

# Progammmation R

Ibrahima Sy

Institut Supérieure de Finance

10/8/2021

**Les fonctions**

**Définir ces propres fonctions**

**Structures de contrôle**

# **Les fonctions**

# Appel de fonctions

- ▶ Il n'y a pas de limite pratique quant au nombre **d'arguments** que peut avoir une fonction.
- ▶ Les arguments d'une fonction peuvent être spécifiés selon **l'ordre établi dans la définition de la fonction**.
- ▶ Cependant, il est beaucoup plus prudent et fortement recommandé de spécifier les arguments par leur nom, avec une construction de la forme `nom = valeur`, surtout après les deux ou trois premiers arguments.
- ▶ L'ordre des arguments est important ; il est donc nécessaire de les nommer s'ils ne sont pas appelés dans l'ordre.
- ▶ Certains arguments ont une valeur par défaut qui sera utilisée si l'argument n'est pas spécifié dans l'appel de la fonction.

# Exemple

```
matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)
```

- ▶ La fonction compte cinq arguments : data, nrow, ncol, byrow et dimnames.
- ▶ Ici, chaque argument a une valeur par défaut (ce n'est pas toujours le cas).
- ▶ Ainsi, un appel à matrix sans argument résulte en une matrice  $1 \times 1$  remplie par colonne (sans importance, ici) de l'objet NA et dont les dimensions sont dépourvues d'étiquettes

```
matrix()
```

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

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

# Appels de fonctions

Appel plus élaboré utilisant tous les arguments. Le premier argument est rarement nommé :

```
matrix(1:6, nrow = 2, ncol = 3, byrow = TRUE, dimnames = list(c("Gauche", "Droit"), c("Rouge", "Vert", "Bleu")))
```

##	Rouge	Vert	Bleu
## Gauche	1	2	3
## Droit	4	5	6

# Quelques fonctions utiles

- ▶ Le langage **R** compte un très grand nombre de fonctions internes.
- ▶ Cette section en présente quelques-unes seulement, les fonctions de base les plus souvent utilisées pour programmer en **R** et pour manipuler des données.

# Fonctions de manipulation de vecteurs

- ▶ **seq** : génération de suites de nombres
- ▶ **seq\_len** : version plus rapide de seq pour générer la suite des nombres de 1 à la valeur de l'argument
- ▶ **rep**: répétition de valeurs ou de vecteurs
- ▶ **sort**: tri en ordre croissant ou décroissant
- ▶ **rank**: rang des éléments d'un vecteur dans l'ordre croissant ou décroissant



# Fonctions de manipulation de vecteurs

- ▶ **rev** : renverser un vecteur
- ▶ **head** : extraction des premiers éléments d'un vecteur ( $n > 0$ ) ou suppression des derniers ( $n < 0$ )
- ▶ **tail** : extraction des derniers éléments d'un vecteur ( $n > 0$ ) ou suppression des premiers ( $n < 0$ )
- ▶ **unique** : extraction des éléments différents d'un vecteur

# Recherche d'éléments dans un vecteur

Soit le vecteur  $x$  suivant :

```
x <- c(4, -1, 2, -3, 6)
```

- ▶ `which` : positions des valeurs TRUE dans un vecteur booléen

```
which(x < 0)
```

- ▶ `which.min` : position du minimum dans un vecteur

```
which.min(x)
```

- ▶ `which.max` : position du maximum dans un vecteur

```
which.max(x)
```

- ▶ `match` : position de la première occurrence d'un élément dans un vecteur

```
match(2, x)
```

# Arrondi

Soit le vecteur x suivant :

```
x <- c(-3.6800000, -0.6666667, 3.1415927, 0.3333333, 2.5200000)
```

- ▶ **round** : arrondi à un nombre défini de décimales (par défaut 0)
- ▶ **floor** : plus grand entier inférieur ou égal à l'argument
- ▶ **ceiling** : plus petit entier supérieur ou égal à l'argument
- ▶ **trunc**: troncature vers zéro ; différent de floor pour les nombres négatifs

# Sommaires et statistiques descriptives

- ▶ **sum, prod** : somme et produit des éléments d'un vecteur
- ▶ **diff** : différences entre les éléments d'un vecteur
- ▶ **mean** : moyenne arithmétique (et moyenne tronquée avec l'argument trim)
- ▶ **var, sd** : variance et écart type

# Sommaires et statistiques descriptives

- ▶ **min, max** : minimum et maximum d'un vecteur
- ▶ **range** : vecteur contenant le minimum et le maximum d'un vecteur
- ▶ **median** : médiane empirique
- ▶ **quantile** : quantiles empiriques
- ▶ **summary** : statistiques descriptives d'un échantillon

# Sommaires cumulatifs et comparaisons élément par élément

- ▶ **cumsum, cumprod** : somme et produit cumulatif d'un vecteur
- ▶ **cummin, cummax** : minimum et maximum cumulatif
- ▶ **pmin, pmax** : minimum et maximum élément par élément (en parallèle) entre deux vecteurs ou plus

# Opérations sur les matrices

- ▶ **nrow, ncol** : nombre de lignes et de colonnes d'une matrice
- ▶ **rowSums, colSums** : sommes par ligne et par colonne, respectivement, des éléments d'une matrice
- ▶ **rowMeans, colMeans** : moyennes par ligne et par colonne, respectivement, des éléments d'une matrice
- ▶ **t**: transposée

# Opérations sur les matrices

► **det**: déterminant

► **solve** :

1. avec un seul argument (une matrice carrée) : inverse d'une matrice ;
2. avec deux arguments (une matrice carrée et un vecteur) : solution du système d'équations linéaires  $Ax = b$

► **diag** :

1. avec une **matrice** en argument : **diagonale** de la matrice ;
2. avec un **vecteur** en argument : **matrice diagonale** formée avec le vecteur ;
3. avec un **scalaire**  $p$  en argument : **matrice identité**  $p \times p$



**Définir ces propres fonctions**

# Syntaxe

- ▶ La définition d'une nouvelle fonction suit la syntaxe suivante : ‘

```
nomFonction <- function(arguments)
{
  traitement
}
```

- ▶ Avec
  - ▶ nomFonction : le nom que l'on décide de donner à la fonction,
  - ▶ arguments : les paramètres de la fonction, et
  - ▶ traitement : le corps de la fonction.

- ▶ Comme on peut le remarquer, on utilise le symbole d'assignation `<-` : les fonctions sont des **objets**.
- ▶ L'appel de la fonction aura la syntaxe suivante :
  - ▶ Il suffit donc de rajouter des parenthèses au nom de la fonction pour l'appeler.
  - ▶ En effet, `nomFonction` désigne l'objet R qui contient la fonction qui est appelée à l'aide de l'expression `nomFonction()`

## Exemple

Par exemple, si on souhaite définir la fonction qui calcule le carré d'un nombre, voici ce que l'on peut écrire :

```
carre <- function(x)
{
  x^2
}
```

```
## Le carre de 2
carre(2)
```

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

# Structure d'une fonction

- ▶ Les fonctions en R, excepté les fonctions primitives du package base, sont composées de trois parties :
  - ▶ une **liste de paramètres**
  - ▶ un **corps**, contenant du code exécuté lors de l'appel à la fonction ;
  - ▶ un **environnement**, qui définit l'endroit où sont stockées les variables.
- ▶ On peut accéder à ces trois parties (et les modifier) avec les fonctions `formals()` pour les paramètres, `body()` pour le corps et `environment()` pour l'environnement.

# Structure d'une fonction : le corps

- ▶ Dans le cas le plus simple, le corps d'une fonction est constitué d'une seule instruction.
- ▶ Si on désire en écrire plusieurs, il est nécessaire de les entourer par des accolades, pour réaliser un regroupement.
- ▶ Le résultat est la valeur de la dernière commande contenue dans le corps de la fonction.
- ▶ Si on souhaite retourner une valeur autre part qu'à la dernière ligne, il faut utiliser la fonction `return()`
  - ▶ utile lorsque l'on emploie des conditions, ou pour prévenir d'une erreur

## Structure d'une fonction : le corps

- Il est possible de retourner une liste, pouvant contenir autant d'objet que l'on souhaite.

```
stat_des <- function(x){  
  list(moyenne=mean(x), ecart_type=sd(x))  
}
```

```
x <- runif(10)  
stat_des(x)
```

```
## $moyenne  
## [1] 0.4073692  
##  
## $ecart_type  
## [1] 0.3322359
```

## Structure d'une fonction : le corps

- Il est également possible de ne pas afficher dans la console le résultat de l'appel à une fonction à l'aide de la fonction `invisible()`.

```
stat_des_2 <- function(x) {  
  invisible(list(moyenne = mean(x), ecart_type = sd(x)))  
}  
x<-runif(10)  
stat_des_2(x)  
str(stat_des_2(x))
```

```
## List of 2  
## $ moyenne      : num 0.334  
## $ ecart_type: num 0.282
```



# Structure d'une fonction : le corps

- ▶ On peut afficher malgré tout le résultat d'une fonction **retournant un résultat invisible** en ayant recours aux parenthèses.
- ▶ Lorsque la dernière instruction est une assignation, nous sommes dans le cas d'un **résultat invisible**.

# Structure d'une fonction : les paramètres d'une fonction

- ▶ Dans l'exemple de la fonction `carre()` que nous avons créée, nous avons renseigné un seul paramètre, appelé `x`.
- ▶ Si la fonction que l'on souhaite créer nécessite **plusieurs paramètres**, il faut les séparer par une virgule.
- ▶ Considérons par exemple le problème suivant.
  - ▶ Nous disposons d'une fonction de production  $Y(L, K, M)$ , qui dépend du nombre de travailleurs  $L$  et de la quantité de capital  $K$ , et du matériel  $M$ , telle que  $Y(L, K, M) = L^{0.3} K^{0.5} M^2$ .

```
production <- function(l, k, m )  
{  
  l^(0.3) * k^(0.5) * m^(0.2)  
}
```

# Structure d'une fonction : les paramètres d'une fonction

- ▶ On peut, lors de la définition de la fonction, choisir de donner une valeur par défaut aux paramètres.
- ▶ On parle de paramètre **formel** pour désigner les paramètres de la fonction (les variables utilisées dans le corps de la fonction)
- ▶ et de paramètre **effectif** pour désigner la valeur que l'on souhaite donner au paramètre formel.
- ▶ Pour définir la valeur à donner à un paramètre formel, on utilise le symbole d'égalité.
- ▶ Lors de l'appel de la fonction, si l'utilisateur ne définit pas explicitement une valeur, celle par défaut sera affectée.

```
# On propose de définir la valeur du capital à 42 par défaut  
production_2 <- function(l, m, k = 42) {  
  l^(0.3) * k^(0.5) * m^(0.2)  
}
```

# Structure d'une fonction : les paramètres d'une fonction

- ▶ Dans l'exemple ci-avant, le paramètre à qui nous avons donné une valeur est placé en dernier.
- ▶ Ce n'est pas obligatoire, mais plus pratique, si le but recherché est de ne pas avoir à saisir le paramètre effectif lors de l'appel de la fonction.
- ▶ De plus, si l'utilisateur ne nomme pas les paramètres lors de l'appel, des problèmes liés à l'ordre peuvent apparaître...

```
production_3 <- function(l, k = 42, m) l^(0.3) * k^(0.5) *  
# production_3(l = 42, m = 40)  
# production_3(42, 40)
```

# Structure d'une fonction : les paramètres d'une fonction

## Le paramètre spécial ...

- ▶ Le paramètre ... que l'on peut voir dans certaines fonctions sert à indiquer que la fonction peut admettre d'autres paramètres que ceux qui ont été définis.
- ▶ Cela sert à, dans la plupart des cas, à passer un paramètre à une autre fonction contenue dans le corps de la fonction que l'on appelle.
- ▶ Attention toutefois, l'utilisation de ... peut induire des soucis. En effet, un paramètre mal écrit sera passé à ... et il n'y aura pas d'erreur de retournée.
- ▶ Par ailleurs, tous les paramètres placés après ... doivent être complètement nommés, pas abrégés.

# Portée des fonctions

- ▶ Lorsqu'une fonction est appelée, le corps de cette fonction est interprété.
- ▶ Les variables ayant été définies dans le corps de la fonction ne vivent qu'à l'intérieur de celle-ci à moins d'avoir spécifié le contraire.
- ▶ On parle alors de **portée** des variable.
- ▶ Ainsis, une variable ayant une portée locale; c'est-à-dire vivant uniquement à l'intérieur du corps de la fonction, peut avoir le même nom qu'une variable globale ; c'est à dire définie dans l'espace de travail de la session, sans pour autant désigner le même objet, ou écraser cet objet.

# Portée des fonctions

```
# Définition d'une variable globale
valeur <- 1
# Définition d'une variable locale à la fonction f
f <- function(x){
  valeur <- 2
  nouvelle_valeur <- 3
  print(paste0("valeur vaut : ",valeur))
  print(paste0("nouvelle_valeur vaut : ",nouvelle_valeur))
  x + valeur
}
```

# Portée des fonctions

```
f(3)
```

```
## [1] "valeur vaut : 2"
```

```
## [1] "nouvelle_valeur vaut : 3"
```

```
## [1] 5
```

```
## [1] "valeur vaut : 2"
```

```
## [1] "nouvelle_valeur vaut : 2"
```

```
## [1] 5
```

```
# valeur n'a pas été modifiée
```

```
valeur
```

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

```
# nouvelle_valeur n'existe pas en dehors de f()
```

```
##nouvelle_valeur
```

```
## Error in eval(expr, envir, enclos): objet 'nouvelle_valeur' n'existe pas
```



# Portée des fonctions

- ▶ Il semble important de connaître quelques principes à propos de la portée des variables.
- ▶ Les variables sont définies dans des environnements, qui sont imbriqués les uns dans les autres. Si une variable n'est pas définie dans le corps d'une fonction, R ira chercher dans un environnement parent.

```
valeur <- 1  
f <- function(x){ x + valeur  
}  
f(2)
```

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

# Portée des fonctions

- ▶ Si on définit une fonction à l'intérieur d'une autre fonction, et qu'on appelle une variable non définie dans le corps de cette fonction, R ira chercher dans l'environnement directement supérieur.
- ▶ S'il ne trouve pas, il ira chercher dans l'environnement encore supérieur, et ainsi de suite.

# Portée des fonctions

*# La variable valeur n'est pas définie dans g(). R va alors*

```
valeur <- 1
f <- function(){
  valeur <- 2
  g <- function(x){ x + valeur
    }
  g(2)
}
f()
```

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

# Portée des fonctions

```
# La variable valeur n'est définie ni dans g() ni dans f()  
f <- function(){  
  g <- function(x){  
    x + valeur  
  }  
  g(2)  
}  
f()
```

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

# Portée des fonctions

Si on définit une variable dans le corps d'une fonction et que l'on souhaite qu'elle soit accessible dans l'environnement **global**, on peut utiliser le symbole «`<-`», ou bien la fonction **assign**

```
f <- function(x) {  
  x <<- x + 1  
}  
f(1)  
x
```

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

```
# En utilisant assign
```

# Portée des fonctions

```
rm(x)
f <- function(x) {
  # envir = .GlobalEnv signifie que l'on
  # veut définir dans l'environnement global
  assign(x = "x", value = x + 1, envir = .GlobalEnv)
}
f(4)
x
```

```
## [1] 5
```

# Exercices

1. Créer une fonction nommée `somme_n_entiers` qui retourne la somme des  $n$  premiers entiers. Son seul paramètre sera  $n$ ;
2. Utiliser la fonction `somme_n_entiers()` pour calculer la somme des 100 premiers entiers ;
3. Créer et appeler une fonction `add1` qui prend en paramètres deux nombres et retourne leur somme
4. Créer et appeler une fonction `produit1` qui prend en paramètre deux nombres et retourne leur produit
5. Créer et appeler les fonctions `add2` et `produit2` qui reprennent respectivement `add1` et `produit1` mais en considérant des valeurs par défaut adéquates
6. Créer et appeler une fonction `cercle` qui prend en paramètre un rayon d'un cercle, calcule et retourne les dimensions de ce cercle sous forme de liste.

## **Structures de contrôle**




# Structures de contrôle

- ▶ Comme pratiquement tout langage de programmation, R offre des structures de contrôles.
- ▶ On pourrait définir les structures de contrôle ainsi : il s'agit d'instructions particulières qui contrôlent l'ordre dans lequel les autres instructions du programme informatique sont exécutées.
- ▶ Ainsi, il est possible de demander à R d'exécuter les instructions d'un programme autrement que séquentiellement.
- ▶ Les deux structures de contrôle de base sont présentées ici :
  - ▶ les alternatives : **structures conditionnelles**,
  - ▶ les boucles : **structures itératives**.

# Structures de contrôle

## Branchement conditionnel « if »

Condition est très souvent une  
opération de comparaison



```
if (condition) {  
    ... instruction(s)  
} else  
{  
    ... instruction(s)  
}
```

## Remarque

1. Attention aux accolades : **else** doit être sur la même ligne que **}**
2. La partie **else** est facultative

## Exemple

```
# Simulation du lancer d'une pièce de monnaie
lancer <- sample(x = c("Pile", "Face"), size = 1)

if (lancer == "Pile") # sans accolades
  print("Je gagne!")
# ou encore
if (lancer == "Pile") { # avec accolades
  print("Je gagne!")
}
```

# Structures de controle

- ▶ Lorsque l'on souhaite exécuter certaines instructions si la condition est fausse, il faut ajouter un **else** à l'alternative, suivi des instructions en questions.
- ▶ Il est alors considéré une bonne pratique de toujours encadrer les instructions d'accolades, même s'il n'y a qu'une seule instruction, de façon à retrouver le mot-clé **else** précédé de **}** et suivi de **{**.

```
if (lancer == "Pile"){  
    print("Je gagne!")  
} else{  
    print("Je perds...")  
}
```

- ▶ On peut aussi imbriquer plusieurs `if. . . else.`

# Exemple

```
#demander le prix ht
print("saisir prix ht")
ht <- scan()
#type de produit
print("type de produit : 1 - luxe, autre - normal")
typprod <- scan()
#calcul du prix ttc
if (typprod == 1)
{
  tva <- 0.33
} else
{
  tva <- 0.196
}
#calcul
ttc <- ht * (1 + tva)
#affichage
print(paste("prix ttc = ", as.character(ttc)))
```

## Le cas particulier de ifelse

- ▶ On peut simplifier la structure de branchement, surtout quand il s'agit d'effectuer une simple affectation
- ▶ Il fonctionne de 2 manières différentes :

1. il renvoie une valeur selon que la condition est vraie ou pas

```
`tva <- ifelse(typprod == 1, 0.33, 0.196)`
```

2. on procède à une opération simple suivie d'une affectation dans les deuxièmes et troisièmes paramètres de la fonction

```
`ifelse(typprod == 1, ttc <- ht * 1.33, ttc <- ht * 1.1
```

## Distinction entre une structure if ... else et la fonction ifelse

- ▶ Sous sa forme condensée, une structure if ... else fait penser à un appel à la fonction ifelse.
- ▶ Quelles sont les différences entre les deux ?
- ▶ Premièrement, la condition dans une structure if ... else doit être une instruction qui retourne un seul TRUE ou un seul FALSE, pas un vecteur logique de longueur supérieure à 1.
- ▶ Si la condition est un vecteur logique de longueur supérieure à 1, seul le premier élément est utilisé et un avertissement est affiché comme dans l'exemple suivant

```
x <- 1:10  
xmodif <- if(x > 2.5 & x < 7.5) 5 else x
```

```
## Warning in if (x > 2.5 & x < 7.5) 5 else x: the condition  
## only the first element will be used
```

## Distinction entre une structure if ... else et la fonction ifelse

- ▶ Cependant, dans cet exemple, la structure de contrôle if ... else n'est probablement pas ce que nous voulions utiliser.
- ▶ Si notre but est de vérifier pour chaque élément de x si la valeur est comprise entre 2.5 et 7.5, si c'est le cas retourner la valeur 5, sinon retourner l'élément de x inchangé, alors c'est la fonction ifelse que nous devrions utiliser.

```
xmodif <- ifelse(x > 2.5 & x < 7.5, 5, x)
xmodif
```

```
## [1] 1 2 5 5 5 5 5 8 9 10
```



## Distinction entre une structure `if ... else` et la fonction `ifelse`

- ▶ `ifelse` est une fonction qui agit de façon vectorielle. Elle teste une condition sur tous les éléments d'un objet et retourne une valeur en fonction du résultat. La dimension de la sortie d'un `ifelse` est la même que la dimension du premier argument qu'on lui a fourni.
- ▶ Ainsi, la fonction `ifelse` n'est pas une structure de contrôle.

# Boucles

- ▶ Les boucles ont pour but de répéter des instructions à plusieurs reprises, de les itérer.
- ▶ Parfois, on connaît d'avance le nombre d'itérations à effectuer.
- ▶ D'autres fois, on ne connaît pas d'avance ce nombre. Il dépend d'une condition à rencontrer.

# Boucles for

- ▶ Si on sait d'avance combien d'itérations doivent être réalisées, on utilise une boucle for.

```
for (valeur in sequence) {  
    instructions  
}
```

- ▶ Ce type de boucle débute par le mot clé for, suivi d'une parenthèse dans laquelle on insère
  - ▶ d'abord un nom quelconque, représenté par valeur dans la syntaxe générale ;
  - ▶ suivi du mot-clé in ;
  - ▶ puis d'une instruction retournant un vecteur contenant l'ensemble des valeurs sur lesquelles itérer, représenté par sequence dans la syntaxe générale.

# Boucles for

- ▶ Ensuite, viennent la ou les instructions à répéter. S'il y a plus d'une instruction à répéter, les accolades sont nécessaires pour les encadrer.

# Boucle for

```
for (valeur in sequence) {  
    instructions  
}
```

**Séquence** peut être stockée dans un vecteur Le pas peut ne pas être régulier La séquence n'est même pas forcément monotone Bref, l'outil est très (trop) souple -> d'où sa lenteur

## Remarque :

On peut « casser » la boucle avec **break**

# Boucle for

- ▶ La boucle effectuera autant de répétitions que la longueur du vecteur ensemble.
- ▶ À la première itération de la boucle, valeur contiendra le premier élément du vecteur ensemble,
- ▶ à la deuxième itération valeur contiendra le deuxième élément d'ensemble, etc.
- ▶ De façon générale, on peut dire qu'à l'itération `i` on a que `valeur = sequence[i]`, pour `i` allant de 1 à `length(ensemble)`.

## Example 1

```
for (lettre in LETTERS){  
  cat(lettre, " ")  
}
```

```
## A B C D E F G H I J K L M N O P Q R S
```

```
for (i in 1:length(LETTERS)){  
  cat(LETTERS[i], " ")  
}
```

```
## A B C D E F G H I J K L M N O P Q R S
```

## Exemple 2

```
# A faire : somme S = SUM_i i^2
# saisie n
n <- scan()

# initialisation de la somme
s <- 0.0
# boucle for
for (i in 1:n){
  s <- s+i^2
}

# affichage des resultats
print("la somme = ")
print(s)
```



# Boucles while ou repeat

- ▶ Parfois, on ne sait pas d'avance combien d'itérations il y a à effectuer. Le nombre d'itérations dépend d'une condition à rencontre. Les boucles R while et repeat sont de ce type.
- ▶ Syntaxe de la boucle while :

```
while (condition) {  
  instructions  
}
```

# Boucles while ou repeat

- ▶ Syntaxe de la boucle **repeat**

```
repeat {  
  instructions  
  if (condition) break  
}
```

- ▶ Un des intérêts d'une boucle repeat est de tester la condition après avoir exécuté les instructions et non avant comme dans une boucle while.
- ▶ Dans une boucle repeat, le mot-clé break doit être utilisé pour mettre fin aux itérations.
- ▶ La condition encore une fois doit être une instruction qui retourne un seul TRUE ou un seul FALSE.

# Exemple

```
# A faire : somme  $S = \sum_i i^2$ 
# saisie n
n <- scan()

# initialisation de la somme et de l'indice
s <- 0.0
i <- 1
# boucle tant que
while(i<n){
  s <- s + i^2
  i <- i + 1
}

# affichage des resultats
print("la somme = ")
print(s)
```

# Mots-clé break et next

- ▶ Deux mots-clés existent pour contrôler l'exécution des instructions à l'intérieur d'une boucle :
  - ▶ **break** : pour terminer complètement l'exécution de la boucle (les itérations restantes ne sont pas effectuées).
  - ▶ **next** : pour terminer immédiatement une itération (sans exécuter les instructions après le mot-clé next) et reprendre l'exécution de la boucle à la prochaine itération.
- ▶ C'est deux mot-clés sont pratiquement toujours utilisé dans une structure if.
- ▶ Le mot-clé break a déjà été illustré dans une boucle repeat. Notons cependant qu'on peut l'utiliser dans une boucle de n'importe quel type.

# Commentaires

- ▶ Un commentaire est une ligne ou une portion de ligne qui sera ignorée par **R**.
- ▶ Ceci signifie qu'on peut y écrire ce qu'on veut, et qu'on va les utiliser pour ajouter tout un tas de commentaires à notre code permettant de décrire les différentes étapes du travail, les choses à se rappeler, les questions en suspens, etc.
- ▶ Un commentaire sous R commence par un ou plusieurs symboles **#**.
  - ▶ Tout ce qui suit ce symbole jusqu'à la fin de la ligne est considéré comme un commentaire. On peut créer une ligne entière de commentaire, par exemple en la faisant débiter par **##** :

**##** Attention au nombre de non réponses !

- ▶ On peut aussi créer des commentaires pour une ligne en cours :  
`x <- 2 # On met 2 dans x, parce qu'il le vaut bien`