existants (ceci est fait en évaluant des *expressions* — e.g., 1 / sin(seq(0, pi, length = 50)))
- Etc.

Constructeurs

Souvent, les objets d'une classe particulière sont créés avec une fonction ayant le même nom que la classe. On peut ainsi créer des objets vides.

```
numeric(5)
```

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

```
complex(3)
```

```
[1] 0+0i 0+0i 0+0i
```

```
logical(4)
```

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

```
character(5)
```

```
[1] "" "" "" "" ""
```

mode() et class()

Il y a de nombreux types d'objets dans . Pour un objet donné :

- `mode(object)` nous renseigne sur la façon dont l'objet est stocké, ce n'est pas très utile en pratique.
- `class(object)` nous donne la **classe** d'un objet. Le principal intérêt de la classe est que les fonctions génériques (comme `print()` ou `plot()`) sachent quoi faire avec.

```
mode(pi)
```

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

```
class(pi)
```

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

Plan détaillé

1 Objets

- Exemples avec des valeurs numériques
- Classes d'objets
- Classes et fonctions génériques

Choix automatique de la bonne fonction

Le mode d'un objet nous dit comment il est stocké. Deux objets peuvent être stockés de la même manière mais avoir une classe différente. La façon dont les objets seront affichés est déterminée par la classe, et non pas par le mode.

```
x <- 1:12
class(x)
```

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

```
y <- matrix(1:12, nrow = 2, ncol = 6)
class(y)
```

```
[1] "matrix"
```

```
print(x)
```

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

```
print(y)
```

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

Des méthodes aux classes

Pour connaître les spécialisations de la méthode générique `print()` on utilise la fonction `methods()`. Nous n'avons listé ici que le 20 premières !

```
methods("print")[1:20]
```

```
[1] "print.acf"  
[2] "print.anova"  
[3] "print.aov"  
[4] "print.aovlist"  
[5] "print.ar"  
[6] "print.Arima"  
[7] "print.arima0"  
[8] "print.AsIs"  
[9] "print.aspell"  
[10] "print.Bibtex"  
[11] "print.browseVignettes"  
[12] "print.by"  
[13] "print.check_code_usage_in_package"  
[14] "print.check_demo_index"  
[15] "print.check_dotInternal"  
[16] "print.check_make_vars"  
[17] "print.check_package_code_syntax"  
[18] "print.check_package_CRAN_incoming"  
[19] "print.check_package_datasets"
```

Des classes aux méthodes

Pour connaître les méthodes spécialisées pour une classe, ici la classe `data.frame`, on utilise encore la fonction `methods()` :

```
methods(class = "data.frame")[1:20]
```

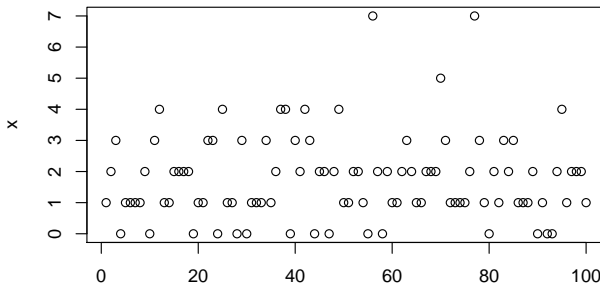
```
[1] "[.data.frame"           "[[.data.frame"
[3] "[[<-.data.frame"       "[<-.data.frame"
[5] "$<-.data.frame"        "aggregate.data.frame"
[7] "anyDuplicated.data.frame" "as.data.frame.data.frame"
[9] "as.list.data.frame"     "as.matrix.data.frame"
[11] "by.data.frame"         "cbind.data.frame"
[13] "dim.data.frame"        "dimnames.data.frame"
[15] "dimnames<-.data.frame" "duplicated.data.frame"
[17] "edit.data.frame"       "format.data.frame"
[19] "formula.data.frame"    "head.data.frame"
```

Intérêt des méthodes génériques

Une fonction peut donc faire des choses très différentes en fonction de la classe des objets. Donnons un exemple graphique simple :

```
x <- rpois(100, lambda = 2)
class(x)
[1] "numeric"
plot(x, main = paste("plot pour la classe", class(x)))
```

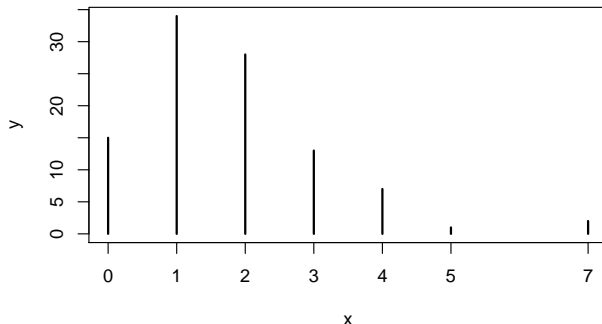
plot pour la classe numeric



Intérêt des méthodes génériques

```
y <- table(x)
class(y)
[1] "table"
plot(y, main = paste("plot pour la classe", class(y)))
```

plot pour la classe table



Intérêt des méthodes génériques

```
z <- hist(x, plot = FALSE)
class(z)
[1] "histogram"
plot(z, main = paste("plot pour la classe", class(z)),
     col = grey(0.7))
```



Méthode par défaut

S'il n'existe pas de méthode spécialisée pour la classe d'objet, on utilise la méthode par défaut :

```
methods(class = "default")[1:20]
```

```
[1] "add1.default"           "aggregate.default"  
[3] "AIC.default"            "all.equal.default"  
[5] "ansari.test.default"    "anyDuplicated.default"  
[7] "ar.burg.default"        "ar.yw.default"  
[9] "as.array.default"       "as.character.default"  
[11] "as.data.frame.default"  "as.Date.default"  
[13] "as.dist.default"        "as.expression.default"  
[15] "as.function.default"    "as.hclust.default"  
[17] "as.list.default"        "as.matrix.default"  
[19] "as.null.default"        "as.person.default"
```

Table des matières

- 1 Objets
- 2 Fonctions**
- 3 Vecteurs
- 4 Matrices et tableaux
- 5 Facteurs
- 6 Listes
- 7 Data.frame
- 8 Structures de contrôle

Les fonctions

```
library(fortunes)  
fortune(52)
```

```
Can one be a good data analyst without being a half-good programmer?  
The short answer to that is, 'No.' The long answer to that is, 'No.'  
-- Frank Harrell  
    1999 S-PLUS User Conference, New Orleans (October 1999)
```

Plan détaillé

2 Fonctions

- Blocs
- Arguments des fonctions
- Identification par position
- Identification par nom
- Identification avec la valeur par défaut
- Définir ses propres fonctions
- L'argument spécial point-point-point
- Neutralisation de l'affichage automatique
- Portée des variable

Les blocs entre parenthèses

Avant de passer aux fonctions, nous avons besoin de préciser quelques points sur les **expressions**.

Les expressions sont évaluées pour créer des objets. Elles sont constituées d'opérateurs (+, *, ^) et d'autres objets (variables ou constantes) par exemple : $2 + 2$, $\sin(x)^2$.


```
a <- 2 + 2  
a
```

```
[1] 4
```

```
b <- sin(a)^2  
b
```

```
[1] 0.57275
```

Expression et objets : blocs

-  a des **blocs d'expressions** qui sont une suite d'expressions encadrées par des accolades.
- Toutes les expressions d'un bloc sont évaluées les unes à la suite des autres. Tous les assignements de variables seront effectifs et tous les appels à `print()` ou `plot()` auront les effets collatéraux attendus.
- Mais le plus important est que le bloc entier est lui-même une expression dont la valeur sera **la dernière expression évaluée dans le bloc**.

Expression et objets : blocs

```
monbloc <- {  
  tmp <- 1:10  
  somme <- sum(tmp)
```

```
}
```

```
tmp
```

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

```
somme
```

```
[1] 55
```

```
monbloc
```


```
[1] 55
```

Plan détaillé

2 Fonctions

- Blocs
- Arguments des fonctions
 - Identification par position
 - Identification par nom
 - Identification avec la valeur par défaut
 - Définir ses propres fonctions
 - L'argument spécial point-point-point
 - Neutralisation de l'affichage automatique
 - Portée des variable

Souplesse de l'identification arguments des fonctions

- La plupart des choses utiles dans  sont faites en appelant des fonctions. Les appels de fonctions ressemblent à un nom suivi **d'arguments** entre parenthèses.
- Les arguments des fonctions peuvent être très nombreux.
- L'identification de la valeur des arguments peut se faire de plusieurs manières de façon à faciliter la vie de l'utilisateur.
- C'est une spécificité du langage S inspirée de l'assembleur JCL des IBM 360.

Souplesse de l'identification arguments des fonctions

À part un argument un peu particulier appelé "...", tous les arguments ont un **nom formel**. Par exemple, les deux premiers arguments de la fonction `plot.default` ont pour nom formel **x** et **y** :

```
args(plot.default)

function (x, y = NULL, type = "p", xlim = NULL, ylim = NULL,
  log = "", main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
  ann = par("ann"), axes = TRUE, frame.plot = axes, panel.first = NULL,
  panel.last = NULL, asp = NA, ...)
NULL
```

Quand une fonction est appelée, elle a besoin de savoir quelles valeurs ont été données à quels arguments. Cette identification de la valeur des arguments peut se faire de plusieurs manières :

- Par position.
- Par nom (éventuellement abrégé).
- Par valeur par défaut.

Plan détaillé

2 Fonctions

- Blocs
- Arguments des fonctions
- **Identification par position**
- Identification par nom
- Identification avec la valeur par défaut
- Définir ses propres fonctions
- L'argument spécial point-point-point
- Neutralisation de l'affichage automatique
- Portée des variable

Identification par position

C'est la notation la plus compacte : on donne les arguments dans le même ordre que celui de la fonction.

```
args(plot.default)
```

```
function (x, y = NULL, type = "p", xlim = NULL, ylim = NULL,  
  log = "", main = NULL, sub = NULL, xlab = NULL, ylab = NULL,  
  ann = par("ann"), axes = TRUE, frame.plot = axes, panel.first = NULL,  
  panel.last = NULL, asp = NA, ...)  
NULL
```

Les deux premiers arguments de la fonction `plot()` sont `x` et `y`. Ainsi, `plot(ceci, cela)` est équivalent à `plot(x = ceci, y = cela)`.

Plan détaillé

2 Fonctions

- Blocs
- Arguments des fonctions
- Identification par position
- **Identification par nom**
- Identification avec la valeur par défaut
- Définir ses propres fonctions
- L'argument spécial point-point-point
- Neutralisation de l'affichage automatique
- Portée des variable

Identification par nom

C'est la façon la plus sûre et la plus souple de contrôler la valeur des arguments, en spécifiant leur nom de manière explicite. Ceci neutralise l'identification par position, de sorte que nous pouvons écrire `plot(y = cela, x = ceci)` et obtenir le même résultat. Les noms formels peuvent être abrégés tant que :

- l'abréviation n'est pas ambiguë.
- l'argument n'est pas après "..."

Identification par nom (sans arguments après ...)

```
args(seq.default)
```

```
function (from = 1, to = 1, by = ((to - from)/(length.out - 1)),  
  length.out = NULL, along.with = NULL, ...)  
NULL
```

```
seq(from = 1, to = 5, by = 1)
```

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

```
seq(f = 1, t = 5, b = 1)
```

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

```
seq(from = 1, to = 10, length.out = 5)
```

```
[1] 1.00 3.25 5.50 7.75 10.00
```

```
seq(1, 10, l = 5)
```

```
[1] 1.00 3.25 5.50 7.75 10.00
```

```
seq(1, 10, 5)
```

```
[1] 1 6
```

Identification par nom (avec arguments après ...)

Il y a une exception à la possibilité d'abréviation des arguments des fonctions : c'est quand un argument formel est après l'argument ..., par exemple

```
args(paste)

function (... , sep = " ", collapse = NULL)
NULL

letters[1:10]
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"

paste(letters[1:10], collapse = "")
[1] "abcdefghij"

paste(letters[1:10], col = "")
[1] "a " "b " "c " "d " "e " "f " "g " "h " "i " "j "
```

Dans le dernier cas, l'argument `col = ""` est absorbé dans l'argument `...` c'est donc une chaîne vide qui va être recyclée pour être collée à tous les éléments de `letters` avec la valeur par défaut pour `sep`.

Plan détaillé

2 Fonctions

- Blocs
- Arguments des fonctions
- Identification par position
- Identification par nom
- **Identification avec la valeur par défaut**
- Définir ses propres fonctions
- L'argument spécial point-point-point
- Neutralisation de l'affichage automatique
- Portée des variable

Utilisation de valeurs par défaut

Les arguments ont souvent des valeurs par défaut. Dans ce cas, si rien n'est précisé au moment de l'appel de la fonction, ce sont ces valeurs par défaut qui seront utilisées.

```
args(paste)
```

```
function (... , sep = " ", collapse = NULL)  
NULL
```

Par exemple, si rien n'est précisé pour l'argument `sep` de la fonction `paste()` on utilisera la chaîne de caractère " ", c'est à dire un espace.

Plan détaillé

2 Fonctions

- Blocs
- Arguments des fonctions
- Identification par position
- Identification par nom
- Identification avec la valeur par défaut
- Définir ses propres fonctions
- L'argument spécial point-point-point
- Neutralisation de l'affichage automatique
- Portée des variable

Les fonctions : création

Une nouvelle fonction est créée par une construction de la forme :
`fun.name <- function(arglist) expr` avec :

- **fun.name** : le nom d'un objet où est stockée la fonction.
- **arglist** : une liste d'arguments formels. Elle peut
 - être vide (dans ce cas la fonction n'a pas d'arguments)
 - avoir quelques noms formels séparés par des virgules.
 - avoir des arguments de la forme `nom = valeur` pour donner une valeur par défaut à l'argument.
- **expr** : typiquement un *bloc d'expressions*.

Dans la fonction il peut y avoir un appel à `return(val)` qui arrête la fonction et renvoie la valeur `val`. Si rien n'est précisé la fonction renvoie la valeur de la dernière expression du bloc.

La fonction "Hello world"

Définition de la fonction :

```
hello <- function() {  
  print("Hello world")  
}
```

Appel de la fonction :

```
hello()  
[1] "Hello world"
```

Une fonction qui retourne ses arguments

```
mafonction <- function(a = 1, b = 2, c) {  
  resultat <- c(a, b, c)  
  names(resultat) <- c("a", "b", "c")  
  return(resultat)  
}  
mafonction(6, 7, 8)
```

```
a b c  
6 7 8
```

```
mafonction(10, c = "string")
```

```
      a      b      c  
"10"  "2"  "string"
```

La fonction args()

Pour une fonction donnée, la liste de ses arguments (avec les valeurs par défaut éventuelles) est donnée par la fonction `args()` :

```
args(mafonction)
```

```
function (a = 1, b = 2, c)  
NULL
```

```
args(plot.default)
```

```
function (x, y = NULL, type = "p", xlim = NULL, ylim = NULL,  
  log = "", main = NULL, sub = NULL, xlab = NULL, ylab = NULL,  
  ann = par("ann"), axes = TRUE, frame.plot = axes, panel.first = NULL,  
  panel.last = NULL, asp = NA, ...)  
NULL
```

Pour les fonctions qui nous sont assez familières, la simple consultation de la liste des arguments remplace parfois avantageusement la lecture de la documentation.

La fonction `body()`

Pour une fonction donnée, le corps de la fonction est donnée par la fonction `body` :

```
body(mafonction)

{
  resultat <- c(a, b, c)
  names(resultat) <- c("a", "b", "c")
  return(resultat)
}
```

On peut aussi entrer le nom de la fonction sans les parenthèses pour avoir `args()`+`body()` :

```
mafonction

function (a = 1, b = 2, c)
{
  resultat <- c(a, b, c)
  names(resultat) <- c("a", "b", "c")
  return(resultat)
}
```

Plan détaillé

2 Fonctions

- Blocs
- Arguments des fonctions
- Identification par position
- Identification par nom
- Identification avec la valeur par défaut
- Définir ses propres fonctions
- L'argument spécial point-point-point
- Neutralisation de l'affichage automatique
- Portée des variable

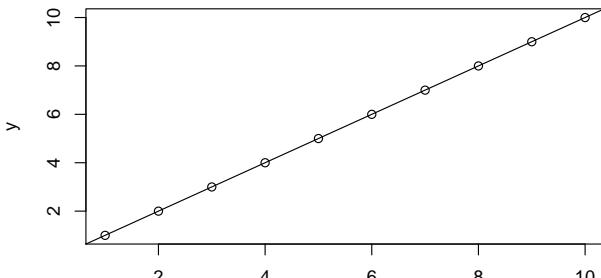
point-point-point ...

- L'argument point-point-point permet à une fonction d'accepter un nombre quelconque d'arguments.
- L'argument point-point-point (...) indique que la fonction accepte n'importe quoi d'autre comme argument. Ce qu'il adviendra de ces arguments est déterminé par la fonction. En général ils sont transmis à d'autres fonctions. Par exemple, une fonction graphique de haut niveau transmettra l'argument point-point-point à des fonctions graphiques de bas niveau pour traitement.

Intérêt de l'argument point-point-point

Supposons que nous voulions définir une fonction qui dessine un nuage de points et y ajoute une droite de régression :

```
f <- function(x, y, ...) {  
  plot.default(x = x, y = y, ...)  
  abline(coef = lm(y ~ x)$coef)  
}  
f(1:10, 1:10)
```

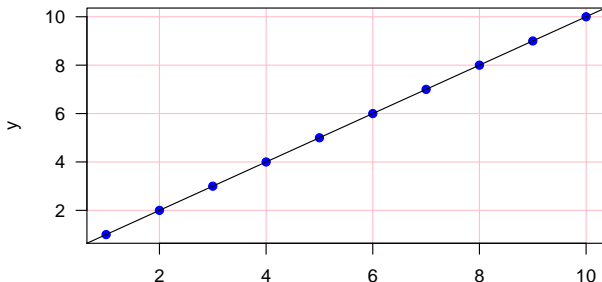


Intérêt de l'argument point-point-point

Comme nous avons transmis l'argument point-point-point à `plot.default()`, tout ce que `plot.default()` sait faire, notre fonction `f()` sait le faire également.

```
f(1:10, 1:10, main = "Titre de ce graphique", pch = 19,  
  col = "blue", las = 1, panel.first = grid(col = "pink",  
  lty = "solid"))
```

Titre de ce graphique




Plan détaillé


2 Fonctions

- Blocs
- Arguments des fonctions
- Identification par position
- Identification par nom
- Identification avec la valeur par défaut
- Définir ses propres fonctions
- L'argument spécial point-point-point
- **Neutralisation de l'affichage automatique**
- Portée des variable

L'affichage automatique

Au niveau de la console de commande , les objets sont affichés automatiquement quand on donne leur nom sans les affecter à une variable.

```
tmp <- "hello"  
tmp  
[1] "hello"
```

- C'est comme si on avait fait `print()` sur cet objet
- Ceci n'arrive pas systématiquement,  a la possibilité de désactiver cet affichage automatique.
- L'affichage automatique ne se produit **jamais** à l'intérieur d'une fonction.
- Cela peut parfois conduire à des comportements surprenants. Utilisez toujours un appel explicite à la fonction `print()` si vous voulez que votre fonction affiche la valeur d'un objet.

L'affichage automatique

Illustration de la neutralisation de l'affichage automatique :

```
f1 <- function(x = pi) {  
  x  
  0  
}  
f1()
```

```
[1] 0
```

```
f2 <- function(x = pi) {  
  print(x)  
  0  
}  
f2()
```

```
[1] 3.141593
```


```
[1] 0
```

Plan détaillé

2 Fonctions

- Blocs
- Arguments des fonctions
- Identification par position
- Identification par nom
- Identification avec la valeur par défaut
- Définir ses propres fonctions
- L'argument spécial point-point-point
- Neutralisation de l'affichage automatique
- Portée des variable

Portée des variable

- Quand une expression fait appel à une variable, comment la valeur de cette variable est déterminée ?
-  utilise ce que l'on appelle la portée lexicale des variables, inspirée du langage Scheme.
- Ce choix permet de simplifier l'écriture des fonctions.

Portée des variable

- À l'intérieur d'une fonction la variable est d'abord recherchée à l'intérieur de la fonction, à savoir :
 - Les variables définies comme arguments de cette fonction
 - Les variables définie à l'intérieur de la fonction
- Si une variable n'est pas trouvée à l'intérieur de la fonction, elle est recherchée *en dehors* de la fonction. Le détail de ce mécanisme est complexe, mais tout ce que l'on a à retenir est que :
 - une variable définie à l'extérieur de la fonction est accessible aussi dans la fonction. Si deux variables avec le même nom sont définies à l'intérieur et à l'extérieur de la fonction, c'est la variable *locale* qui sera utilisée.
- Une erreur aura lieu si aucune variable avec le nom demandé n'est trouvée.

Illustration de la portée des variable

```
mvariable <- 1
mafonction1 <- function() {
  mvariable <- 5
  print(mvariable)
}
mafonction1()
```

```
[1] 5
```


```
mvariable
```

```
[1] 1
```

```
mafonction2 <- function() {
  print(mvariable)
}
mafonction2()
```

```
[1] 1
```

Illustration de la portée des variable

La portée lexicale des variables dans  permet de simplifier l'écriture des fonctions emboîtées en autorisant des constructions du type :

```
cube <- function(n) {  
  carre <- function() n * n  
  n * carre()  
}  
cube(2)
```

```
[1] 8
```

Dans la plupart des langages de programmation, la fonction `carre` ne peut pas être définie ainsi parce que `n` est inconnue (sauf à définir `n` globalement).

Table des matières

- 1 Objets
- 2 Fonctions
- 3 Vecteurs**
- 4 Matrices et tableaux
- 5 Facteurs
- 6 Listes
- 7 Data.frame
- 8 Structures de contrôle

Vecteurs

```
fortune(75)
```

```
Eric Lecoutre: I don't want to die being idiot...
```

```
Peter Dalgaard: With age, most of us come to realise that that is the  
only possible outcome.
```

```
-- Eric Lecoutre and Peter Dalgaard
```


```
  R-help (October 2004)
```

Plan détaillé

3 Vecteurs

- Exemples de vecteurs
- Fonctions de création de vecteurs
- Indexation des vecteurs
- Indexation par des entiers positifs
- Indexation par des entiers négatifs
- Indexation par un vecteur logique
- Indexation par un nom

Tout est vecteur dans R

Les types de données les plus élémentaires dans  sont déjà des vecteurs. Les formes les plus simples sont **numeric**, **character** et **logical** (TRUE ou FALSE) :

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

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

```
c("toto", "titi", "tata")
```

```
[1] "toto" "titi" "tata"
```

```
c(T, T, F, T)
```

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

NB : T et F sont des abréviations valides pour TRUE et FALSE, respectivement.

Longueur d'un vecteur

La longueur des vecteurs est donnée par la fonction `length()`, c'est le nombre d'éléments du vecteur :

```
x <- c(1, 2, 3, 4, 5)
length(x)
```

```
[1] 5
```

```
letters
```

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

```
length(letters)
```

```
[1] 26
```

Valeurs particulières

- NA Valeur manquante (Not Available)
- NaN Pas un nombre (Not a Number', e.g., 0/0)
- -Inf, Inf Infini positif ou négatif, e.g. 1/0
- NULL Objet nul, utilisé surtout en programmation

La bonne gestion des valeurs manquantes est une spécificité des bons logiciels de statistiques. Attention, NA n'est pas un NaN, ni une chaîne de caractères :

```
is.na(c(1, NA))
```

```
[1] FALSE TRUE
```

```
is.na(paste(c(1, NA)))
```

```
[1] FALSE FALSE
```


Plan détaillé

3 Vecteurs

- Exemples de vecteurs
- Fonctions de création de vecteurs
- Indexation des vecteurs
- Indexation par des entiers positifs
- Indexation par des entiers négatifs
- Indexation par un vecteur logique
- Indexation par un nom

seq()

seq() (séquence) génère une série de nombres équidistants

```
seq(from = 1, to = 10, length = 25)
```

```
[1] 1.000 1.375 1.750 2.125 2.500 2.875 3.250 3.625 4.000  
[10] 4.375 4.750 5.125 5.500 5.875 6.250 6.625 7.000 7.375  
[19] 7.750 8.125 8.500 8.875 9.250 9.625 10.000
```

```
seq(from = 1, to = 5, by = 0.5)
```

```
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

```
seq(along = letters)
```

```
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22  
[23] 23 24 25 26
```

l'opérateur :

Pour les séries entières il est plus commode d'utiliser l'opérateur deux-points (:) ou la fonction `seq_len()` :

```
1:10
```

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

```
-5:5
```

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

```
6:-6
```

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

```
seq_len(10)
```

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

$c()$

`c()` **concaténation** de un ou plusieurs vecteurs :

```
c(1, 5, 6, 9)
```

```
[1] 1 5 6 9
```

```
c(1:5, seq(10, 20, length = 6))
```

```
[1] 1 2 3 4 5 10 12 14 16 18 20
```

rep()

rep **réplication** d'un vecteur :

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

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

```
rep(1:5, length = 12)
```

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

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

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

```
rep(c("un", "deux"), c(6, 3))
```

```
[1] "un"    "un"    "un"    "un"    "un"    "un"    "deux"  "deux"  "deux"
```

Création de vecteurs

Il est parfois utile de pouvoir créer des vecteurs vides pour pouvoir les manipuler ensuite avec une boucle explicite. Pour les types simples ceci se fait avec les fonctions suivantes :

```
numeric(10)
```

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

```
logical(5)
```

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

```
character(5)
```


```
[1] "" "" "" "" ""
```

Création de vecteurs

Voici un exemple d'utilisation pour calculer les carrés des dix premiers entiers :

```
x <- numeric(10)
for (i in 1:10) x[i] <- i^2
x
```

```
[1] 1 4 9 16 25 36 49 64 81 100
```

Mais sous  on utilise dans ce cas plutôt une construction plus compacte du type :

```
x <- (1:10)^2
x
```

```
[1] 1 4 9 16 25 36 49 64 81 100
```

Plan détaillé

3 Vecteurs

- Exemples de vecteurs
- Fonctions de création de vecteurs
- **Indexation des vecteurs**
 - Indexation par des entiers positifs
 - Indexation par des entiers négatifs
 - Indexation par un vecteur logique
 - Indexation par un nom

Extraction d'un élément d'un vecteur

L'extraction de l'élément de rang k d'un vecteur se fait avec l'opérateur crochet (`[]`) :

```
letters[10]
```


```
[1] "j"
```

À noter :

```
letters[100]
```

```
[1] NA
```

Indexation

L'extraction d'un ou plusieurs éléments d'un vecteur se fait par **l'indexation**. Il y a plusieurs types d'indexation possibles sous , dont :

- Indexation par un vecteur d'entiers positifs
- Indexation par un vecteur d'entiers négatifs
- Indexation par un vecteur logique
- Indexation par un vecteur de noms

Plan détaillé

3 Vecteurs

- Exemples de vecteurs
- Fonctions de création de vecteurs
- Indexation des vecteurs
- **Indexation par des entiers positifs**
- Indexation par des entiers négatifs
- Indexation par un vecteur logique
- Indexation par un nom

Indexation par des entiers positifs

On donne les rangs des éléments à conserver :

```
letters[c(3, 5, 7)]
```

```
[1] "c" "e" "g"
```

```
ind <- c(3, 5, 7)  
letters[ind]
```

```
[1] "c" "e" "g"
```

```
letters[8:13]
```

```
[1] "h" "i" "j" "k" "l" "m"
```

```
letters[c(1, 2, 1, 2)]
```

```
[1] "a" "b" "a" "b"
```

Points intéressants :

- L'utilisation d'un indice supérieur à la longueur du vecteur donne un NA.
- Les indices peuvent être répétés, faisant qu'un élément est sélectionné plusieurs fois.

C'est assez utile en pratique.

Plan détaillé

3 Vecteurs

- Exemples de vecteurs
- Fonctions de création de vecteurs
- Indexation des vecteurs
- Indexation par des entiers positifs
- **Indexation par des entiers négatifs**
- Indexation par un vecteur logique
- Indexation par un nom

Indexation par des entiers négatifs

L'utilisation d'entiers négatifs permet d'exclure les éléments correspondants.

```
letters[-5]
```

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

```
ind <- c(3, 5, 7)  
letters[-ind]
```

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

Attention : on ne peut pas mélanger des indices positifs et négatifs.

Plan détaillé

3 Vecteurs

- Exemples de vecteurs
- Fonctions de création de vecteurs
- Indexation des vecteurs
- Indexation par des entiers positifs
- Indexation par des entiers négatifs
- Indexation par un vecteur logique
- Indexation par un nom

Indexation par un vecteur logique

Seuls les éléments correspondant à une valeur TRUE sont retenus.

```
ind <- rep(c(TRUE, FALSE), length = length(letters))
ind
```

```
[1] TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE
[12] FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE
[23] TRUE FALSE TRUE FALSE
```

```
letters[ind]
```

```
[1] "a" "c" "e" "g" "i" "k" "m" "o" "q" "s" "u" "w" "y"
```

```
letters[c(T, F)]
```

```
[1] "a" "c" "e" "g" "i" "k" "m" "o" "q" "s" "u" "w" "y"
```

```
voyelles <- c("a", "e", "i", "o", "u")
letters %in% voyelles
```

```
[1] TRUE FALSE FALSE FALSE TRUE FALSE FALSE FALSE TRUE FALSE FALSE
[12] FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE TRUE FALSE
[23] FALSE FALSE FALSE FALSE
```

```
letters[letters %in% voyelles]
```

```
[1] "a" "e" "i" "o" "u"
```


Plan détaillé

3 Vecteurs

- Exemples de vecteurs
- Fonctions de création de vecteurs
- Indexation des vecteurs
- Indexation par des entiers positifs
- Indexation par des entiers négatifs
- Indexation par un vecteur logique
- Indexation par un nom

Indexation par un nom

Ceci ne fonctionne que pour les vecteurs ayant des noms.

```
x <- 1:10  
names(x) <- letters[1:10]  
x
```

```
a b c d e f g h i j  
1 2 3 4 5 6 7 8 9 10
```

```
x[c("a", "b", "c", "f")]
```

```
a b c f  
1 2 3 6
```

Exclusion par un nom

On ne peut pas directement exclure des éléments par leur nom, il faut utiliser la fonction `match()` pour récupérer leur rang avec des constructions du type :

```
x
```

```
a  b  c  d  e  f  g  h  i  j
1  2  3  4  5  6  7  8  9 10
```

```
x[-match("a", names(x))]
```

```
b  c  d  e  f  g  h  i  j
2  3  4  5  6  7  8  9 10
```

```
x[-match(c("a", "b", "c", "f"), names(x))]
```

```
d  e  g  h  i  j
4  5  7  8  9 10
```

Table des matières

- 1 Objets
- 2 Fonctions
- 3 Vecteurs
- 4 Matrices et tableaux**
- 5 Facteurs
- 6 Listes
- 7 Data.frame
- 8 Structures de contrôle

Matrices et tableaux

```
fortune(62)
```


```
Please bear with a poor newbie, who might be doing everything  
backwards (I was brought up in pure math).
```

```
-- Thomas Poulsen  
   R-help (May 2004)
```

Plan détaillé

- 4 Matrices et tableaux
 - Dimensions
 - Création de matrices
 - Opérations avec des matrices

Matrices et tableaux

Les matrices (et plus généralement les tableaux de dimensions quelconques) sont stockées dans  comme des vecteurs ayant des dimensions :

```
x <- 1:12  
dim(x) <- c(3, 4)  
x
```

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

nrow() et ncol()

Les fonctions `nrow()` et `ncol()` donnent le nombre de lignes et le nombre de colonnes d'une matrice, respectivement :

`x`

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

`nrow(x)`

[1] 3

`ncol(x)`

[1] 4

`x` est donc ici une matrice de 3 lignes et 4 colonnes.

rownames() et colnames()

On peut donner des noms aux lignes et aux colonnes avec les fonctions `rownames()` et `colnames()` :

```
rownames(x) <- paste("ligne", 1:nrow(x), sep = "")  
x
```

	[,1]	[,2]	[,3]	[,4]
ligne1	1	4	7	10
ligne2	2	5	8	11
ligne3	3	6	9	12

```
colnames(x) <- paste("colonne", 1:ncol(x), sep = "")  
x
```

	colonne1	colonne2	colonne3	colonne4
ligne1	1	4	7	10
ligne2	2	5	8	11
ligne3	3	6	9	12

Un tableau à trois dimensions

Utilisation du même vecteur pour créer un tableau à trois dimensions :

```
dim(x) <- c(2, 2, 3)
x
```

```
, , 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

Plan détaillé

- 4 Matrices et tableaux
 - Dimensions
 - Création de matrices
 - Opérations avec des matrices

La fonction `matrix()`

Les matrices peuvent aussi être créées facilement avec la fonction `matrix()` en précisant le nombre de lignes `nrow` et de colonnes `ncol` :

```
x <- matrix(1:12, nrow = 3, ncol = 4)  
x
```

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

Notez que le remplissage se fait par défaut en colonnes.

La fonction matrix()

Pour remplir une matrice ligne par ligne on utilise l'argument `byrow` :

```
x <- matrix(1:12, nrow = 3, ncol = 4, byrow = TRUE)
x
```

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

Création de matrices à partir de vecteurs

Les fonctions `cbind()` (**column bind**) et `rbind()` (**row bind**) créent des matrices à partir de vecteurs ou de matrices plus petites en les apposant :

```
y <- cbind(A = 1:4, B = 5:8, C = 9:12)
y
```

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

```
rbind(y, 0)
```

```
      A B C
[1,] 1 5 9
[2,] 2 6 10
[3,] 3 7 11
[4,] 4 8 12
[5,] 0 0 0
```

Notez que le petit vecteur (0) est recyclé.

Création de matrices diagonales

On utilise la fonction `diag()` :

```
diag(1, nrow = 5, ncol = 5)
```

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

Homogénéité des types

Les matrices ne sont pas forcément `numeric`, elles peuvent être `character` ou `logical`, par exemple :

```
matrix(month.name, nrow = 6)
```

```
      [,1]      [,2]  
[1,] "January" "July"  
[2,] "February" "August"  
[3,] "March"    "September"  
[4,] "April"    "October"  
[5,] "May"      "November"  
[6,] "June"     "December"
```

Mais tous les éléments sont toujours du même type dans une matrice.

Plan détaillé

- 4 Matrices et tableaux
 - Dimensions
 - Création de matrices
 - Opérations avec des matrices

Transposition

Pour écrire les lignes en colonnes :

x

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

t(x)

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

Les opérations arithmétiques et les fonctions mathématiques usuelles travaillent toujours **élément par élément**.

Addition

x

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

x + x

	[,1]	[,2]	[,3]	[,4]
[1,]	2	4	6	8
[2,]	10	12	14	16
[3,]	18	20	22	24

Soustraction

x

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

x - x

	[,1]	[,2]	[,3]	[,4]
[1,]	0	0	0	0
[2,]	0	0	0	0
[3,]	0	0	0	0

Multiplication (produit d'Hadamard)

x

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

x * x

	[,1]	[,2]	[,3]	[,4]
[1,]	1	4	9	16
[2,]	25	36	49	64
[3,]	81	100	121	144

Division

x

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

x/x

	[,1]	[,2]	[,3]	[,4]
[1,]	1	1	1	1
[2,]	1	1	1	1
[3,]	1	1	1	1

Fonction mathématique

x

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

log(x)

	[,1]	[,2]	[,3]	[,4]
[1,]	0.000000	0.6931472	1.098612	1.386294
[2,]	1.609438	1.7917595	1.945910	2.079442
[3,]	2.197225	2.3025851	2.397895	2.484907

Multiplication par un scalaire

x

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

pi * x

	[,1]	[,2]	[,3]	[,4]
[1,]	3.141593	6.283185	9.424778	12.56637
[2,]	15.707963	18.849556	21.991149	25.13274
[3,]	28.274334	31.415927	34.557519	37.69911

Multiplication matricielle

L'opérateur de multiplication matricielle est `%*%` :

`x`

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

`x %*% t(x)`

	[,1]	[,2]	[,3]
[1,]	30	70	110
[2,]	70	174	278
[3,]	110	278	446

Inversion d'une matrice

Pour inverser une matrice on utilise la fonction `solve()` :

```
x <- matrix((1:9)^2, nrow = 3)
x
```

```
      [,1] [,2] [,3]
[1,]     1    16    49
[2,]     4    25    64
[3,]     9    36    81
```

```
solve(x)
```

```
      [,1] [,2] [,3]
[1,]  1.291667 -2.166667  0.9305556
[2,] -1.166667  1.666667 -0.6111111
[3,]  0.375000 -0.500000  0.1805556
```

```
zapsmall(solve(x) %*% x)
```

```
      [,1] [,2] [,3]
[1,]     1     0     0
[2,]     0     1     0
[3,]     0     0     1
```

Valeurs propres et vecteur propres : eigen()

```
eigen(x)
```

```
$values
```

```
[1] 112.9839325 -6.2879696 0.3040371
```

```
$vectors
```

```
      [,1]      [,2]      [,3]
[1,] -0.3993327 -0.8494260 0.7612507
[2,] -0.5511074 -0.4511993 -0.6195403
[3,] -0.7326760 0.2736690 0.1914866
```

```
zapsmall(solve(eigen(x)$vectors) %*% x %*% eigen(x)$vectors)
```

```
      [,1]      [,2]      [,3]
[1,] 112.9839 0.00000 0.00000
[2,] 0.0000 -6.28797 0.00000
[3,] 0.0000 0.00000 0.30404
```

Table des matières

- 1 Objets
- 2 Fonctions
- 3 Vecteurs
- 4 Matrices et tableaux
- 5 Facteurs**
- 6 Listes
- 7 Data.frame
- 8 Structures de contrôle

Les facteurs

```
fortune(59)
```

```
Let's not kid ourselves: the most widely used piece of software for  
statistics is Excel.
```

```
-- Brian D. Ripley ('Statistical Methods Need Software: A View of  
Statistical Computing')
```


```
Opening lecture RSS 2002, Plymouth (September 2002)
```

Plan détaillé

5 Facteurs

- Variables qualitatives
- Création de facteurs
- Manipulation des facteurs

Les facteurs : représentation des variables qualitatives

Les facteurs sont la représentation sous  des **variables qualitatives** (e.g., la couleur des yeux, le genre (σ , φ), un niveau de douleur). Les valeurs possibles d'une variable qualitative sont les modalités (`levels`).

Il existe deux types de variables qualitatives :

- Les variables qualitatives **non ordonnées**, par exemple le genre (σ , φ).
- Les variables qualitatives **ordonnées**, par exemple un niveau de douleur (rien, léger, moyen, fort).

Plan détaillé

5 Facteurs

- Variables qualitatives
- Création de facteurs
- Manipulation des facteurs

Création de variables qualitatives ordonnées

```
douleur <- c("rien", "fort", "moyen", "moyen", "leger")  
douleur
```

```
[1] "rien" "fort" "moyen" "moyen" "leger"
```

```
fdouleur <- factor(douleur, levels = c("rien", "leger",  
    "moyen", "fort"), ordered = TRUE)  
fdouleur
```

```
[1] rien fort moyen moyen leger  
Levels: rien < leger < moyen < fort
```

Création de variables qualitatives ordonnées

Attention, si l'argument `levels` n'est pas précisé, c'est l'ordre alphabétique qui sera utilisé. C'est une solution qui n'est pas portable et qui ne donne pas forcément ce que l'on veut :

```
douleur <- c("rien", "fort", "moyen", "moyen", "leger")
douleur

[1] "rien" "fort" "moyen" "moyen" "leger"

fdouleur <- factor(douleur, ordered = TRUE)
fdouleur

[1] rien fort moyen moyen leger
Levels: fort < leger < moyen < rien
```

Création de variables qualitatives non ordonnées

```
couleurs <- c("bleu", "bleu", "bleu", "blanc", "rouge",  
             "rouge", "noir")  
fcouleurs <- factor(couleurs, levels = c("vert", "bleu",  
             "blanc", "rouge"))  
fcouleurs
```

```
[1] bleu  bleu  bleu  blanc rouge rouge <NA>  
Levels: vert bleu blanc rouge
```

Plan détaillé

5 Facteurs

- Variables qualitatives
- Création de facteurs
- Manipulation des facteurs

Accéder au codage interne des modalités

Les modalités d'un facteur sont stockées en interne par des entiers, pour y accéder utiliser la fonction `as.integer()` :

```
fdouleur
```

```
[1] rien fort moyen moyen léger  
Levels: fort < léger < moyen < rien
```

```
as.integer(fdouleur)
```

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

```
fcouleurs
```

```
[1] bleu bleu bleu blanc rouge rouge <NA>  
Levels: vert bleu blanc rouge
```

```
as.integer(fcouleurs)
```

```
[1] 2 2 2 3 4 4 NA
```

Changer les modalités

Utiliser la fonction `levels()` :

```
fcouleurs
```

```
[1] bleu  bleu  bleu  blanc rouge rouge <NA>  
Levels: vert bleu blanc rouge
```

```
as.integer(fcouleurs)
```

```
[1]  2  2  2  3  4  4 NA
```

```
levels(fcouleurs) <- c("green", "blue", "white", "red")  
fcouleurs
```

```
[1] blue  blue  blue  white red   red   <NA>  
Levels: green blue white red
```

Effectifs des modalités

Utiliser la fonction `table()` :

```
fdouleur
```

```
[1] rien fort moyen moyen léger
Levels: fort < léger < moyen < rien
```

```
table(fdouleur)
```

```
fdouleur
fort léger moyen rien
      1      1      2      1
```

```
fcouleurs
```

```
[1] blue blue blue white red red <NA>
Levels: green blue white red
```

```
table(fcouleurs)
```

```
fcouleurs
green blue white red
      0      3      1      2
```

Table des matières

- 1 Objets
- 2 Fonctions
- 3 Vecteurs
- 4 Matrices et tableaux
- 5 Facteurs
- 6 Listes**
- 7 Data.frame
- 8 Structures de contrôle

Les listes

```
fortune(48)
```

```
Release 1.0.0
```

```
(silence)
```

```
Wow! Thank you! [...] If I am allowed to ask just one question today:  
How do you fit 48 hours of coding in an ordinary day? Any hints will  
be appreciated ... :-)
```

```
-- Detlef Steuer (on 2000-02-29)
```



```
    R-help (February 2000)
```

Plan détaillé


6 Listes

- Intérêt des listes
- Manipulation des listes

Intérêt des listes

- Les listes sont une structure de données **très flexible** et très utilisée dans .
- Une liste est un vecteur dont les éléments ne sont pas nécessairement du même type. Un élément d'une liste est un objet  **quelconque**, y compris une autre liste.
- la fonction `list()` permet de créer des listes.
- les éléments de la liste sont en général extraits par leur noms (avec l'opérateur `$`)

Les fonctions renvoient souvent des listes

Les fonctions de  renvoient souvent une liste plutôt qu'un simple vecteur. Rien n'est plus facile alors d'extraire ce qui nous intéresse.

```
echantillon <- rnorm(100)
resultat <- t.test(echantillon)
is.list(resultat)
```

```
[1] TRUE
```

```
names(resultat)
```

```
[1] "statistic"    "parameter"    "p.value"      "conf.int"
[5] "estimate"    "null.value"   "alternative"   "method"
[9] "data.name"
```

```
resultat$conf.int
```

```
[1] -0.37724829  0.07623924
attr("conf.level")
[1] 0.95
```

Les listes comme objets composites

Les listes permettent de créer des *objets composites* contenant des objets divers et variés.

```
x <- list(fonction = seq, longueur = 10)
x$fonction
```


```
function (...)
UseMethod("seq")
<environment: namespace:base>
```

```
x$longueur
```

```
[1] 10
```

```
x$fonction(length = x$longueur)
```

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

Les fonctions sont aussi des objets . Dans ce cas, l'élément `fonction` est la fonction `seq()` et peut être appelée comme n'importe quelle autre fonction.

Plan détaillé

6 Listes

- Intérêt des listes
- Manipulation des listes

Les listes

Voyons un exemple concret avec un jeu de données sur la prise alimentaire dans un groupe de femmes avant et après leurs règles :

```
prise.avant <- c(5260, 5470, 5640, 6180, 6390, 6515)
prise.apres <- c(3910, 4220, 3885, 5160, 5645, 4680)
maliste <- list(avant = prise.avant, apres = prise.apres)
maliste
```

```
$avant
```

```
[1] 5260 5470 5640 6180 6390 6515
```

```
$apres
```

```
[1] 3910 4220 3885 5160 5645 4680
```

Extraction d'un élément d'une liste

Un élément d'une liste peut être extrait par nom avec l'opérateur dollar (\$) ou par position avec l'opérateur double crochet ([]).

```
maliste$avant
```

```
[1] 5260 5470 5640 6180 6390 6515
```

```
maliste[[2]]
```

```
[1] 3910 4220 3885 5160 5645 4680
```


Abréviation des noms des éléments

Les noms des éléments d'une liste peuvent être abrégés, mais attention l'opérateur d'extraction `$` ne prévient pas toujours en cas d'ambiguïté :

```
maliste
```

```
$avant
```

```
[1] 5260 5470 5640 6180 6390 6515
```

```
$apres
```

```
[1] 3910 4220 3885 5160 5645 4680
```

```
maliste$apres
```

```
[1] 3910 4220 3885 5160 5645 4680
```

```
maliste$av
```

```
[1] 5260 5470 5640 6180 6390 6515
```

```
uneliste <- list(a = 0, alpha = 1)
```

```
uneliste$a
```

```
[1] 0
```

Différence entre `[[` et `[`

Attention : l'opérateur `[[` ne permet d'extraire qu'un seul élément à la fois (à la différence de l'opérateur `[` pour les vecteurs).
L'opérateur `[` sur les listes ne fait pas d'extraction mais renvoie une **liste** avec les éléments sélectionnés.

```
maliste <- list(a = 1, b = 2, c = 3)
maliste[[2]]
```

```
[1] 2
```

```
maliste$b
```

```
[1] 2
```

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

```
$a
```

```
[1] 1
```

```
$c
```

```
[1] 3
```

Ajout d'un élément à une liste

Il est très facile de rajouter des éléments à une liste existante :

```
maliste
```

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

```
$b  
[1] 2
```

```
$c  
[1] 3
```

```
maliste$message <- "hello"  
maliste$message
```

```
[1] "hello"
```

Suppression d'un élément d'une liste

Comme pour les vecteurs, on peut utiliser des indices négatifs pour supprimer un élément d'une liste :

```
maliste <- list(x = 1, y = 2)
maliste
```

```
$x
[1] 1
```

```
$y
[1] 2
```

```
maliste[-1]
```

```
$y
[1] 2
```

Où mettre l'élément nommé à NULL :

```
maliste <- list(x = 1, y = 2)
maliste$x <- NULL
maliste
```

```
$y
[1] 2
```

Insertion d'un élément dans une liste

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

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

```
$b  
[1] 2
```

```
$c  
[1] 3
```

```
c(maliste[1:2], insert = pi, maliste[3:3])
```

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

```
$b  
[1] 2
```

```
$insert  
[1] 3.141593
```

```
$c  
[1] 3
```

Table des matières

- 1 Objets
- 2 Fonctions
- 3 Vecteurs
- 4 Matrices et tableaux
- 5 Facteurs
- 6 Listes
- 7 Data.frame**
- 8 Structures de contrôle

Le type data.frame

```
fortune(2)
```

```
Bug, undocumented behaviour, feature? I don't know. It all seems to  
work in 1.6.0, so everyone should downgrade now... :)
```

```
-- Barry Rowlingson  
   R-help (July 2003)
```

Plan détaillé

7 Data.frame

- Intérêt des data.frame
- Création d'un data.frame
- Réarrangements d'un data.frame

Intérêt des data.frame

La classe `data.frame` est la plus appropriée pour stocker les jeux de données et est probablement la classe la plus fréquemment utilisée en pratique. Ce sont essentiellement des listes dont tous les éléments ont la même longueur.

- Chaque élément d'un `data.frame` doit être un vecteur du type `factor`, `numeric`, `character` ou `logical`.
- Tous ces éléments doivent avoir la même longueur
- Ils sont similaires aux matrices de part leur structure en `table rectangulaire`, la seule différence est que les colonnes peuvent être de types différents.

Exemple concret de data.frame

Les Iris de Fisher, un jeu de données classique en statistiques.



Iris setosa



Iris versicolor



Iris virginica

Les iris de Fisher

```
data(iris)
head(iris)
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5.0	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa

```
class(iris$Sepal.Length)
```

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

```
class(iris$Species)
```

```
[1] "factor"
```

Plan détaillé

7 Data.frame

- Intérêt des data.frame
- Création d'un data.frame
- Réarrangements d'un data.frame

Création par lecture dans un fichier

C'est de loin la situation la plus courante :

```
t3var <- read.table("http://pbil.univ-lyon1.fr/R/donnees/t3var.txt",  
  header = TRUE)  
head(t3var)
```

	sexe	poi	tai
1	h	60	170
2	f	57	169
3	f	51	172
4	f	55	174
5	f	50	168
6	f	50	161

```
class(t3var)
```

```
[1] "data.frame"
```

La fonction data.frame

Un objet de type `data.frame` peut être créé par la fonction `data.frame()` :

```
d <- data.frame(prise.avant, prise.apres)
d
```

	prise.avant	prise.apres
1	5260	3910
2	5470	4220
3	5640	3885
4	6180	5160
5	6390	5645
6	6515	4680

```
d$prise.apres
```

```
[1] 3910 4220 3885 5160 5645 4680
```

Comme la classe `data.frame` hérite de la classe `list`, l'opérateur `$` peut être utilisé pour extraire des colonnes.

Indexation des matrices et jeux de données

L'indexation des matrices et des variables du type `data.frame` sont très similaire. Elles utilisent aussi l'opérateur crochet `[`, mais ont besoin de deux indices. Si un indice n'est pas précisé, toutes les lignes ou colonnes correspondant sont sélectionnées.

```
x <- matrix(1:12, 3, 4)
x
```

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

```
x[1:2, 1:2]
```

```
[,1] [,2]
[1,]  1  4
[2,]  2  5
```

```
x[1:2, ]
```

```
[,1] [,2] [,3] [,4]
[1,]  1  4  7 10
[2,]  2  5  8 11
```

Indexation des matrices et jeux de données

Si une seule ligne ou colonne est sélectionnée, elle est convertie en un vecteur. Ceci peut être neutralisé en ajoutant `drop = FALSE`

```
x[1, ]
```

```
[1] 1 4 7 10
```

```
x[1, , drop = FALSE]
```

```
      [,1] [,2] [,3] [,4]  
[1,]    1    4    7   10
```


Indexation des matrices et jeux de données

Les variables du type `data.frame` se comportent de la même manière :

```
d[1:3, ]
```

```
  prise.avant prise.apres  
1      5260      3910  
2      5470      4220  
3      5640      3885
```

```
d[1:3, "prise.avant"]
```

```
[1] 5260 5470 5640
```

```
d[d$prise.apres < 5000, 1, drop = FALSE]
```

```
  prise.avant  
1      5260  
2      5470  
3      5640  
6      6515
```

Plan détaillé

7 Data.frame

- Intérêt des data.frame
- Création d'un data.frame
- Réarrangements d'un data.frame

Tri d'un data.frame

C'est un cas particulier d'indexation utilisant la fonction `order()` :

```
order(d$prise.apres)
```

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

```
d[order(d$prise.apres), ]
```

	prise.avant	prise.apres
3	5640	3885
1	5260	3910
2	5470	4220
6	6515	4680
4	6180	5160
5	6390	5645

Table des matières

- 1 Objets
- 2 Fonctions
- 3 Vecteurs
- 4 Matrices et tableaux
- 5 Facteurs
- 6 Listes
- 7 Data.frame
- 8 Structures de contrôle**

Plan détaillé

8 Structures de contrôle

- Faire des choix
- Répéter une action
- Boucles implicites

if(cond) expr

Pour faire un choix simple :

```
f <- function(x) {  
  if (x%%2 == 0) {  
    return("pair")  
  }  
}
```

```
f(2)
```

```
[1] "pair"
```

```
f(3)
```

```
if(cond) expr1 else expr2
```

Pour faire choisir entre une condition et son alternative :

```
f <- function(x) {  
  if (x%%2 == 0) {  
    return("pair")  
  }  
  else {  
    return("impair")  
  }  
}  
f(2)  
[1] "pair"  
  
f(3)  
[1] "impair"
```

switch(expr, ...)

Pour faire des choix multiples :

```
f <- function(x) {  
  switch(x, "un", "deux", "trois", "quatre")  
}  
f(1)
```

```
[1] "un"
```

```
f(2)
```

```
[1] "deux"
```

```
f(5)
```

```
NULL
```


switch(expr, ...)

En travaillant avec une expression de type chaîne de caractères on peut préciser un choix par défaut :

```
f <- function(chaine) {  
  switch(chaine, un = 1, deux = 2, trois = 3, quatre = 4,  
         "je ne sais pas")  
}  
f("un")
```

```
[1] 1
```

```
f("deux")
```

```
[1] 2
```

```
f("cent")
```

```
[1] "je ne sais pas"
```

```
ifelse(test, oui, non)
```

Il existe une version **vectorisée** très puissante :

```
x <- rnorm(10)
```

```
x
```

```
[1] -0.47162735 -0.77978020 -1.07174132 -1.29467154 -0.47645939
```

```
[6]  0.52778427  0.49125277 -1.59025716 -2.74457419 -0.03007822
```

```
ifelse(x > 0, "positif", "negatif")
```

```
[1] "negatif" "negatif" "negatif" "negatif" "negatif" "positif"
```

```
[7] "positif" "negatif" "negatif" "negatif"
```

Plan détaillé

8 Structures de contrôle

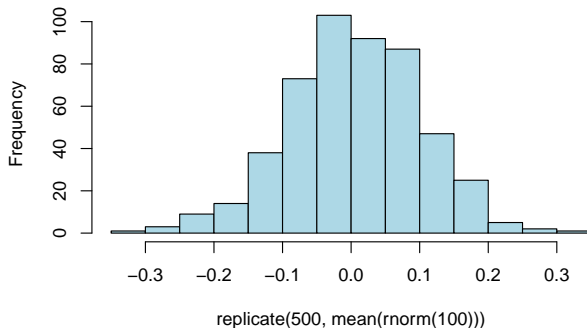
- Faire des choix
- Répéter une action
- Boucles implicites

replicate()

On est souvent amené à faire des simulations pour apprécier la distribution d'échantillonnage d'une statistique. La fonction `replicate()` permet de le faire très facilement :

```
hist(replicate(500, mean(rnorm(100))), col = "lightblue")
```

Histogram of replicate(500, mean(rnorm(100)))



```
for(var in seq) expr
```

On peut aussi faire des boucles explicites à l'ancienne :

```
for (i in 1:5) print(i)
```

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

```
for (carac in letters[1:5]) print(carac)
```

```
[1] "a"  
[1] "b"  
[1] "c"  
[1] "d"  
[1] "e"
```


for(var in seq) expr

Remplacer toutes les valeurs négatives d'un vecteur par -1.

Approche laborieuse classique :

```
x <- rnorm(10)
for (i in 1:length(x)) {
  if (x[i] < 0)
    x[i] <- -1
}
x
```

```
[1] -1.00000000  0.57698909  0.52651301 -1.00000000  1.06504920
[6]  0.00999927 -1.00000000  0.71572971 -1.00000000 -1.00000000
```

Approche sous  :

```
x <- rnorm(10)
x[x < 0] <- -1
x
```

```
[1]  0.99210114  0.81975449  2.21701969  1.46007751  0.01846901
[6] -1.00000000 -1.00000000  0.11346376 -1.00000000 -1.00000000
```

```
for(var in seq) expr
```

Remarque : on aurait pu utiliser aussi ici le if vectorisé ainsi :

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

```
[1] -1.0000000  1.3615624  1.5683767  1.1430187 -1.0000000 -1.0000000
[7]  0.4646724  1.1394565 -1.0000000 -1.0000000
```

while(cond) expr

Tant que la condition est vraie on répète l'expression :

```
i <- 1
while (i <= 5) {
  print(i)
  i <- i + 1
}
```

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


repeat expr

On répète l'expression tant qu'un break n'en fait pas sortir :

```
i <- 1
repeat {
  print(i)
  i <- i + 1
  if (i > 5)
    break
}
```

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

next

On peut sauter un tour dans une boucle. Par exemple pour mettre à zéro tous les éléments d'une matrice sauf les éléments diagonaux :


```
(x <- matrix(rnorm(9), 3, 3))
```

```
      [,1]      [,2]      [,3]
[1,] 1.2478061 0.06974192 -0.1352661
[2,] 1.6225568 0.64071224 -0.5948983
[3,] 0.5394343 1.68095070  1.0463645
```

```
for (i in 1:3) {
  for (j in 1:3) {
    if (i == j)
      next
    x[i, j] <- 0
  }
}
x
```

```
      [,1]      [,2]      [,3]
[1,] 1.247806 0.0000000 0.0000000
[2,] 0.000000 0.6407122 0.0000000
[3,] 0.000000 0.0000000 1.046365
```

next

Remarque : sous , on ferait plus simplement :

```
(x <- matrix(rnorm(9), 3, 3))
```

```
      [,1]      [,2]      [,3]  
[1,] 1.146043 -0.2329530 -0.1141616  
[2,] 1.040547 -1.6224200  1.9552784  
[3,] 0.954766 -0.8191705 -1.0065945
```

```
(x <- diag(diag(x)))
```

```
      [,1]      [,2]      [,3]  
[1,] 1.146043  0.000000  0.000000  
[2,] 0.000000 -1.62242  0.000000  
[3,] 0.000000  0.00000 -1.006594
```

Plan détaillé

8 Structures de contrôle

- Faire des choix
- Répéter une action
- Boucles implicites

lapply()

`lapply()` permet d'appliquer une fonction à tous les éléments d'une liste ou d'un vecteur :

```
maliste <- as.list(1:3)
f <- function(x) x^2
lapply(maliste, f)
```

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

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

```
[[3]]
[1] 9
```

lapply()

`lapply()` retourne une liste :

```
lapply(1:4, f)
```

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

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

```
[[3]]  
[1] 9
```

```
[[4]]  
[1] 16
```

sapply()

sapply() essaye de simplifier le résultat en un vecteur :

```
sapply(maliste, f)
```

```
[1] 1 4 9
```

```
sapply(1:10, f)
```

```
[1] 1 4 9 16 25 36 49 64 81 100
```

tapply()

La fonction **tapply()** permet d'appliquer une fonction à des groupes définis par une variable qualitative :

```
data(iris)
head(iris)
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5.0	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa

```
tapply(iris$Sepal.Length, iris$Species, mean)
```

setosa	versicolor	virginica
5.006	5.936	6.588

apply()

`apply()` permet d'appliquer une fonction aux lignes (1) ou aux colonnes (2) d'une matrice :

```
(mat <- matrix(rpois(12, 2), 3, 4))
```

	[,1]	[,2]	[,3]	[,4]
[1,]	1	2	3	3
[2,]	3	3	3	1
[3,]	0	1	2	2

```
apply(mat, 1, sum)
```

```
[1] 9 10 5
```

```
apply(mat, 2, sum)
```

```
[1] 4 6 8 6
```

apply()

Remarque : les fonctions `colSums()` et `rowSums()` permettent d'obtenir le même résultat :

```
rowSums(mat)
```

```
[1]  9 10  5
```

```
colSums(mat)
```

```
[1] 4 6 8 6
```

apply()

Exemple d'application : on considère le jeu de données **airquality** :

```
data(airquality)
head(airquality)
```

	Ozone	Solar.R	Wind	Temp	Month	Day
1	41	190	7.4	67	5	1
2	36	118	8.0	72	5	2
3	12	149	12.6	74	5	3
4	18	313	11.5	62	5	4
5	NA	NA	14.3	56	5	5
6	28	NA	14.9	66	5	6

Il y a des données **manquantes**. Que faire ?

apply()

Première solution : ne garder que les individus entièrement documentés :

```
head(airquality)
```

	Ozone	Solar.R	Wind	Temp	Month	Day
1	41	190	7.4	67	5	1
2	36	118	8.0	72	5	2
3	12	149	12.6	74	5	3
4	18	313	11.5	62	5	4
5	NA	NA	14.3	56	5	5
6	28	NA	14.9	66	5	6

```
head(airquality[complete.cases(airquality), ])
```

	Ozone	Solar.R	Wind	Temp	Month	Day
1	41	190	7.4	67	5	1
2	36	118	8.0	72	5	2
3	12	149	12.6	74	5	3
4	18	313	11.5	62	5	4
7	23	299	8.6	65	5	7
8	19	99	13.8	59	5	8

apply()

Deuxième solution : remplacer les valeurs manquantes par la moyenne de la variable. Approche à l'ancienne :

```
for (i in 1:nrow(airquality)) {
  for (j in 1:ncol(airquality)) {
    if (is.na(airquality[i, j])) {
      airquality[i, j] <- mean(airquality[, j],
                             na.rm = TRUE)
    }
  }
}
head(airquality)
```

	Ozone	Solar.R	Wind	Temp	Month	Day
1	41.00000	190.0000	7.4	67	5	1
2	36.00000	118.0000	8.0	72	5	2
3	12.00000	149.0000	12.6	74	5	3
4	18.00000	313.0000	11.5	62	5	4
5	42.12931	185.9315	14.3	56	5	5
6	28.00000	185.9315	14.9	66	5	6

apply()

Approche avec **apply()** :

```
data(airquality)
head(apply(airquality, 2, function(x) ifelse(is.na(x),
      mean(x, na.rm = TRUE), x)))
```

	Ozone	Solar.R	Wind	Temp	Month	Day
[1,]	41.00000	190.0000	7.4	67	5	1
[2,]	36.00000	118.0000	8.0	72	5	2
[3,]	12.00000	149.0000	12.6	74	5	3
[4,]	18.00000	313.0000	11.5	62	5	4
[5,]	42.12931	185.9315	14.3	56	5	5
[6,]	28.00000	185.9315	14.9	66	5	6

Il est rare que l'on ait besoin de faire des boucles explicites dans .