Introduction à R

Fonctions avancées

Pascal Bessonneau

06/2015

Les fonctions

Les fonctions sont un type d'objets R à part entière. Ainsi il existe comme pour les autres types d'objets une fonction *is* correspondante :

```
is.function( { function ( x ) {
 x^2
} } )

## [1] TRUE
```

Les fonctions

Ce qui peut être perturbant pour les débutants est l'utilisation que vous avez pu voir de fonctions anonymes : les fonctions sont utilisées directement par exemple dans une fonction *apply*.
Mais les fonctions peuvent être également stockées pour être réutilisées plusieurs fois.

```
my.square <- function ( x ) {
 return(x^2)
}
my.square(3)

## [1] 9
```

Les fonctions

Par défaut, si la dernière ligne renvoie une valeur, cette valeur est retournée par la fonction. Néanmoins pour rendre le code plus lisible et surtout plus robuste, il convient d'utiliser la fonction *return* qui prend **un seul** argument qui est renvoyé comme valeur de retour de la fonction.
Les fonctions en R ne renvoie qu'un seul objet. Par conséquent, il est souvent nécessaire de renvoyer des *lists* ou des *data.frames* pour récupérer l'ensemble du matériel créé au sein de la fonction.

Les fonctions

Il existe une autre fonction similaire à *return* : *invisible*. Elle est utilisée abondamment dans R notamment par les commandes graphiques (ou *t.test* par exemple).
Elle permet de ne renvoyer une valeur que lorsque l'appel de la fonction est dans un contexte d'évaluation.

```
my.square <- function ( x ) {
 invisible(x^2)
}
my.square(3)
(my.square(3))

## [1] 9
```

Portée des variables dans une fonction

Dans R, les fonctions héritent de l'environnement père : c'est-à-dire que les objets disponibles dans l'environnement d'appel de la fonction le sont aussi au sein de la fonction.
Mais les objets passés à la fonction sont des copies. Par conséquent, en R, toutes les modifications faîtes sur les objets au sein d'une fonction sont perdus. De plus si un objet est créé avec un nom existant dans l'environnement père, le nom de cet objet fait désormais référence à l'objet créé au sein de la fonction (et non à l'objet de même nom dans l'environnement père).

Portée des variables dans une fonction

Pour les personnes disposant d'un bagage informatique solide, R utilise des passages par valeurs (et non par références) et utilise un procédé d'évaluation dit *lazy*...
Pour simplifier, tout objet n'est évalué que si l'évaluation est effectivement nécessaire dans le code. Il en va de même pour les objets copiés.
Ce phénomène est bien expliqué dans les manuels de R et dans les ouvrages avancés sur R.

Portée des variables dans une fonction

On peut donc accéder à une valeur définie hors de la fonction.

```
z <- 2
 my.square <- function ( x ) {
 return(z*x^2)
}
my.square(3)

## [1] 18
```

Portée des variables dans une fonction

A l'intérieur de la fonction, l'objet peut être modifié mais les changements resteront locaux et seront perdus à la fermeture de la fonction.

```
z <- 2
 my.square <- function ( x ) {
 z <- 4
 return(z*x^2)
}
my.square(3)

## [1] 36

z

## [1] 2
```

Environnement

Les variables créées dans la fonction sont détruites après la fin de l'exécution.

Les arguments d'une fonction

Les arguments peuvent être soit obligatoires soit optionnels.
Les arguments obligatoires ne prennent pas de valeur par défaut.
C'est le cas pour le x de la fonction présentée précédemment dans ce document.
Les arguments sont avant tout positionnels. Mais pas seulement.
Voyons la syntaxe de l'aide de la fonction *t.test*...

Les arguments

```
t.test(x, ...)

## Default S3 method:
t.test(x, y = NULL,
       alternative = c("two.sided", "less", "greater"),
       mu = 0, paired = FALSE, var.equal = FALSE,
       conf.level = 0.95, ...)
```

Les arguments

La première ligne indique que la fonction n'attends qu'un paramètre obligatoire x. On retrouve cette information dans la partie qui est réservée à l'appel par défaut de la fonction : il n'y a pas de valeurs par défaut pour x.
Par contre, tous les autres arguments ont des valeurs par défaut ce qui indique qu'ils sont optionnels.

Les arguments

On pourrait par exemple comparer la moyenne de deux vecteurs en appelant la fonction :

```
t.test(rnorm(1000),y=rnorm(1000,2))

##
##  Welch Two Sample t-test
##
## data:  rnorm(1000) and rnorm(1000, 2)
## t = -46.404, df = 1996.6, p-value < 2.2e-16
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -2.164571 -1.989029
## sample estimates:
##   mean of x   mean of y
## -0.07859808  1.99820182
```

Les arguments

Mais les arguments étant en premier lieu positionnels, cet appel suffit :

```
t.test(rnorm(1000),rnorm(1000,2))

##
##  Welch Two Sample t-test
##
## data:  rnorm(1000) and rnorm(1000, 2)
## t = -45.875, df = 1994.6, p-value < 2.2e-16
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -2.132258 -1.957425
## sample estimates:
##   mean of x   mean of y
## -0.02428612  2.02055538
```

Les arguments

Les arguments peuvent être passés de façon positionnels mais alourdirait le code. Aussi, on peux plus simplement préciser un couple *nom/valeur par défaut*.

Les arguments

```
t.test(rnorm(1000),rnorm(1000,2),var.equal=TRUE)

##
##  Two Sample t-test
##
## data:  rnorm(1000) and rnorm(1000, 2)
## t = -43.926, df = 1998, p-value < 2.2e-16
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -2.109948 -1.929595
## sample estimates:
##   mean of x   mean of y
## -0.04069228  1.97907916
```

Les arguments

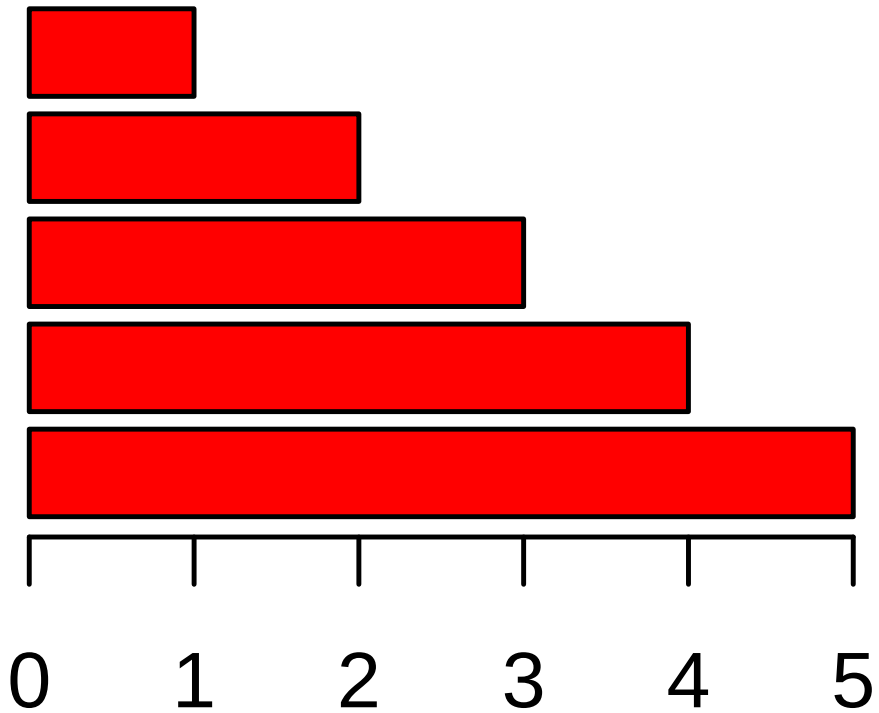
En temps normal lorsqu'un nom de paramètre incorrect est utilisé, R lève une exception.
Toutefois, lors de la création de la fonction, on peut utiliser un argument spécial : "...".
L'utilisation de cet argument indique à R que des arguments supplémentaires peuvent être passés à la fonction.
R ne lèvera pas d'exception si la correspondance entre le nom des arguments d'appel et le nom des arguments définis n'est pas correct.
Par contre il conserve les arguments supplémentaires et peut les passer à une autre fonction appelée au sein de la première fonction.

Les arguments

C'est extrèmement pratique pour *surcharger* une fonction existante. Le plus souvent pour des fonctions graphiques qui ont de nombreux paramètres.
Par exemple, pour créer des *barplot* différents des barplots par défaut...

```
my.barplot <- function( x, horiz=T, ... ) {
  barplot( x, horiz=horiz, ... )
}
#my.barplot( c(5,4,3,2,1), col="red" )
```

Les arguments



Changement dans l'environnement père...

En fait il existe une possibilité pour changer la valeur d'une variable dans l'environnement père.
C'est pratique pour modifier une *data.frame* encombrante par exemple.

Changement dans l'environnement père...

```
i <- 1
a <- function (x) { i <- 2 }
i

## [1] 1

i <- 1
a <- function (x) { i <<- 2 }
a(7);i;

## [1] 2
```

Changement dans l'environnement père...

L'inconvénient est que cela rend la fonction dépendante de l'environnement père et du nom des variables dans celui-ci. Son utilisation est donc à limiter sauf cas particuliers.

Les boucles

Les boucles sont à éviter car lentes à exécuter. Il faut leur préférer les fonctions de type *apply*. La syntaxe d'une boucle est la suivante...

```
for ( mavar in sequence ) {
        ... code R...
}
```

la variable *mavar* prend à chaque itération un élement de *sequence* dans l'ordre. Les itérations peuvent se faire sur un type quelconque comme des entiers (usuels) mais également un vecteur de *character* par exemple. Ou bien un vecteur de fonctions...

Les tests

Les tests ont la structure suivante :

```
if ( valeur ) {
        ... code R...
}
```

ou

```
if ( valeur ) {
 ... code R...
} else {
... code R...
}
```

Les tests

La condition est executé si la valeur est *TRUE*, *T* ou différent de 0. Attention, le vecteur booléen doit être de longueur 1. A l'intérieur d'un test, R attend *T* ou *F* et pas *c(T,F,T)*.

Les tests

Les fonctions à connaître sont donc *any* qui renvoie vrai si au moins un élement est vrai dans le vecteur passé en argument.
Et la fonction *all* qui renvoie vrai si toutes les valeurs du vecteur passé en argument sont vrai.

Les tests

Des opérations sur les booléens disponibles :

- qui *renvoient* des vecteurs de longueur plus grande que 1

```
& : et
| : ou
```

- qui *renvoient* des vecteurs de longueur 1

```
&& : et
|| : ou
```

Les tests

Il y a une fonction à connaître car très rapide et très simple :

`ifelse( mavar, valeur_si_vrai, valeur_si_faux )`

Par exemple :

```
ifelse( rnorm(10) > 0, 1, -1 )

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

Stopper l'exécution

La fonction *stop* permet d'arrêter un script et d'indiquer une erreur.

```
if ( class != "numeric" ) stop("Non numerique")
```

Les différentes fonctions

Dans la famille *apply*, on a en fait :

```
lapply(X, FUN, ...)
sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)
vapply(X, FUN, FUN.VALUE, ..., USE.NAMES = TRUE)
replicate(n, expr, simplify = TRUE)
```

Les différentes fonctions

Par exemple, nous voulons par exemple récupérer les quantiles de toutes les variables numériques. Pour cela, nous utilisons la fonction *apply*.

Les différentes fonctions

```
(r <- apply(iris[,1:4],2,quantile))

##      Sepal.Length Sepal.Width Petal.Length
## 0%            4.3         2.0         1.00
## 25%           5.1         2.8         1.60
## 50%           5.8         3.0         4.35
## 75%           6.4         3.3         5.10
## 100%          7.9         4.4         6.90
##      Petal.Width
## 0%           0.1
## 25%          0.3
## 50%          1.3
## 75%          1.8
## 100%         2.5
```

Les différentes fonctions

La fonction *apply* permet d'appliquer une fonction sur une *data.frame* dans le sens :

- des lignes, ligne par ligne, avec l'indice 1
- des colonnes, colonne par colonne, avec l'indice 2
- cellule par cellule avec l'indice *1 :2* (ou *c(1,2)*)

Les fonctions apply

Donc pour l'exemple précédent, calculer les quantiles, on demande à R de passer chaque colonne à la fonction quantile.
La fonction quantile rend un vecteur et R se "débrouille" tout seul avec les vecteurs résultats : il les aggrège sous forme de matrice.

Les différentes fonctions

Par exemple sapply, prends comme argument une *list* et renvoie quelque chose de simplifié quand elle le peut.
Par exemple pour retrouver les colonnes numeric d'une *data.frame*...

```
sapply( iris, is.numeric )

## Sepal.Length  Sepal.Width Petal.Length
##         TRUE         TRUE         TRUE
##  Petal.Width      Species
##         TRUE        FALSE
```

Les différentes fonctions

Pourquoi ça marche ?
Parce que *data.frame* peut être convertie en *list* puis la fonction est appliquée à chaque élement de la *list*.

```
str(as.list(iris))

## List of 5
##  $ Sepal.Length: num [1:150] 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num [1:150] 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 .
##  $ Petal.Length: num [1:150] 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5
##  $ Petal.Width : num [1:150] 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1
```

Les différentes fonctions

L'avantage de *sapply* est qu'elle renvoie un objet simplifié par rapport à *lapply*.
vapply est identique avec un contrôle sur le type d'objet renvoyé.

Les différentes fonctions

replicate est une fonction extrêmement utile. Un des gros avantages de R est qu'il permet très aisement de simuler des données.
replicate est une des fonctions qui permet de le faire en répétant une boucle tout en générant des nombres aléatoires.

les autres fonctions apply

mapply se distingue car elle peut prendre plusieurs arguments.
vapply est utilisé sur les vecteurs et permet la vérification du type en sortie.

. . .

Les différentes fonctions

```
set.seed(42)
system.time(
res1 <- replicate( 10000, function() { return(mean(rnorm(1000))) } )
)

##    user  system elapsed
##   0.008   0.000   0.006

system.time({
res2 <- numeric(10000)
for ( ii in 1:10000 ) { res2[ii] <- mean(rnorm(1000)) }
})

##    user  system elapsed
##   0.868   0.000   0.870
```

Les différentes fonctions

Ce qu'il ne faut surtout pas faire :

```
system.time({
res2 <- c()
for ( ii in 1:10000 ) { res2 <- c( res2, mean(rnorm(1000)) ) }
})

##    user  system elapsed
##   1.028   0.000   1.030
```

Un exemple, le bootstrap...

```
n <- 1000
set.seed(42)
b <- replicate( n, mean( sample( patient$totalechelle,
                                 length(patient$totalechelle),
                                 replace = T ), na.rm=T ) )
mean((b-mean(b))^2)
```

Les boucles

```
for ( ii in 1:4 ) { print(ii) }
```

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

```
for ( ww in LETTERS[1:4] ) { print(ww) }
```

```
## [1] "A"
## [1] "B"
## [1] "C"
## [1] "D"
```

Les boucles

En vrai, une boucle pourrait servir à ça :

```
a <- numeric(4)
for ( ii in 1:4 ) { a[ii] <- mean(rnorm(1000)) }
a

## [1] -0.057470355  0.005562816  0.008292973
## [4] -0.009122556
```

Ce qui s'écrit plus simplement et surtout beaucoup plus efficacement :

```
a <- vapply(1:4,function(x) mean(rnorm(x)),numeric(1))
a

## [1]  0.7700131  1.2346050 -0.5141222  0.2071179
```

Les boucles

En vrai, une boucle pourrait servir à ça :

```
vars <- colnames(iris)[sapply(iris,is.numeric)]
for ( ii in vars ) { iris[ii] <- scale(iris[ii]) }
```

Ce qui s'écrit plus simplement et surtout beaucoup plus efficacement :

```
vars <- colnames(iris)[sapply(iris,is.numeric)]
iris[,vars] <- apply(iris[,vars],2,scale)
```

Les boucles

Une utilisation justifiée des boucles.

```
for ( ww in c( function(x) {x^1}, function(x) {x^2}, function(x) {x^3}
```

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

En fait, non

```
power <- function(n,x) {x^n}
sapply(as.list(1:3),power,x=2)
```

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

Split...

La fonction *split* permet de découper une *data.frame* en fonction des modalités d'une variable et de récupérer une *list* en sortie avec pour chaque modalité la partie correspondante de la *data.frame*.

```
str(split(iris,factor(iris$Species)))
```

```
## List of 3
##  $ setosa    :'data.frame': 50 obs. of  5 variables:
##   ..$ Sepal.Length: num [1:50] -0.898 -1.139 -1.381 -1.501 -1.018 ..
##   ..$ Sepal.Width : num [1:50] 1.0156 -0.1315 0.3273 0.0979 1.245 ..
##   ..$ Petal.Length: num [1:50] -1.34 -1.34 -1.39 -1.28 -1.34 ...
##   ..$ Petal.Width : num [1:50] -1.31 -1.31 -1.31 -1.31 -1.31 ...
##   ..$ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1
##  $ versicolor:'data.frame': 50 obs. of  5 variables:
##   ..$ Sepal.Length: num [1:50] 1.397 0.672 1.276 -0.415 0.793 ...
##   ..$ Sepal.Width : num [1:50] 0.3273 0.3273 0.0979 -1.7375 -0.5904
##   ..$ Petal.Length: num [1:50] 0.534 0.42 0.647 0.137 0.477 ...
##   ..$ Petal.Width : num [1:50] 0.263 0.394 0.394 0.132 0.394 ...
##   ..$ Species     : Factor w/ 3 levels "setosa","versicolor",..: 2 2
##  $ virginica :'data.frame': 50 obs. of  5 variables:
##   ..$ Sepal.Length: num [1:50] 0.5515 -0.0523 1.5176 0.5515 0.793 ..
```

do.call

do.call est une fonction assez complexe. Elle permet notamment de définir l'environnement dans lequel exécuté une commande R. Toutefois elle a une utilisation simple à connaître. Elle permet en une ligne d'aggréger des résultats provenant d'une commande lapply.

```
stats <- function (x) { c(
  quantile( x$Sepal.Length,probs=c(0,0.25,0.5,0.75,1)),
  mean(x$Sepal.Length),
  sd(x$Sepal.Length) )
}
res <- lapply( split(iris,iris$Species), stats )
str(res)

## List of 3
##  $ setosa    : Named num [1:7] -1.8638 -1.26 -1.0184 -0.7769 -0.0523
##   ..- attr(*, "names")= chr [1:7] "0%" "25%" "50%" "75%" ...
##  $ versicolor: Named num [1:7] -1.1392 -0.2939 0.0684 0.5515 1.3968
##   ..- attr(*, "names")= chr [1:7] "0%" "25%" "50%" "75%" ...
##  $ virginica : Named num [1:7] -1.139 0.461 0.793 1.276 2.484 ...
##   ..- attr(*, "names")= chr [1:7] "0%" "25%" "50%" "75%"
```

do.call

```
do.call( rbind, res )
```

```
##                   0%        25%         50%
## setosa     -1.86378 -1.2599638 -1.01843718
## versicolor -1.13920 -0.2938574  0.06843254
## virginica  -1.13920  0.4609133  0.79301235
##                  75%        100%
## setosa     -0.7769106 -0.05233076 -1.0111914
## versicolor  0.5514857  1.39682886  0.1119073
## virginica   1.2760656  2.48369858  0.8992841
##
## setosa     0.4256782
## versicolor 0.6233453
## virginica  0.7679092
```

do.call

Si l'exemple peut être réalisé par exemple avec plyr, il est bonne illustration de *do.call*.
Plutôt qu'une matrice, si les résultats sont de types différents, on peut écrire dans certains cas :

```
do.call( data.frame, res )
```

lapply

La fonction *lapply* est une fonction dont l'utilisation doit croître avec l'expérience. Elle est centrale dans R et s'annonce de plus en plus indispensable car elle est à la base des fonctions de vectorisation des calculs dans R.
Par exemple, un jackknife, est très facile à réaliser avec une fonction *lapply*.

```
mm <- mean(iris[,"Sepal.Length"])
res <- sapply( as.list(1:nrow(iris)),
               function (x) {
                 (mean(iris[-x,"Sepal.Length"])-mm)^2
} )
vv <- sqrt(sum(as.numeric(res))/(nrow(iris)*(nrow(iris)-1)))
paste( "[", qt(0.025,nrow(iris)-1)*vv+mm,
       ":", qt(0.975,nrow(iris)-1)*vv+mm, "]" )

## [1] "[ -0.0010828241633901 7 : 0.0010828241633901 7 ]"
```

Calculs parallèles

La vectorisation est pour l'instant assez peu documenté. Il existe l'ouvrage de McCallum (2012) et quelques ressources dans les blogs sur R.
Sous les systèmes de type GNU/Linux, la vectorisation sur une même machine est d'une simplicité évangélique. Il suffit de charger le paquet *parallel* et de spécifier le nombre de processeurs à utiliser et d'utiliser la fonction *mclapply*.
Ce qui donne pratiquement le même code que précedemment pour un jackknife. . .

Calculs parallèles

```
mm <- mean(iris[,"Sepal.Length"])
res <- mclapply( as.list(1:nrow(iris)), function (x)
  (mean(iris[-x,"Sepal.Length"])-mm)^2,
  mc.cores=4
)

vv <- sqrt(sum(as.numeric(res))/(nrow(iris)*(nrow(iris)-1)))

paste( "[", qt(0.025,nrow(iris))*vv+mm, ":",
       qt(0.975,nrow(iris))*vv+mm, "]" )
```

Calculs parallèles

Avec ce mécanisme, 10 processus R vont être lancés en parallèle sur la machine. La mémoire nécessaire à chaque processus doit être disponible. Ce qui revient à demander à la machine 4 fois la mémoire nécessaire à l'éxecution du processus.
Le système utilise la commande *fork* du système d'exploitation. Par conséquent, chaque processus récupère l'environnement (variables) et paquets de la session courante. Pratique.
Dans le cas de simulation, il est nécessaire de bien lire l'aide du package pour obtenir selon ses besoins des seeds parallèles ou asynchrone.

Calculs parallèles

Dans le cas de machine Windows, cette méthode ne fonctionne pas en raison du fonctionnement de Windows (quelque soit sa version). Aussi dans ce cas et pour faire du calcul parallèle en gérant plus finement les ressources matériels et plusieurs ordinateurs quelque soit leur système d'exploitation, il est nécessaire de passer plutôt par l'utilisation des framework SNOW et MPI par exemple. L'utilisation est plus délicate car l'utilisateur doit notamment indiquer quelles variables, quels paquets, ... doivent être injectés dans les processus avant le lancement du calcul.

Calculs parallèles

Une vue entière est dédiée au problème des calculs lourds. . .

High-Performance and Parallel Computing with R

Lancement d'un script automatiquement

Pour lancer un script automatiquement, on peut le faire dans un fichier *batch*, c'est-à-dire un petit executable qui se termine en *.bat* sous *Windows*.
Il est conseillé de mettre le chemin de R dans le PATH *Windows* pour ne pas avoir à taper le chemin complet d'accès à R.
On peut ainsi appeler un script :

```
R -f Monscript.R
```

Lancement d'un script automatiquement

Mais R a une commande spécialement conçues pour réaliser des opérations depuis des fichiers exécutables...

```
R CMD BATCH Monscript.R
```

Un fichier *.Rout* est généré automatiquement et contient tout ce qui est apparu dans la console.

source

La fonction *source* permet d'exécuter le contenu d'un script depuis un autre script.
Cela permet par exemple de stocker des fonctions génériques puis de les rappeler en suite sans faire de paquets...

```
source("MesFonctions.R")
monbarplot(iris$Species)
```

Les règles de rédaction des scripts

R est un langage de programmation...
Pour la relecture et la lisibilité du code penser à commenter et à indenter !

Les règles de rédaction des scripts

Il est souvent plus simple d'utiliser un éditeur de texte tel que *emacs* ou *notepad++* pour profiter de la coloration syntaxique puis de copier-coller dans la console R.
ou *RStudio*.