Introduction au logiciel R

Manipuler des données

Laurence Viry

MaiMoSiNE - Collège des écoles doctorales Grenoble

7-16 Février 2017

Plan du cours

Expressions et blocs d'expressions

- Un **bloc d'expressions** une suite d'expressions constituées d'opérateurs et d'autres objets (variables, constantes,fonctions) **encadréees 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.
- **Le bloc entier est lui-même une expression** dont la valeur sera **la dernière expression éevaluée dans le bloc**.

Blocs d'expressions

```
> monbloc <- { a <- 1:10
+ somme <- sum(a)}
> a
 [1]  1  2  3  4  5  6  7  8  9 10
> somme
[1] 55
> monbloc
[1] 55
```

La condition (if,else)

if (cond) expr: la commande est exécutée si et seulement si la condition est vraie.

```
> i <- 1
> if (i<3) i <- i+1
> print(i)
```

```
[1] 2
```

if (cond) expr1 else expr2: une autre condition peut être ajoutée à la suite du if

```
> x <- 7
> if( x %% 2 == 0){
+ parit <- "pair"
+ } else {
+ parit = "impair"
+ }
> parit
```

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

Attention, l'ordre else doit être sur la même ligne que la parenthèse fermante "}"

switch(expr,...)

Pour faire **des choix multiples**.

```
> x <- 1
> switch(x,"un","deux","trois","quatre")
[1] "un"
> x <-4 ; switch(x,"un","deux","trois","quatre")
[1] "quatre"
> x <-5 ; switch(x,"un","deux","trois","quatre")
```

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

```
> chaine="trois"
> switch(chaine,un = 1,deux = 2,trois = 3,quatre = 4,"je ne sais pas
[1] 3
> chaine="cent"
> switch(chaine,un = 1,deux = 2,trois = 3,quatre = 4,"je ne sais pas
[1] "je ne sais pas"
```

ifelse(test, oui, non)

Un version **vectorisée** très puissante:

```
> x <- rnorm(10)
> x
 [1] -0.528113820  0.562950284 -0.514657615  0.447984322 -0.48172607
 [6] -1.185417322 -0.008104181  0.560554665 -1.021918362 -0.90857696

> ifelse(x > 0, "positif", "negatif")
 [1] "negatif" "positif" "negatif" "positif" "negatif" "negatif" "ne
 [8] "positif" "negatif" "negatif"
```

Les boucles for

Les **boucles** classiques de la programmation sont disponibles sous R.

```
> for (i in 1:99) print(i) # affiche les entiers de 1 a 99
> for (i in seq(1,99,by=2)) print(i)  # affiche de 2 en 2
```

La méthode se généralise à **un vecteur quelconque**:

```
> vecteur <- c("lundi","mardi","mercredi","jeudi")
> for (i in vecteur) print(i)
```

```
[1] "lundi"
[1] "mardi"
[1] "mercredi"
[1] "jeudi"
```

Il y a souvent **plusieurs instructions par itération**, elles sont encadrées par {}:

```
> for (i in seq(1,10,by=2))  {
+ i*2
+ i^2
+
+ }
```

Alternative boucle for sous R

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
```

Approche sous R:

```
> x[x < 0] <- -1
```

La condition while et commande repeat

while: tant que la condition est vraie on répète l'expression :

```
> i <- 1
> while (i <= 3) {
+   print(i); i <- i+1
+ }
```

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

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

```
> i <- 1
> repeat{
+ print(i)
+ i <- i + 1
+ if(i > 3) break
+ }
```

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

next

On peut **sauter un tour dans une boucle**.

Pour mettre à zéro tous les éléments d'une matrice sauf les éléments diagonaux :

```
> mat <- matrix(1:9,ncol=3,nrow=3)
> for(i in 1:3){
+ for(j in 1:3){
+ if(i == j) next
+ mat[i,j] <- 0
+ }
+ }
```

lapply() et sapply()

- **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
> a <- lapply(maliste, f)
> class(a)
```

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

lapply() retourne une liste.

- **sapply**() fait la même chose et retourne **un résultat de type vecteur** :

```
> b <-sapply(maliste, f)
> b
```

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

```
> class(b)
```

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

tapply()

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

```
> data("iris")
> summary(iris)
```

```
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
 Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
 Median :5.800   Median :3.000   Median :4.350   Median :1.300
 Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
 Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
       Species
 setosa    :50
 versicolor:50
 virginica :50
```

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

```
    setosa versicolor  virginica
     5.006      5.936      6.588
```

apply()

La fonction **apply()** permet d'appliquer une fonction aux lignes (MARGIN=1) ou aux colonnes (MARGIN=2) d'une matrice :

apply <- **function** (X, MARGIN, FUN, ...)

```
> mat <- matrix(1:12, 3, 4)
> apply(mat,MARGIN=1,sum) # somme des lignes
[1] 22 26 30
> apply(mat,MARGIN=2,sum) # somme des colonnes
[1]  6 15 24 33
```

Les fonctions **colSums()** et **rowSums()** permettent d'obtenir le même résultat :

```
> colSums(mat)
[1]  6 15 24 33
> rowSums(mat)
[1] 22 26 30
```

apply: exemple d'application

On considère le jeu de données airquality (dataset)

```
> 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**.

Le choix est fait de **remplacer les valeurs manquantes par la moyenne de la variable**.

apply: exemple d'application

Utilisation des boucles for:

```
> 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: exemple d'application

Approche avec apply():

```
> 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 R.

Les fonctions

- Une **fonction** permet d'effectuer un certains nombre d'instructions R.
- Elle peut **dépendre d'arguments** fournis **en entrée**.
- Un argument fournit sous la forme **nom = valeur** permet de donner **une valeur par défaut** à cet argument.
- Elle fournit **un résultat unique en sortie** transmit par la fonction **return**.
- Par défaut, en l'absence d'appel à return, **le dernier résultat obtenu avant la sortie de la fonction** est retourné comme **résultat**.

Une nouvelle fonction est créée par une construction de la forme :

fun.name <- **function**(arglist) bloc d'instructions

```
> som <- function(n){
+   result <- sum(1:n)
+   return(result)
+ }
> som(3)  # appel à la fonction
[1] 6
```

fonction: exemple

Une fonction qui retourne ses arguments.

Des valeurs par défaut sont donnée aux paramètres a et b.

```
> 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 =
    panel.last = NULL, asp = NA, ...)
NULL
```

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)
}
```

L'argument ...

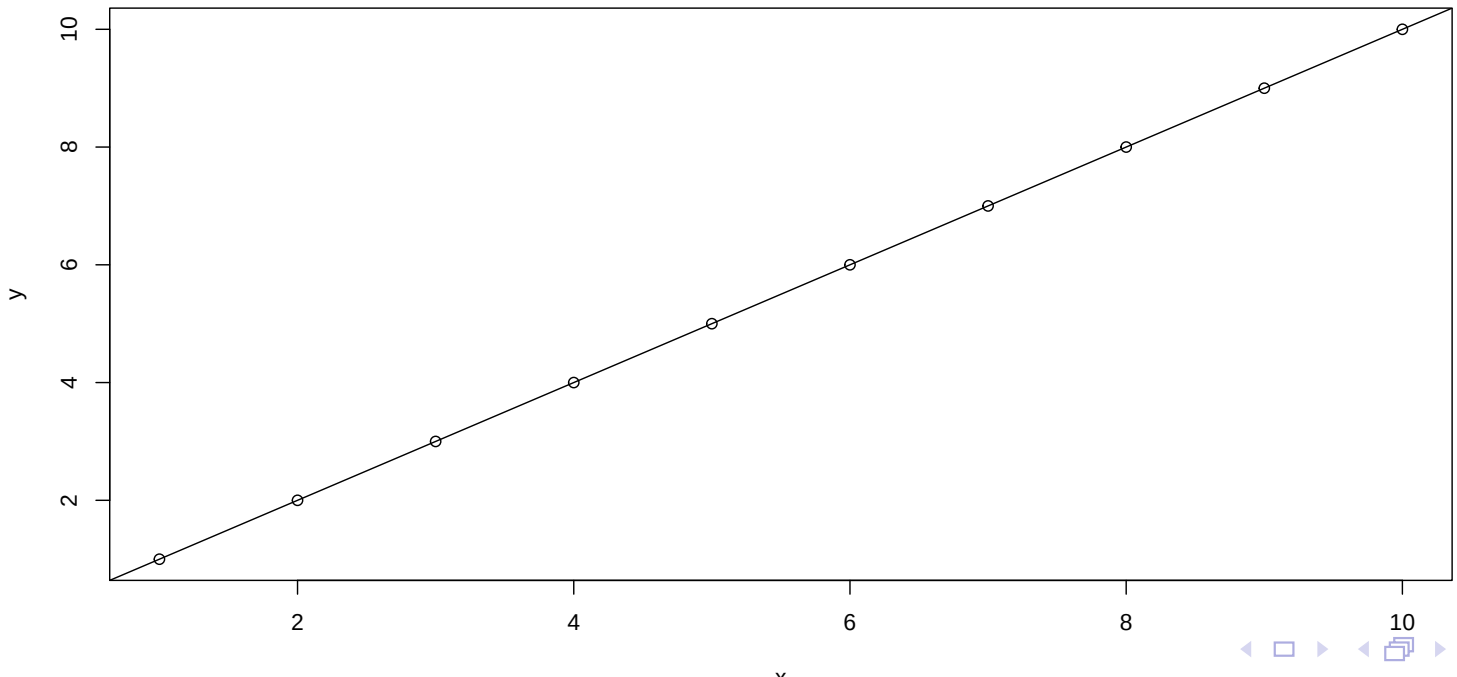
- L'argument **point-point-point** permet à une fonction d'accepter un **nombre quelconque d'arguments**.
- L'argument point-point-point (. . .) indique que la fonction accepte n'importe quoi d'autre comme argument. **Ce qu'il adviendra de ces arguments est déterminé par la fonction**.
- En général ils 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.

Utilisation de l'argument ...

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

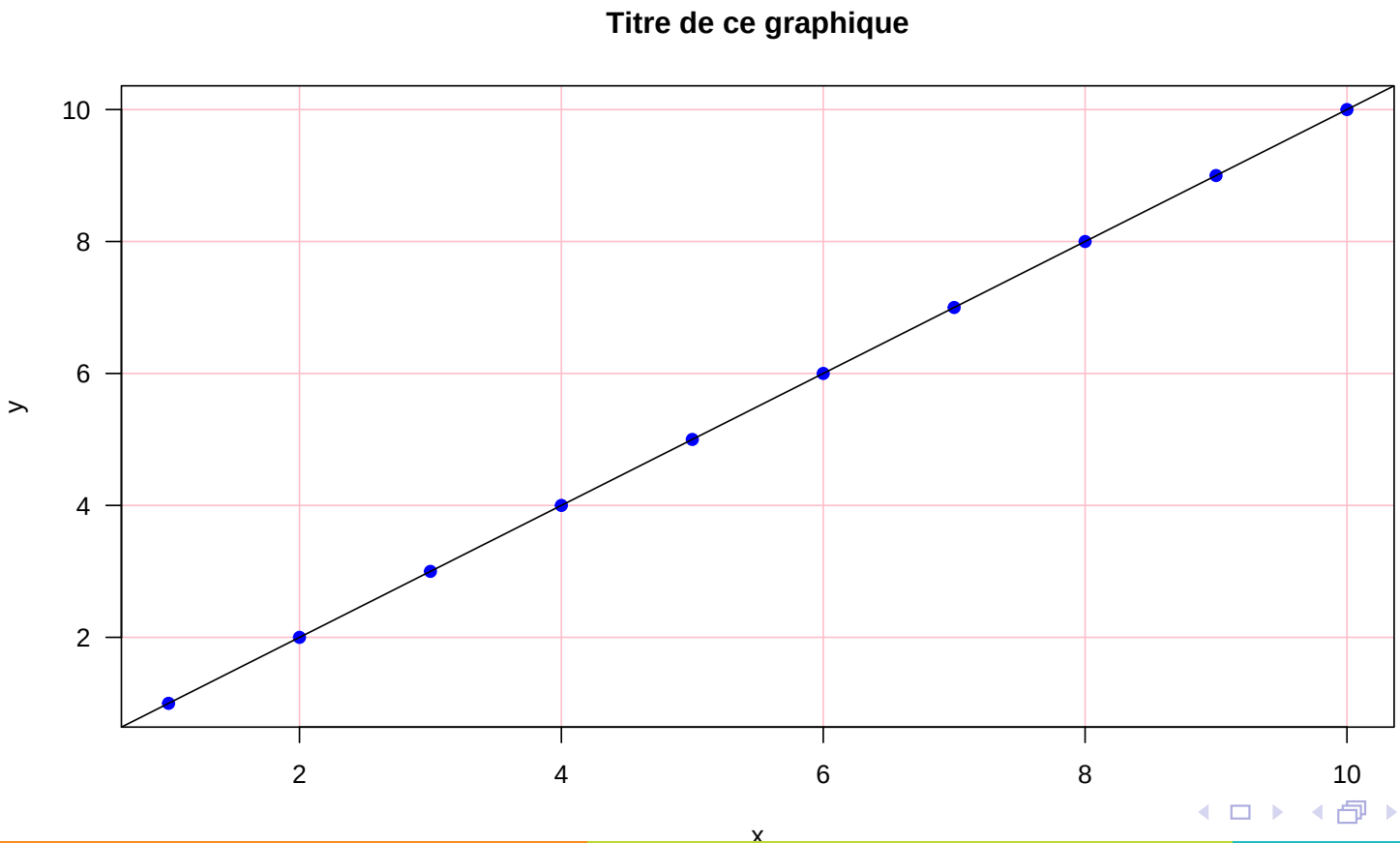
```
> f <- function(x, y, ...) {
+ plot.default(x = x, y = y, ...)
+ abline(coef = lm(y ~ x)$coef)
+ }
> f(1:10, 1:10)
```



Utilisation de l'argument ...

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"))
```



Fonction qui renvoie plusieurs objets

Le résultat sera fourni sous forme d'une liste.

Considérons une fonction avec **deux arguments en entrée**: **facteur** et **facteur2**, deux **variables qualitatives**.

Cette fonction (**mafonc**) renvoie le **tableau de contingence** et **le vecteur de caractères des niveaux** de facteur1 et facteurs2 pour lesquels l'effectif est conjointement nul.

```
> mafonc <- function(facteur1,facteur2) {
+   res1 <-table(facteur1,facteur2)  # tableau de contingence
+   selection <- which(res1 == 0,arr.ind=TRUE)
+   res2 <- matrix("",nrow=nrow(selection),ncol=2)
+   res2[,1] <- levels(facteur1)[selection[,1]]
+   res2[,2] <- levels(facteur2)[selection[,2]]
+   # deux objets à retourner, utilisation d'une liste
+   return(list(tab=res1,niveau=res2))
+ }
```

Fonction qui renvoie plusieurs objets: appel

```
> # appel de la fonction
> tension <- factor(c(rep("Faible",5),rep("Forte",5)))
> laine <- factor(c(rep("Mer",3),rep("Ang",3),rep("Tex",4)))
> #
> res <- mafonc(tension,laine)
> class(res)
[1] "list"
> res
$tab
        facteur2
facteur1 Ang Mer Tex
  Faible   2   3   0
  Forte    1   0   4

$niveau
     [,1]     [,2]
[1,] "Forte"  "Mer"
[2,] "Faible" "Tex"
```

Portée des variables

- A 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 définies 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**.
- 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.

Portée des variables

```
> mavariable <- 1
> #
> mafonction1 <- function() {
+ mavariable <- 5
+ print(mavariable)
+ }
> #
> mafonction1()

[1] 5

> mavariable

[1] 1

> mafonction2 <- function() {
+ print(mavariable)
+ }
> mafonction2()

[1] 1
```