Tests et exécutions conditionnelles, boucles, fonctions de boucles implicites. Traitement des chaînes de caractères.

Christophe Pouzat

Mathématiques Appliquées à Paris 5 (MAP5)

Université Paris-Descartes et CNRS UMR 8145

christophe.pouzat@parisdescartes.fr

Mercredi 17 avril 2013

Outline

Introduction

Conditionnelles

Boucles

Fonctions de boucles implicites

Traitement des chaînes de caractères

Topic

Introduction

Conditionnelles

Boucles

Fonctions de boucles implicites

Traitement des chaînes de caractères

Références

- Ce qui suit est essentiellement une traduction de diapos de Ross Ihaka (l'un des deux créateurs de R) associées à un cours intitulé Statistical Computing ;
- Une autre excellente référence pour ce cours et pour R en général est le bouquin de Pierre Lafaye de Micheaux, Rémy Drouilhet et Benoît Liquet « Le logiciel R. Maîtriser le langage. Effectuer des analyses statistiques. » (2011, Springer).

Rappels sur les expressions (1)

Un programme en R est constitué d'une suite d'**expressions**.
Ces expressions peuvent être « simples » comme :

```
(maVariable <- "Ma variable")
```

```
 [1] "Ma variable"
```

Elles peuvent aussi être « composées » lorsque plusieurs expressions simples sont **séparées** par un « ; » ou un saut de ligne et **regroupées** entre des accolades « { } » comme :

```
{ expr1; expr2; ...; exprN }
```

Rappels sur les expressions (2)

Toute expression dans R a une valeur et la valeur d'une expression composée est celle de la dernière expression simple qui la constitue :

```
x <- {10; 20}
x

 [1] 20
```

Affectation dans les expressions composées

Il est possible d'avoir des affectations (attributions de valeurs à des variables) dans les expressions composées. Les valeurs des variables ainsi affectées peuvent être utilisées dans d'autres expressions :

```
z <- { x <- 10; y <- x^2; x+y }

x

 [1] 10

y

 [1] 100

z

 [1] 110
```

Topic

Introduction

Conditionnelles

Boucles

Fonctions de boucles implicites

Traitement des chaînes de caractères

`if-then-else`

Avec `if-then-else` il est possible de choisir entre deux expressions (potentiellement composées) suivant la valeur d'une condition (expression logique) :

```
if (condition) expr.si.vrai else expr.si.faux
```

Si **condition** est vraie (TRUE), **expr.si.vrai** est évaluée, sinon **expr.si.faux** est évaluée.

Remarques

- le premier élément de **condition** est le seul à être pris en compte ;
- la valeur de l'expression complète est la valeur de celle des deux expressions qui est évaluée.

`if-then-else` : exemples

L'expression :

```
if (x > 0) y <- sqrt(x) else y <- -sqrt(-x)
```

constitue un exemple de construction `if-then-else` qui semblera familière à un programmeur `Pascal`, `Fortran`, `C`, `C++` ou `Java`.
Cette expression serait néanmoins typiquement écrite en `R` :

```
y <- if (x > 0) sqrt(x) else -sqrt(-x)
```

ce qui sera naturel pour un programmeur `Lisp` ou `Algol`.

`if-then-else` : forme simplifiée

Il y a une forme simplifiée de l'expression `if-then-else` qui sera employée lorsqu'il n'y a pas d'**expr.si.faux**. Cette expression a la forme générale :

```
if (condition) expr.si.vrai
```

et elle est complètement équivalente à :

```
if (condition) expr.si.vrai else NULL
```

La fonction `switch`

La fonction `switch` permet de choisir entre plusieurs alternatives. C'est une sorte de `if-then-else` généralisé.

```
switch(expr,
       étiquette-1 = bloc-de-code-1,
       étiquette-2 = bloc-de-code-2,
       .
       .
       . )
```

La fonction sélectionne un des blocs de code suivant la valeur d'**expr**.

`switch` par indice

Si **expr** a une valeur numérique (entière) n, alors le n-ième bloc de code est évalué. Dans ce cas, les blocs de code n'ont pas besoin d'être étiquetés :

```
k <- 5
switch(k,"I","II","III","IV","V",
       "VI","VII","VIII","IX","X")

 [1] "V"
```

Si n n'est pas compris dans les indices possible (par exemple, n = 11, dans le cas illustré ci-dessus), alors l'expression `switch` prend la valeur `NULL`.

`switch` par étiquette

Si **expr** à une valeur de type *chaîne de caractères* alors les étiquettes sont examinées afin d'en trouver une correspondant exactement à **expr**. Si une telle étiquette est trouvée, alors le bloc de code correspondant est évalué et sa valeur devient celle de l'expression `switch`.

```
x <- rnorm(10)
loc.est <- "mean"
switch(loc.est,
       median = median(x),
       mean = mean(x))
```

```
 [1] -0.4761289
```

```
mean(x)
```

```
 [1] -0.4761289
```

Variations

Si il n'y a pas d'étiquette correspondante alors :

1. si il y a un bloc de code sans étiquette, il est évalué et sa valeur « est retournée » (c.-à-d. devient la valeur de l'expression `switch`) ;
2. si il n'y a pas de bloc de code sans étiquette, la valeur `NULL` est retournée.

```
switch("c", a = 1, 3, b = 2)
```

```
 [1] 3
```

```
switch("c", a = 1, b = 2)
```

Blocs de code vides

Des blocs de code vides peuvent être utilisés pour faire correspondre plusieurs étiquettes au même bloc de code.

```
switch("a",
       a =,
       b = "Je l'ai",
       c = "Raté" )

 [1] "Je l'ai"
```

Dans la situation ci-dessus, "a" et "b" vont donner "Je l'ai" comme valeur retournée.

ifelse

La fonction `ifelse` permet d'exécuter l'une ou l'autre de deux instructions appliquées à un vecteur suivant que les valeurs d'une condition logique sont vraies ou fausses.

```
x <- c(6:-4)
sqrt(x)
```

```
 [1] 2.45 2.24 2.00 1.73 1.41 1.00 0.00  NaN  NaN
[10]  NaN  NaN
Message d'avis :
In sqrt(x) : production de NaN
```

```
sqrt(ifelse(x >= 0, x, NA))
```

```
 [1] 2.45 2.24 2.00 1.73 1.41 1.00 0.00   NA   NA
[10]   NA   NA
```

Topic

Introduction

Conditionnelles

Boucles

Fonctions de boucles implicites

Traitement des chaînes de caractères

Boucles `for`

Nous souhaitons souvent effectuer une opération donnée sur tous les éléments d'un vecteur, d'une liste ou plus généralement d'un objet « indiçable ». Dans R, ces tâches répétitives sont effectuées avec une « boucle for ».

Une boucle `for` a la forme :

```
for (variable in vecteur) expression
```

L'effet de cette boucle est d'affecter successivement à **variable** la valeur de chaque élément de **vecteur** avant d'évaluer **expression**.

Boucles `for` : exemple 1

Supposons que le vecteur **x** contient des valeurs numériques et que nous souhaitons calculer la somme de ces valeurs. Une manière d'effectuer ce calcul et d'initialiser une variable à zéro avant d'additionner successivement tous les éléments de **x** à cette dernière.

```
s <- 0
for (i in 1:length(x))
  s  <- s + x[i]
```

Ici, **i** se voit succéssivement assigner les valeurs 1, 2, ..., `length(x)` et pour chaque valeur de **i**, l'expression

```
 s  <- s + x[i]
```

est évaluée.

Boucles `for` : exemple 2

L'exemple précédent est typique des boucles utilisées par de nombreux langages de programmation, mais R n'a pas besoin d'utiliser une **variable de boucle** de type entier.

Ainsi, la boucle précédente pourrait être écrite :

```
s <- 0
for (elt in x)
  s  <- s + elt
```

Cette version est à la fois plus simple et plus efficace.

L'instruction `next`

Quand certaines conditions sont remplies, il peut être utile de ne pas effectuer les dernières instructions d'une boucle pour passer directement à l'itération suivante. Ceci peut être fait en évaluant l'instruction `next` lorsque les conditions sont remplies.

```
for (variable in vecteur) {
  expression-1
  expression-2
  if (condition)
    next
  expression-3
  expression-4 }
```

Quand **condition** est vraie (TRUE), **expression-3** et **expression-4** ne sont pas évaluées.

La boucle `while`

Les boucles `for` évaluent une expression un nombre de fois déterminé. Il est aussi utile de pouvoir évaluer une expression jusqu'à ce qu'une condition soit fausse (FALSE). La « boucle while » fournit ce type de structure de contrôle.

```
while (condition) expression
```

Comme d'habitude, **condition** est une expression qui doit avoir pour valeur TRUE ou FALSE et **expression** est une expression simple ou composée.

Exemple de boucle `while`

Comme exemple simple, considérons le problème de sommer les entiers jusqu'à ce que la somme excède un seuil. Pour un seuil de 100, nous pouvons effectuer la tâche de la façon suivante :

```
seuil <- 100
n <- 0
s <- 0
while (s <= seuil) {
  n <- n + 1
  s <- s + n }
c(n,s)

 [1]  14 105
```

La boucle `repeat`

Le dernier type de boucle disponible en R est la « boucle repeat ».

Ce type de boucle a la forme :

```
repeat expression
```

L'effet de cette boucle est de réévaluer **expression** « pour toujours ». Pour sortir de ces boucles il faut utiliser l'instruction `break`.

Sortir d'un boucle avec `break`

L'instruction `break` permet de sortir simplement de **tout type** de boucle. Par exemple :

```
seuil <- 100
n <- 0
s <- 0
repeat {
  if (s > seuil)
    break
  n <- n + 1
  s <- s + n }
c(n,s)
```

```
 [1]  14 105
```

(Cette boucle est complètement équivalente à la boucle `while` que nous avons vu précédemment.)

Topic

Introduction

Conditionnelles

Boucles

Fonctions de boucles implicites

Traitement des chaînes de caractères

Moyennes des lignes et colonnes d'une matrice

Il arrive souvent que l'on souhaite obtenir la moyenne (ou toute autre statistique) des lignes ou des colonnes d'une matrice. Il est bien sur possible des les obtenir avec une boucle `for`. Nous pourrions ainsi obtenir la moyenne des lignes d'une matrice **x** avec :

```
rm <- numeric(nrow(x))
for (i in 1:nrow(x))
  rm[i] <- mean(x[i,])
```

Remarquez qu'il n'y a ici rien de spécial dans la fonction **mean**, le même code peut être utilisé pour tout autre statistique à valeur scalaire comme **var** ou **sd**.

La fonction `apply`

Comme le type de calcul que nous venons d'effectuer sur des structures multidimensionnelles (matrice, etc) est si fréquent, les concepteurs de `R` l'ont doté de fonctions spécialement prévues à cet effet. La fonction `apply` peut ainsi être utilisée pour appliquer une fonction aux lignes ou colonnes d'une matrice. L'expression

```
apply(matrice, 1, fonction)
```

applique **fonction** à chaque ligne de **matrice** et à pour valeur un vecteur qui a autant d'éléments que **matrice** à de lignes. De même,

```
apply(matrice, 2, fonction)
```

applique **fonction** à chaque colonne de **matrice** et à pour valeur un vecteur qui a autant d'éléments que **matrice** à de colonnes.

Moyennes des lignes et colonnes d'une matrice

Les moyennes des lignes et des colonnes d'une matrice **x** peuvent ainsi être calculées avec :

```
(x <- matrix(1:4, nr=2, nc=2))
```

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

```
apply(x,1,mean)
```

```
[1] 2 3
```

```
apply(x,2,mean)
```

```
[1] 1.5 3.5
```

Comme la fonction **mean** n'a rien de spécial on peut lui substituer la fonction **sd** pour obtenir l'écart type des lignes :

```
apply(x,1,sd)
```

```
[1] 1.41 1.41
```

`apply` et fonctions anonymes

Dans l'exemple précédent nous avons utilisé le **nom** de la fonction a appliquer aux lignes et colonnes de la matrice **x**. Mais il est aussi possible de spécifier une fonction directement en utilisant **sa définition** plutôt que son nom. Nous pouvons ainsi calculer la somme des carrés des colonnes de **x** avec :

```
apply(x, 2, function(x) sum(x^2))
```

```
 [1]  5 25
```

Calculer la somme des carrés des variables centrées sur leur moyenne est aussi très simple :

```
apply(x, 2, function(x) sum((x - mean(x))^2))
```

```
 [1] 0.5 0.5
```

`apply` et fonctions à valeurs vectorielles

Il est possible d'utiliser comme troisième paramètre de `apply` une fonction à valeurs vectorielles. La fonction `apply` renvoie alors une matrice dont les colonnes sont les résultats de l'application de la fonction aux lignes (si le second paramètre est 1) ou aux colonnes (si le second paramètre est 2) du premier paramètre.

```
apply(x, 1, range)

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

apply(x, 2, range)

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

Paramètres additionnels des fonctions passées à `apply`

Il est possible de spécifier des paramètres additionnels des fonctions passées à `apply` comme paramètres additionnels de `apply`. Un petit exemple devrait rendre la situation plus claire : la fonction `mean` prend un paramètre optionnel, `trim`, qui indique la fraction des éléments « extrêmes » à supprimer du vecteur dont on souhaite la moyenne avant de calculer cette dernière. La plus grande et la plus petite `trim/2` fraction des observations sont retirées avant le calcul. Pour retirer les 5% valeurs les plus petites et les plus grandes nous utiliserons donc `trim` = 0.1, ce que nous pourrions faire dans `apply` de la façon suivante :

```
apply(matrice, marge, mean, trim = 0.1)
```

`lapply` et `sapply` des variations d'`apply` pour des objets unidimensionnels (1)

Les fonctions `lapply` et `sapply` sont des versions d'`apply` adaptées au objets unidimensionnels comme les **listes** et les **vecteurs**. L'objet étant unidimensionnel, nous avons pas besoin du second paramètre (`MARGIN`) de `apply`. `lapply` renvoie une liste et `sapply` renvoie un vecteur ou une matrice quand c'est possible :

```
lapply(1:3, function(x) x%%2==0)

 [[1]]
 [1] FALSE

 [[2]]
 [1] TRUE

 [[3]]
 [1] FALSE
```

`lapply` et `sapply` des variations d'`apply` pour des objets unidimensionnels (2)

```
sapply(1:10, function(x) x%%2==0)

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

Retirer une statistique d'une matrice avec sweep

Après le calcul d'une statistique, nous somme souvent amenés à soustraire cette dernière des observations d'où elle est issue afin d'obtenir des résidus. Quand cette opération doit être effectuée sur les lignes ou les colonnes d'une matrice, nous pouvons utiliser une boucle `for`. Cette tâche étant fréquemment effectué, `R` met à notre disposition la fonction `sweep`. Ainsi, pour soustraire la statistique définie par la fonction **statistique** des lignes de la matrice **x**, nous employons :

```
sweep(matrice, 1, apply(matrice, 1, statistique))
```

Pour faire la même chose sur les colonnes, nous employons :

```
sweep(matrice, 2, apply(matrice, 2, statistique))
```

Variations sur sweep (1)

Par défaut, sweep soustrait de son premier paramètre, le vecteur définit par son troisième paramètre selon la dimension spécifiée par son deuxième paramètre (MARGIN = 1 pour les lignes et MARGIN = 2 pour les colonnes). Un quatrième paramètre optionnel peut être utilisé pour spécifier une opération binaire autre que la soustraction. Par exemple, pour diviser au lieu de soustraire, il suffit de faire :

```
sweep(matrice,
      1,
      apply(matrice, 1, statistique),
      "/")
```

et

```
sweep(matrice,
      2,
      apply(matrice, 2, statistique),
      "/")
```

Variations sur sweep (2)

Plus concrètement, pour diviser les colonnes d'une matrice par leurs moyennes respectives, nous employons :

```
(x <- cbind(1:3, 9:11))
```

```
     [,1] [,2]
[1,]    1    9
[2,]    2   10
[3,]    3   11
```

```
sweep(x, 2, apply(x, 2, mean), "/")
```

```
     [,1] [,2]
[1,]  0.5  0.9
[2,]  1.0  1.0
[3,]  1.5  1.1
```

Un exemple en statistique

Analysons le taux de mortalité infantile (morts pour 1000 naissances) en fonction de la région des États-Unis et du niveau d'éducation du père (période 1964-1966). C'est un exemple typique de données rencontrées en analyse statistique.

	Éducation du père (années)				
Region	<=8	9-11	12	13-15	>=16
Nord Est	25.3	25.3	18.2	18.3	16.3
Centre Nord	32.1	29.0	18.8	24.3	19.0
Sud	38.8	31.0	19.3	15.7	16.8
Ouest	25.4	21.1	20.3	24.0	17.5

Création de la matrice

```
mortalité  <-
  matrix(c(25.3, 25.3, 18.2, 18.3, 16.3,
           32.1, 29.0, 18.8, 24.3, 19.0,
           38.8, 31.0, 19.3, 15.7, 16.8,
           25.4, 21.1, 20.3, 24.0, 17.5),
         nrow = 4, byrow = TRUE,
         dimnames = list(
           Region =
           c("Nord Est", "Centre Nord",
             "Sud", "Ouest"),
           "Éducation du père (années)" =
           c("<=8", "9-11", "12",
             "13-15", ">=16")))
```

Un modèle linéaire à deux facteurs

Classiquement, on cherche à expliquer ce type de table via :

- un effet global ;
- un effet « région » ;
- un effet « niveau d'éducation du père ».

Le modèle le plus simple auquel on peut penser est un modèle additif, dit « modèle linéaire » :

$$y_{i,j} = \mu + \alpha_i + \beta_j + \epsilon_{i,j} \ ,$$

où $y_{i,j}$ représente l'entrée de la table, μ représente la « mortalité de base », α_i est un effet commun à toutes les observations de la ligne i (effet de la région), β_j est un effet commun à toutes les observations de la colonne j (effet de l'éducation du père) et les $\epsilon_{i,j}$ sont de (petits) résidus décrivant les déviations entre les prédictions du modèle et les observations.

Ajustement par soustraction des effets

Nous allons ajuster notre modèle en **soustrayant successivement** les effets d'une matrice de **résidus courants**. Nous commençons par estimer la « mortalité de base », que nous définissons plus précisément comme la moyenne globale, avant de la soustraire des données brutes. Cela nous donne une matrice de déviations par-rapport à la moyenne globale :

```
r <- mortalité
μ <- mean(mortalité)
r <- r - μ
μ
```

```
 [1] 22.8
```

Soustraction de l'effet de la région

Nous estimons ensuite l'effet de ligne (ou l'effet de région) comme la moyenne suivant les lignes de **r** avant de soustraire, ligne à ligne, cet effet de **r**. Vous aurez devinez que la combinaison de `apply` et de `sweep` est idéale pour cette tâche :

```
α <- apply(r, 1, mean)
r <- sweep(r, 1, α)
α
```

```
   Nord Est Centre Nord          Sud        Ouest
       -2.14        1.82         1.50        -1.16
```

```
sum(α)
```

```
 [1] 3.77e-15
```

Soustraction de l'effet de l'éducation du père

Nous terminons en estimant l'effet de colonne (niveau d'éducation du père) en calculant la moyenne des colonnes de **r** avant de soustraire, colonne par colonne, cet effet :

```
β <- apply(r, 2, mean)
r <- sweep(r, 2, β)
β
```

```
  <=8  9-11    12 13-15  >=16
  7.57  3.78 -3.67 -2.25 -5.42
```

```
sum(β)
```

```
 [1] 0
```

Les résidus

Après avoir terminé notre procédure, il nous reste une matrice de résidus :

```
r
```

	Éducation du père (années)				
Region	<=8	9-11	12	13-15	>=16
Nord Est	-2.955	0.845	1.19	-0.13	1.045
Centre Nord	-0.115	0.585	-2.17	1.91	-0.215
Sud	6.905	2.905	-1.34	-6.37	-2.095
Ouest	-3.835	-4.335	2.32	4.59	1.265

Effets restants ?

Nous pouvons vérifier que nous avons bien soustrait les effets de lignes et de colonnes en calculant les moyennes suivant les lignes et le colonnes de **r**.

```
round(apply(r, 1, mean), digits=4)
```

```
    Nord Est Centre Nord          Sud        Ouest
               0             0            0              0
```

```
round(apply(r, 2, mean), digits=4)
```

```
   <=8  9-11    12 13-15  >=16
     0     0     0     0     0
```

Elles sont (essentiellement) nulles.

Interactions ?

Il arrive que les résidus possèdent une structure qui indique que le modèle n'est pas aussi bien ajusté qu'il le devrait. Il y a ainsi une petite indication d'interactions dans nos résidus :

```
r[order(α), order(β)]
```

	Éducation du père (années)				
Region	>=16	12	13-15	9-11	<=8
Nord Est	1.045	1.19	-0.13	0.845	-2.955
Ouest	1.265	2.32	4.59	-4.335	-3.835
Sud	-2.095	-1.34	-6.37	2.905	6.905
Centre Nord	-0.215	-2.17	1.91	0.585	-0.115

Produits externes (généralisés)

La fonction `outer` fournit une fonctionnalité utile dans le cadre des calculs matriciels. En mathématiques, le produit externe de deux vecteurs **x** et **y** est la matrice dont l'élément (i, j) est :

$$x_i \times y_j \ .$$

La fonction R `outer` généralise le produit externe de sorte que, lorsque qu'une fonction *f* est donnée comme paramètre optionnel, le résultat est une matrice dont l'élément (i, j) est :

$$f(x_i, y_j) \ .$$

Par défaut la fonction utilisée par R est $f(x, y) = x \times y$ ce qui donne le produit externe classique.

Exemple : une table de multiplications

Pour générer une table de multiplication jusqu'à 8x8 il suffit de faire :

```
outer(1:8,1:8)
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,]    1    2    3    4    5    6    7    8
[2,]    2    4    6    8   10   12   14   16
[3,]    3    6    9   12   15   18   21   24
[4,]    4    8   12   16   20   24   28   32
[5,]    5   10   15   20   25   30   35   40
[6,]    6   12   18   24   30   36   42   48
[7,]    7   14   21   28   35   42   49   56
[8,]    8   16   24   32   40   48   56   64
```

Exemple : génération de labels

```
outer(LETTERS[1:8],1:5,
      function(x,y) paste(x, y, sep=""))

     [,1] [,2] [,3] [,4] [,5]
[1,] "A1" "A2" "A3" "A4" "A5"
[2,] "B1" "B2" "B3" "B4" "B5"
[3,] "C1" "C2" "C3" "C4" "C5"
[4,] "D1" "D2" "D3" "D4" "D5"
[5,] "E1" "E2" "E3" "E4" "E5"
[6,] "F1" "F2" "F3" "F4" "F5"
[7,] "G1" "G2" "G3" "G4" "G5"
[8,] "H1" "H2" "H3" "H4" "H5"
```

Exemple : génération de labels (suite)

```
c(outer(LETTERS[1:8],1:5,
        function(x,y) paste(x, y, sep="")))

 [1] "A1" "B1" "C1" "D1" "E1" "F1" "G1" "H1" "A2"
[10] "B2" "C2" "D2" "E2" "F2" "G2" "H2" "A3" "B3"
[19] "C3" "D3" "E3" "F3" "G3" "H3" "A4" "B4" "C4"
[28] "D4" "E4" "F4" "G4" "H4" "A5" "B5" "C5" "D5"
[37] "E5" "F5" "G5" "H5"

c(outer(1:5, LETTERS[1:8],
        function(x,y) paste(y, x, sep="")))

 [1] "A1" "A2" "A3" "A4" "A5" "B1" "B2" "B3" "B4"
[10] "B5" "C1" "C2" "C3" "C4" "C5" "D1" "D2" "D3"
[19] "D4" "D5" "E1" "E2" "E3" "E4" "E5" "F1" "F2"
[28] "F3" "F4" "F5" "G1" "G2" "G3" "G4" "G5" "H1"
[37] "H2" "H3" "H4" "H5"
```

Un exemple plus intéressant

- l'usage de `apply` combiné à celui de `outer` permet de réaliser de façon efficace des calculs assez compliqués ;
- dans cet exemple nous allons chercher pour chaque valeur $y_1, y_2, \ldots, y_m$ l'élément le plus proche dans l'ensemble $\{x_1, \ldots, x_n\}$ (une sorte de régression vers le plus proche voisin) ;
- ce problème peut être résolu avec des boucles imbriquées, mais il admet une solution tenant sur une ligne.

La fonction `which`

La fonction `which` prend comme paramètre un vecteur de booléen et renvoie un vecteur contenant les indices des éléments vrais de l'argument :

```
x <- c(-1, 0, 1)
which( x >= 0)

 [1] 2 3

which( x == 0)

 [1] 2

which( x > 2)

 integer(0)
```

Cette fonction peut être simplement recodée par :

```
which  <-  function(x) (1:length(x))[x]
```

Trouver le plus proche voisin

L'algorithme est :

1. construire la matrice D dont l'élément (i, j) est la distance entre y_i et x_j (on utilisera évidemment `outer` pour cela) ;
2. pour chaque ligne de D, trouver l'indice de la colonne contenant la plus petite distance ;
3. renvoyer la valeur de x correspondant à cet indice.

Mise en œuvre

```
plusProche  <- function(x, y)
  x[apply(outer(y, x,
                function(y, x)
                abs(y - x)),
          1,
          function(x) which(x == min(x))[1])]
x = 0:10/10
(y = round(runif(4), 4))

 [1] 0.4764 0.4498 0.2766 0.0286

plusProche(x, y)

 [1] 0.5 0.4 0.3 0.0
```

Topic

Introduction

Conditionnelles

Boucles

Fonctions de boucles implicites

Traitement des chaînes de caractères

Création de (vecteurs de) chaînes de caractères

Il y a deux manières de créer des chaînes de caractères :

Avec des guillemets simples ou doubles :

```
(chaîne <- c("a",'bb',"ccc"))
```

```
 [1] "a"   "bb"  "ccc"
```

Avec la fonction `as.character` :

```
formule <- y ~ a + b + c
as.character(formule)
```

```
 [1] "~"          "y"          "a + b + c"
```

```
as.character(1:3)
```

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

Chaînes prédéfinies (1)

R dispose de plusieurs vecteurs de chaînes de caractères prédéfinis :

```
letters
```

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

```
LETTERS
```

```
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K"
[12] "L" "M" "N" "O" "P" "Q" "R" "S" "T" "U" "V"
[23] "W" "X" "Y" "Z"
```

Chaînes prédéfinies (2)

```
month.name

 [1] "January"   "February"  "March"
 [4] "April"     "May"       "June"
 [7] "July"      "August"    "September"
[10] "October"   "November"  "December"

month.abb

 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul"
 [8] "Aug" "Sep" "Oct" "Nov" "Dec"
```

La fonction générique `format`

La **fonction générique** `format` « imprime de jolie façon » (*pretty prints*) certains objets `R`. Utilisée avec la fonction `ISOdate`, elle nous permet d'imprimer les mois de l'année ou leurs abréviations suivant le « codage local », c.-à-d. en français :

```
format(ISOdate(2013, 1:12, 1), "%B")
```

```
 [1] "janvier"   "février"   "mars"
 [4] "avril"     "mai"       "juin"
 [7] "juillet"   "août"      "septembre"
[10] "octobre"   "novembre"  "décembre"
```

```
format(ISOdate(2013, 1:12, 1), "%b")
```

```
 [1] "janv." "févr." "mars"  "avril" "mai"
 [6] "juin"  "juil." "août"  "sept." "oct."
[11] "nov."  "déc."
```

(Jetez un œil sur les exemples de la documentation de `format` pour vous faire une idée ses nombreuses possibilités.)

La fonction noquote

Si les guillemets anglo-saxonnes vous irritent, supprimez les avec la fonction noquote :

```
noquote(format(ISOdate(2013, 1:12, 1), "%B"))
```

```
 [1] janvier   février   mars      avril
 [5] mai       juin      juillet   août
 [9] septembre octobre   novembre  décembre
```

La fonction nchar

La fonction `nchar` compte le nombre de symboles d'une chaîne et est vectorisée, c'est-à-dire qu'elle peut être appliquée à un vecteur de chaînes.

```
chaîne1 <- c("un","Cours R","a","Quoi!!!","A","À")
nchar(chaîne1)
```

```
 [1] 2 7 1 8 1 1
```

Remarquez qu'un espace compte bien comme un symbole :

```
nchar(" ")
```

```
 [1] 1
```

La fonction `paste`

Nous l'avons déjà utilisée lorsque nous avons vu la fonction `outer`. Elle permet de concaténer (rabouter) des chaînes comme :

```
paste("R,","c'est super !")
```

```
 [1] "R, c'est super !"
```

Elle fait aussi des conversions automatiques si nécessaire :

```
paste("A",1:3)
```

```
 [1] "A 1" "A 2" "A 3"
```

Avec son paramètre optionnel `collapse`, elle permet de convertir un vecteur de chaînes en une seule chaîne :

```
paste("A",1:3, collapse=", ")
```

```
 [1] "A 1, A 2, A 3"
```

La fonction `substring`

La fonction `substring` permet d'extraire des sous-chaînes d'une chaîne :

```
substring("R, c'est super !", first=4, last=8)
```

```
 [1] "c'est"
```

Elle est aussi vectorisée :

```
paste(substring(c("Windows",
                  "R, c'est super !",
                  "nul"),
                first=c(1,2,1), last=c(7,9,3)),
      collapse="")
```

```
 [1] "Windows, c'est nul"
```

La fonction `strsplit`

La fonction `strsplit` permet de découper des chaînes lorsqu'un « motif » ou, plus précisément, une expression rationnelle (*regular expression*) est trouvée :

```
strsplit("A <- \"A\"", split=" ")

 [[1]]
 [1] "A"     "<-"    "\"A\""
```

Elle est aussi vectorisée et comme vous pouvez le deviner, elle renvoie une liste avec autant d'éléments qu'il y d'éléments dans le vecteur passé comme premier paramètre :

```
strsplit(c("A <- \"A\"", "plot(x, y, type=\"l\")"),
         split=" ")

 [[1]]
 [1] "A"     "<-"    "\"A\""

 [[2]]
 [1] "plot(x,"     "y,"          "type=\"l\")"
```

La fonction `grep`

La fonction `grep` permet de chercher un motif / expression rationnelle dans un vecteur et renvoie les indices des élément du vecteur qui contiennent ce motif. Un exemple simple :

```
grep("r|R",c("R","C","Fortran","Common Lisp"))
```

```
 [1] 1 3
```

Un exemple plus compliqué où j'appelle la fonction `list.files` et où je cherche les fichier dont l'extension est « .tex » dans le répertoire courant :

```
list.files()[grep(".tex$",list.files())]
```

```
 [1] "Pouzat_MNHN_130417-V1.tex"
 [2] "Pouzat_MNHN_130417-V2.tex"
 [3] "Pouzat_MNHN_130417-V3.tex"
```

Fonctions `tolower` et `toupper`

Les fonctions `tolower` et `toupper` permettent de changer la casse (typographique) d'une chaîne de caractère :

```
toupper("ceci a commencé en minuscules...")
```

```
 [1] "CECI A COMMENCÉ EN MINUSCULES..."
```

```
tolower("Et Est Passé en Majuscules...")
```

```
 [1] "et est passé en majuscules..."
```