Fiche TD avec le logiciel  : tdr13

Objets

J.R. Lobry, A.B. Dufour & D. Chessel

Les bases du langage. Les vecteurs. Les tableaux (data frames). Les fonctions. Calcul sur les vecteurs. Statistiques élémentaires. Les facteurs.

Table des matières

$=$	égalité	`==`
$\wedge$	ET	`&`
$\vee$	OU	`|`
$\neg$	NON	`!`

TAB. 1 – Les opérateurs logiques

1 Logique ternaire : indécidabilité

Les trois valeurs logiques possibles dans R sont :

1. le vrai (`TRUE` que l'on peut abréger par `T`)
2. le faux (`FALSE` que l'on peut abréger par `F`)
3. l'indécidable (`NA` à ne pas confondre avec `NaN`)

Les opérateurs logique de base sont donnés dans la table 1. L'utilisation d'une logique ternaire est *indispensable* pour une bonne gestion des informations manquantes, ce qui est très courant en statistique, mais peut conduire à des résultats *a priori* surprenants si vous êtes habitués à une logique binaire classique, par exemple :

```
NA == NA
```
```
[1] NA
```

Nous allons construire les tables de vérité des opérateurs logiques et nous concentrer sur le cas de l'indécidabilité. Construisons un vecteur comportant les trois valeurs logiques possibles et donnons un nom à ces éléments :

```
x <- c(NA, FALSE, TRUE)
names(x) <- as.character(x)
x
```
```
<NA> FALSE  TRUE
  NA FALSE  TRUE
```

1.1 Négation logique : !

```
nonx <- !(x)
nonx
```
```
<NA> FALSE  TRUE
  NA  TRUE FALSE
```

La négation du vrai est le faux, la négation du faux est le vrai, et la négation de l'indécidable est l'indécidable.

1.2 Le ET logique : &

```
outer(x, x, "&")
```
```
      <NA> FALSE  TRUE
<NA>    NA FALSE    NA
FALSE FALSE FALSE FALSE
TRUE    NA FALSE  TRUE
```

Notez que l'indécidabilité ne se propage pas systématiquement puisque `NA & FALSE` est `FALSE`.

1.3 Le OU logique : |

```
 outer(x, x, "|")
      <NA> FALSE TRUE
<NA>    NA    NA TRUE
FALSE   NA FALSE TRUE
TRUE  TRUE  TRUE TRUE
```

Notez que l'indécidabilité ne se propage pas systématiquement puisque `NA | TRUE` est `TRUE`.

1.4 L'égalité logique : ==

```
 outer(x, x, "==")
      <NA> FALSE  TRUE
<NA>    NA    NA    NA
FALSE   NA  TRUE FALSE
TRUE    NA FALSE  TRUE
```

Notez que l'indécidabilité se propage *systématiquement*. C'est voulu pour éviter que l'indécidabilité ne conduise à des certitudes. Mais alors comment fait-on pour tester la valeur de vérité d'une proposition logique ?

1.5 L'égalité exacte : `identical()`

Lire avec *attention* la documentation de la fonction `identical()` :

> `Users often use the comparison operators, such as '==' or '!=', in these situations. It looks natural, but it is not what these operators are designed to do in R. They return an object like the arguments. If you expected 'x' and 'y' to be of length 1, but it happened that one of them wasn't, you will _not_ get a single 'FALSE'. Similarly, if one of the arguments is 'NA', the result is also 'NA'. In either case, the expression 'if(x == y)....' won't work as expected.`

```
 identical(NA, NA)
[1] TRUE
 identical(TRUE, TRUE)
[1] TRUE
 identical(FALSE, FALSE)
[1] TRUE
```

La fonction `identical()` est le moyen sûr et efficace de tester l'identité exacte entre deux objets. Ne pas confondre avec l'opérateur logique `==` qui est un opérateur logique au comportement ... très logique.

1.6 Exercice Pierre Paul Jacques

La taille de Pierre est 1.90 m, celle de Paul est 1.5 m, on ne connaît pas la taille de Jacques :

```
Pierre <- 1.9
Paul <- 1.5
Jacques <- NA
```

Donner la valeur de vérité des propositions suivantes :

1. Pierre est très grand, il fait plus de 2 m.
```
Pierre > 2
```
```
[1] FALSE
```
2. Pierre est plus grand que Paul.
```
[1] TRUE
```
3. Pierre est plus grand que Jacques.
```
[1] NA
```
4. La proposition "Pierre est plus grand que Jacques" est indécidable.
```
[1] TRUE
```
5. Les tailles de Pierre, Paul et Jacques sont toutes supérieure à 1 m.
```
[1] NA
```
6. On sait que l'on ne sait pas si les tailles de Pierre, Paul et Jacques sont toutes supérieure à 1 m.
```
[1] TRUE
```
7. De Pierre, Paul et Jacques, il y en a au moins un qui fait plus de 1 m.
```
[1] TRUE
```
8. La proposition "de Pierre, Paul et Jacques, il y en a au moins un qui fait plus de 1 m" est indécidable.
```
[1] FALSE
```
9. Jacques a la même taille que Jacques.
```
[1] NA
```
10. La proposition "Jacques a la même taille que Jacques" est vraie.
```
[1] FALSE
```
11. La proposition "Jacques a la même taille que Jacques" est indécidable.
```
[1] TRUE
```
12. La proposition "La proposition "Jacques a la même taille que Jacques" est indécidable" est vraie.
```
[1] TRUE
```

Vous pouvez, à juste titre, être choqué par le fait que la proposition "Jacques a la même taille que Jacques" est indécidable sous R. Même si je ne connais pas la taille de Jacques, ne puis-je pas affirmer que Jacques a exactement la même taille que Jacques ? Vous pouvez le faire pour l'objet mathématique abstrait (un point de $\mathbb{R}$) qui représente la taille de Jacques, indépendamment du réel. Vous ne pouvez pas le faire pour l'objet R concret qui donne la taille de Jacques dans le réel. C'est une différence essentielle entre les mathématiques et la statistique : si un objet statistique n'est pas documenté, on ne peut *rien* tirer de cet objet tout simplement *parce qu'il n'existe pas*. On ne peut pas dire que Jacques a la même taille que Jacques parce que l'objet statistique "taille de Jacques" n'existe pas tant que l'on ne l'a pas mesuré.

```
NA == NA
```
```
[1] NA
```

2 Imprécision numérique

Soient x_1 et x_2 deux éléments de $\mathbb{R}$ tels que :

$$x_1 = \frac{5}{10} - \frac{3}{10}$$

$$x_2 = \frac{3}{10} - \frac{1}{10}$$

On peut dire que x_1 et x_2 sont égaux parce que

$$\begin{aligned} & x_1 = x_2 \\ \Longleftrightarrow \quad & \tfrac{5}{10} - \tfrac{3}{10} = \tfrac{3}{10} - \tfrac{1}{10} \\ \Longleftrightarrow \quad & \tfrac{5-3}{10} = \tfrac{3-1}{10} \\ \Longleftrightarrow \quad & \tfrac{2}{10} = \tfrac{2}{10} \end{aligned}$$

est une tautologie. Mais qu'en est-il en pratique ?

```
 (x1 <- 5/10 - 3/10)
[1] 0.2
 (x2 <- 3/10 - 1/10)
[1] 0.2
 x1 == x2
[1] FALSE
```

Il y a toujours des erreurs d'arrondi :

```
 format(x1, digits = 22)
[1] "0.2"
 format(x2, digits = 22)
[1] "0.2"
 x1 - x2
[1] 2.775558e-17
```

Pour tester l'égalité de deux valeurs en tenant compte des imprécisions numériques il faut utiliser la fonction `all.equal()`.

```
 all.equal(x1, x2)
[1] TRUE
```

Pour tester la valeur de retour de cette fonction on utilise `identical()` :

```
 identical(all.equal(x1, x2), TRUE)
[1] TRUE
```

On peut définir son propre opérateur binaire infixe pour alléger les notations :

```
 "%==%" <- function(x, y) identical(all.equal(x, y), TRUE)
 x1 %==% x2
[1] TRUE
 1 %==% (1 + .Machine$double.eps^0.5)
[1] TRUE
 1 %==% (1 + 2 * (.Machine$double.eps^0.5))
[1] FALSE
```

3 Valeurs manquantes

Les valeurs manquantes sont représentées par `NA`. L'indétermination se propage *systématiquement* dans les calculs pour éviter de fausser le résultat final :

```
 NA + 1
[1] NA
 2 * NA
[1] NA
```

La plupart des fonctions courantes possèdent un paramètre `na.rm` qui permet d'enlever les valeurs manquantes avant de faire le calcul demandé, mais il faut le faire explicitement, sa valeur par défaut est toujours `FALSE` pour éviter les catastrophes. Prenons la fonction `mean` qui permet de calculer la moyenne.

```
 mean(c(pi, NA))
[1] NA
 mean(c(pi, NA), na.rm = TRUE)
[1] 3.141593
```

Il y a de nombreuses fonctions utilitaires sous R pour aider à la bonne gestion des valeurs manquantes comme par exemple `is.na()` et `na.omit()`.

4 Les vecteurs

4.1 Intérêt du calcul sur les vecteurs

Dans R même une simple valeur numérique est un vecteur :

```
 pi
[1] 3.141593
 is.vector(pi)
[1] TRUE
```

Prenons la définition de la variance d'un échantillon de taille n d'une variable continue X :

$$\text{var}\,(\text{X}) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

Dans un langage de programmation classique nous écririons quelque chose du type :

```
variance <- function(X) {
    n <- length(X)
    sum1 <- 0
    for (i in seq(from = 1, to = n)) {
        sum1 <- sum1 + X[i]
    }
    moyenne <- sum1/n
    sum2 <- 0
    for (i in seq(from = 1, to = n)) {
        sum2 <- sum2 + (X[i] - moyenne) * (X[i] - moyenne)
    }
    return(sum2/n)
}
```

Prenons un échantillon constitué des 10 premiers entiers. Il s'écrit :

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

Et la variance de cet échantillon est obtenue par :

```
variance(1:10)
```
```
[1] 8.25
```

Mais dans R, nous pouvons manipuler directement les vecteurs de la façon suivante :

```
variance <- function(X) {
    n <- length(X)
    moyenne <- sum(X)/n
    return(sum((X - moyenne)^2)/n)
}
variance(1:10)
```
```
[1] 8.25
```

Les expressions sont ainsi beaucoup plus compactes et proches des notations mathématiques.

Exercice

1. Donner le carré des 10 premiers entiers :
   ```
   [1]   1   4   9  16  25  36  49  64  81 100
   ```
2. Donner le carré des 10 premiers entiers moins un :
   ```
   [1]  0  3  8 15 24 35 48 63 80 99
   ```
3. Donner le cube des 10 premiers entiers :
   ```
   [1]    1    8   27   64  125  216  343  512  729 1000
   ```
4. Donner le sinus (`sin()`) des 10 premiers entiers :
   ```
   [1]  0.8414710  0.9092974  0.1411200 -0.7568025 -0.9589243 -0.2794155  0.6569866
   [8]  0.9893582  0.4121185 -0.5440211
   ```
5. Donner 10 à la puissance des 10 premiers entiers :
   ```
   [1] 1e+01 1e+02 1e+03 1e+04 1e+05 1e+06 1e+07 1e+08 1e+09 1e+10
   ```
6. Donner le logarithme népérien (`log()`) de 10 à la puissance des 10 premiers entiers :
   ```
   [1]  2.302585  4.605170  6.907755  9.210340 11.512925 13.815511 16.118096 18.420681
   [9] 20.723266 23.025851
   ```
7. Donner le logarithme décimal (`log10()`) de 10 à la puissance des 10 premiers entiers :
   ```
   [1]  1  2  3  4  5  6  7  8  9 10
   ```
8. Donner la racine carré (`sqrt()`) des 10 premiers entiers :
   ```
   [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427 3.000000
  [10] 3.162278
   ```
9. Donner la somme (`sum()`) des 10 premiers entiers :
   ```
   [1] 55
   ```
10. Donner la somme cumulée (`cumsum()`) des 10 premiers entiers :
    ```
    [1]  1  3  6 10 15 21 28 36 45 55
    ```
11. Donner le produit (`prod()`) des 10 premiers entiers :
    ```
    [1] 3628800
    ```
12. Donner le produit cumulé (`cumprod()`) des 10 premiers entiers :
    ```
    [1]       1       2       6      24     120     720    5040   40320  362880 3628800
    ```
13. Donner les différences (`diff()`) des 10 premiers entiers avec leur précédent :
    ```
    [1] 1 1 1 1 1 1 1 1 1
    ```

4.2 Statistiques élémentaires

Lire la documentation des fonctions pour comprendre ce qu'elles calculent. Pourquoi le résultat de `var()` n'est pas le même que celui de la fonction `variance()` que nous avons définie plus haut ?

Considérons les notes de 14 étudiants d'un groupe de TP et calculons les paramètres statistiques élémentaires. L'écriture d'un vecteur est une combinaison (cf le paragraphe sur les séries de valeurs numériques ci-après).

```
notes <- c(15, 8, 14, 12, 14, 10, 18, 15, 9, 5, 12, 13, 12, 16)
sort(notes)
 [1]  5  8  9 10 12 12 12 13 14 14 15 15 16 18
length(notes)
[1] 14
min(notes)
[1] 5
max(notes)
[1] 18
range(notes)
[1]  5 18
median(notes)
[1] 12.5
quantile(notes)
   0%   25%   50%   75%  100%
 5.00 10.50 12.50 14.75 18.00
mean(notes)
[1] 12.35714
var(notes)
[1] 11.93956
sd(notes)
[1] 3.455367
unique(notes)
 [1] 15  8 14 12 10 18  9  5 13 16
sort(unique(notes))
 [1]  5  8  9 10 12 13 14 15 16 18
```

4.3 Affectation (`<-`, `->`, `=`)

Mettre la valeur 7 dans l'objet de nom `x` :

```
x <- 7
```

Pour voir ce que contient un objet, il suffit d'entrer son nom :

```
x
[1] 7
```

On peut effectuer d'un coup l'affectation et l'affichage en entourant avec des parenthèses :

```
(x <- 7)
[1] 7
```

Exercice. Mettre la valeur 12 dans l'objet `x`, vous devez obtenir le résultat suivant :

```
x
[1] 12
```

L'affectation peut se faire avec la flèche dans l'autre sens (`7 -> x`). Cette construction est très pratique quand on a évalué une expression assez complexe que l'on rappelle avec la touche ↑. Il suffit alors de la compléter par `-> x` pour la sauvegarder dans l'objet `x`.

Exercice.

1. Calculez l'expression suivante :
   ```
   ((1 + sqrt(5))/2)^2
   [1] 2.618034
   ```
2. Rappelez la commande avec la touche ↑ et sauvegardez le résultat dans `x` :
   ```
   ((1 + sqrt(5))/2)^2 -> x
   ```
3. Que vaut `x` ?
   ```
   x
   [1] 2.618034
   ```

Le signe égal (`=`) est parfois utilisé pour faire des affectations :

```
x = 7
```

Mais ceci ne fonctionnera pas dans tous les contextes, par exemple si l'affectation est elle-même encapsulée dans une fonction. Si on utilise la fonction `system.time()` pour évaluer le temps que prend une affectation, cela fonctionne parfaitement avec l'opérateur `<-` :

```
system.time(x <- 7)
   user  system elapsed
      0       0       0
```

mais donnera une erreur avec l'opérateur `=` :

```
system.time(x = 7)
Error in system.time(x = 7) : unused argument(s) (x ...)
```

En effet, l'opérateur `=` est utilisé également pour donner une valeur aux paramètres d'une fonction lors de leur appel. Mais ici, il n'y a pas de paramètre `x` pour la fonction `system.time()`, d'où l'erreur produite. Pour éviter la confusion il est préférable de n'utiliser l'opérateur `=` que pour donner une valeur aux paramètres lors de l'appel d'une fonction.

4.4 Séries de valeurs numériques

4.4.1 Séries d'entiers : :

L'opérateur deux points `:` permet de générer des séries d'entiers :

```
x <- 1:12
```

Exercice. Générer la série suivante :

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

4.4.2 Séries de valeurs numériques : `seq()`

La fonction `seq()` permet de générer une série de nombres équidistants :

```
 seq(from = 1, to = 5)
[1] 1 2 3 4 5
 seq(from = 1, to = 2, by = 0.1)
 [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0
 seq(from = 10, to = 50, length = 10)
 [1] 10.00000 14.44444 18.88889 23.33333 27.77778 32.22222 36.66667 41.11111 45.55556
[10] 50.00000
 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 24 25 26
```

Exercice. Générer les séries suivantes :

```
[1]  0 -1 -2 -3 -4 -5
 [1] 0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09 0.10
[1] 1.000000 1.666667 2.333333 3.000000
```

4.5 Répétition de valeurs : `rep()`

```
 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"
```

Exercice. Générer les séries suivantes :

```
[1] 1 2 3 1 2 3 1 2 3
 [1] 1 2 3 4 1 2 3 4 1 2 3 4
 [1] 1 1 1 2 2 2 3 3 3 4 4 4
[1] "un"   "un"   "un"   "deux" "deux" "deux" "deux" "deux" "deux"
```

4.6 Combinaison : `c()`

```
 x <- c(1.5, 7.2, pi)
 x <- c(1:3, 10:12)
```

Exercice. Construire les vecteurs suivants :

```
[1] 11.1  2.7  3.3
[1] -5.00 -4.00 -3.00 -2.00 -1.00  0.00  1.00  1.10 88.77
```

Remarque :

```
 x <- c(2, 5, -3, 8, "a")
```

`x` est un vecteur de chaînes de caractères.

4.7 Mode : `mode()`

Reprenons le vecteur `x` du paragraphe précédent.

```
mode(x)
```
```
[1] "character"
```

`x` est un vecteur de chaînes de caractères.

```
x <- c(1, 5, -36, 3.66)
mode(x)
```
```
[1] "numeric"
```

`x` est un vecteur numérique.

```
x <- c(T, T, T, F, F)
mode(x)
```
```
[1] "logical"
```

`x` est un vecteur logique.

4.8 Saisie au clavier : `scan()`

```
x <- scan()
```

```
1: 145
2: -1
3: 28.88
4: 2e-02
5: 3.45e1
6:
```

Le premier Entrée après une chaîne vide met fin à la saisie.

```
x
```
```
[1] 145.00  -1.00  28.88   0.02  34.50
```

4.9 Éléments des vecteurs

4.9.1 Indexation par des entiers positifs

Reprenons le vecteur `x` du paragraphe précédent.
Le quatrième élément :

```
x[4]
```
```
[1] 0.02
```

Du second au troisième élément :

```
x[2:3]
```
```
[1] -1.00 28.88
```

On peut reprendre plusieurs fois le même :

```
x[c(2, 2, 3)]
```
```
[1] -1.00 -1.00 28.88
```

Les éléments hors bornes ne sont pas disponibles (`NA`, not available, donnée manquante) :

```
x[100]
```
```
[1] NA
```

4.9.2 Indexation par des entiers négatifs

Tous sauf le quatrième :

```
 x[-4]
[1] 145.00  -1.00  28.88  34.50
```

Exercice 1. Donner tous les éléments sauf le premier.

```
[1] -1.00 28.88  0.02 34.50
```

Exercice 2. A l'aide de la fonction `length()`, donner tous les éléments sauf le dernier :

```
[1] 145.00  -1.00  28.88   0.02
```

On ne peut pas mélanger les indexations par des entiers positifs et négatifs simultanément. Il faut procéder en deux temps.

Exercice 3. Donner le deuxième et le troisième élément une fois que l'on a enlevé le premier élément :

```
[1] -1.00 28.88
```

4.9.3 Indexation par un vecteur logique

```
 x[c(T, F, F, T, T)]
[1] 145.00   0.02  34.50
 x[c(T, F)]
[1] 145.00  28.88  34.50
```

Si le vecteur logique n'est pas assez long il sera recyclé autant de fois que nécessaire.

Exercice. Donner un élément sur deux à partir du deuxième :

```
[1] -1.00  0.02
```

Le vecteur logique d'indexation peut être issu d'un calcul logique, c'est l'utilisation la plus courante :

```
 x > 10
[1]  TRUE FALSE  TRUE FALSE  TRUE
 x[x > 10]
[1] 145.00  28.88  34.50
 x > 0 & x < 1
[1] FALSE FALSE FALSE  TRUE FALSE
 x[x > 0 & x < 1]
[1] 0.02
```

Exercice. Donner la liste des éléments compris entre 10 et 50 :

```
[1] 28.88 34.50
```

4.9.4 Indexation par des noms

Ceci ne fonctionne que si les éléments du vecteur ont un nom. Donnons comme nom les 5 premières lettres de l'alphabet aux éléments du vecteur `x` :

```
(names(x) <- letters[1:5])
[1] "a" "b" "c" "d" "e"
```

Quelles sont les valeurs des éléments `a` et `e` ?

```
x[c("a", "e")]
    a     e
145.0  34.5
```

Pour exclure des éléments par leur nom, il faut d'abord récupérer leurs indices avec la fonction `match()`. Quelles sont les valeurs des éléments autre que `a` et `e` ?

```
x[-match(c("a", "e"), names(x))]
    b     c     d
-1.00 28.88  0.02
```

Exercice. Quelles sont les consonnes ?

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

4.10 Permutations de l'ordre des valeurs : `rev()`, `order()`

La fonction `rev()` donne les éléments dans l'ordre inverse :

```
rev(1:10)
 [1] 10  9  8  7  6  5  4  3  2  1
```

Exercice. Donner les lettres dans l'ordre alphabétique inverse :

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

La fonction `sort()` donne le vecteur des composantes rangées par ordre croissant :

```
sort(10:1)
 [1]  1  2  3  4  5  6  7  8  9 10
```

En pratique, on utilise plutôt la fonction `order()` qui donne la permutation qui donne le rangement par ordre croissant :

```
(x <- 10:1)
 [1] 10  9  8  7  6  5  4  3  2  1
order(x)
 [1] 10  9  8  7  6  5  4  3  2  1
x[order(x)]
 [1]  1  2  3  4  5  6  7  8  9 10
all.equal(x[order(x)], sort(x))
[1] TRUE
```

5 Les facteurs : `factor()`

5.1 Les modalités : `levels()`

Les facteurs sont la représentation sous R des *variables qualitatives* (la couleur des yeux, le genre (♂, ♀), un niveau de douleur, etc). De telles données sont souvent codées numériquement, mais doivent être converties avant d'être utilisées.

```
douleur <- c(0, 3, 2, 2, 1)
fdouleur <- factor(douleur, levels = 0:3)
is.numeric(fdouleur)
```

```
[1] FALSE
```

```
is.character(fdouleur)
```

```
[1] FALSE
```

```
is.factor(fdouleur)
```

```
[1] TRUE
```

```
summary(fdouleur)
```

```
0 1 2 3
1 1 2 1
```

```
table(fdouleur)
```

```
fdouleur
0 1 2 3
1 1 2 1
```

La fonction `levels()` permet de récupérer ou de modifier les modalités des variables qualitatives, la fonction `as.numeric()` donne le codage numérique des modalités :

```
levels(fdouleur)
```

```
[1] "0" "1" "2" "3"
```

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

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

```
levels(fdouleur)
```

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

```
as.numeric(fdouleur)
```

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

5.2 Passage quantitatif → qualitatif : `cut()`

La fonction `cut()` permet facilement de transformer une variable quantitative en qualitative :

```
Z <- rnorm(25)
cut(Z, breaks = -3:3)
```

```
 [1] (1,2]   (-1,0]  (0,1]   (1,2]   (0,1]   (0,1]   (-1,0]  (0,1]   (1,2]   (-1,0]
[11] (0,1]   (0,1]   (-2,-1] (-1,0]  (0,1]   (1,2]   (-1,0]  (-2,-1] (-1,0]  (0,1]
[21] (1,2]   (0,1]   (0,1]   (1,2]   (-1,0]
Levels: (-3,-2] (-2,-1] (-1,0] (0,1] (1,2] (2,3]
```

Remarque : `rnorm(25)` est le résultat du tirage de 25 valeurs prises au hasard dans une loi normale centrée réduite.

6 Les listes

6.1 Création d'une liste : `list()`

Les listes sont une structure de données très flexible et très utilisée dans R. Un élément d'une liste est un objet R *quelconque*, y compris une autre liste. La fonction `list()` permet de créer des listes :

```
maliste <- list(a = pi, b = "une chaine", c = c(T, F, NA))
maliste
```

```
$a
[1] 3.141593
$b
[1] "une chaine"

$c
[1]  TRUE FALSE    NA
```

6.2 Extraction d'un élément d'une liste : `$` et `[[`

Un élément d'une liste est en général extrait par son nom (avec l'opérateur `$`), mais on peut aussi utiliser sa position dans la liste avec l'opérateur `[[` :

```
maliste$a
```

```
[1] 3.141593
```

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

```
[1] 3.141593
```

Exercice.

1. Donner l'élément `b` de la liste :

   ```
   [1] "une chaine"
   ```

2. Donner le troisième élément de la liste :

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

6.3 Ajout d'un élément dans une liste : `$`

```
maliste$d <- 1:10
maliste
```

```
$a
[1] 3.141593
$b
[1] "une chaine"

$c
[1]  TRUE FALSE    NA

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

6.4 Sélection d'éléments d'une liste : `[`

L'opérateur `[` sur les listes ne fait pas d'extraction mais renvoie une liste avec les éléments sélectionnés. Par exemple, pour supprimer le premier élément de la liste :

```
maliste <- maliste[-1]
maliste
```

```
$b
[1] "une chaine"
$c
[1]  TRUE FALSE    NA

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

7 Les tableaux : `data frames`

7.1 Les objets de type `data.frame`

Les data frames sont des listes dont tous les éléments ont la même longueur. On peut mélanger dans un `data.frame` des variables quantitatives, qualitatives, logiques et textuelles[1]. C'est typiquement le type d'objet que l'on récupère en lisant des données dans un fichier, par exemple :

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

```
[1] TRUE
```

```
names(t3var)
```

```
[1] "sexe" "poi"  "tai"
```

```
summary(t3var)
```

```
sexe        poi             tai
f:25   Min.   :47.00   Min.   :150.0
h:41   1st Qu.:53.00   1st Qu.:168.0
       Median :65.50   Median :174.5
       Mean   :64.52   Mean   :174.1
       3rd Qu.:73.00   3rd Qu.:180.0
       Max.   :86.00   Max.   :200.0
```

7.2 Éléments des data frames

: `$` et `tab[i,j]`

Les data frames étant des listes, les variables (en colonne) sont directement accessibles par leur nom :

```
t3var$tai
```

```
 [1] 170 169 172 174 168 161 162 189 160 175 165 164 175 184 178 158 164 179 182 174
[21] 158 163 172 185 170 178 180 189 172 174 200 178 178 168 170 160 163 168 172 175
[41] 180 162 177 169 173 182 183 184 181 180 178 178 168 161 171 180 174 175 182 181
[61] 188 182 189 178 150 186
```

Exercice.

1. Donner la moyenne des tailles :

```
[1] 174.0606
```

[1] c'est la différence avec les matrices qui elles ne peuvent contenir qu'un seul type de données

2. Donner la médiane des poids :

```
[1] 65.5
```

3. Donner la variance des poids :

```
[1] 123.5767
```

4. Donner l'écart-type des poids :

```
[1] 11.11651
```

On peut également utiliser la notation de type `tab[i,j]` où `i` représente l'index des lignes et `j` celui des colonnes. Par exemple, pour avoir les 5 premiers individus :

```
t3var[1:5, ]
  sexe poi tai
1    h  60 170
2    f  57 169
3    f  51 172
4    f  55 174
5    f  50 168
```

Pour avoir les première et troisième colonnes des 5 premiers individus :

```
t3var[1:5, c(1, 3)]
  sexe tai
1    h 170
2    f 169
3    f 172
4    f 174
5    f 168
```

Tous les types d'indexation vu pour les vecteurs (entiers positifs, négatifs, logiques, noms) sont utilisables.

Exercice.

1. Donner les individus 1, 5 et 55 :

```
   sexe poi tai
1     h  60 170
5     f  50 168
55    h  73 171
```

2. Pour les 10 premiers individus, donner toutes les variables sauf la première :

```
   poi tai
1   60 170
2   57 169
3   51 172
4   55 174
5   50 168
6   50 161
7   48 162
8   72 189
9   52 160
10  64 175
```

3. Donner un individu sur quatre à partir du premier :

```
   sexe poi tai
1     h  60 170
5     f  50 168
9     f  52 160
13    h  61 175
17    f  53 164
21    f  49 158
25    f  53 170
29    f  70 172
33    h  76 178
37    f  53 163
```

```
41    h  75 180
45    h  55 173
49    h  71 181
53    h  62 168
57    h  60 174
61    h  82 188
65    f  47 150
```

4. Donner tous les individus de sexe féminin :

```
   sexe poi tai
2     f  57 169
3     f  51 172
4     f  55 174
5     f  50 168
6     f  50 161
7     f  48 162
9     f  52 160
11    f  53 165
16    f  51 158
17    f  53 164
21    f  49 158
22    f  50 163
25    f  53 170
29    f  70 172
30    f  62 174
34    f  51 168
35    f  52 170
36    f  57 160
37    f  53 163
38    f  55 168
39    f  66 172
42    f  50 162
43    f  53 177
54    f  47 161
65    f  47 150
```

5. Donner tous les individus qui font plus de 175 cm :

```
   sexe poi tai
8     h  72 189
14    h  78 184
15    h  68 178
18    h  79 179
19    h  74 182
24    h  80 185
26    h  73 178
27    h  70 180
28    h  72 189
31    h  77 200
32    h  70 178
33    h  76 178
41    h  75 180
43    f  53 177
46    h  72 182
47    h  75 183
48    h  73 184
49    h  71 181
50    h  66 180
51    h  71 178
52    h  79 178
56    h  72 180
59    h  85 182
60    h  73 181
61    h  82 188
62    h  86 182
63    h  85 189
64    h  65 178
66    h  74 186
```

6. Donner tous les individus de sexe féminin qui font plus de 175 cm :

```
   sexe poi tai
43    f  53 177
```

7. Donner le poids et la taille tous les individus de sexe féminin qui font plus de 175 cm :

```
   poi tai
43  53 177
```

7.3 Sélection des lignes : `subset()`

Il est très courant de sélectionner les individus (en ligne) en fonction de critères calculés sur les variables (en colonne). Pour éviter d'expliciter à chaque fois le nom du `data.frame` pour désigner une variable, on peut avantageusement utiliser la fonction `subset()`. Ainsi, la sélection des individus de sexe féminin qui font plus de 175 cm s'écrit aussi :

```
subset(t3var, sexe == "f" & tai > 175)
```

```
   sexe poi tai
43    f  53 177
```

7.4 Agrégation des lignes : `aggregate()` et `split()`

Il est courant que les modalités d'une variable qualitative représente des groupes d'individus, par exemple ici les hommes et les femmes. La fonction `aggregate()` permet de calculer des statistiques simples par groupe. Par exemple, le poids et la taille moyenne des hommes et des femmes est :

```
aggregate(t3var[, c("poi", "tai")], list(sexe = t3var$sexe), mean)
```

```
  sexe      poi      tai
1    f 53.40000 165.6400
2    h 71.29268 179.1951
```

Exercice.

1. Calculer la somme des poids des hommes et des femmes :

   ```
     sexe    x
   1    f 1335
   2    h 2923
   ```

2. Calculer la variance des poids des hommes et des femmes :

   ```
     sexe       x
   1    f 30.7500
   2    h 58.0622
   ```

3. Calculer la médiane des tailles des hommes et des femmes :

   ```
     sexe   x
   1    f 165
   2    h 179
   ```

Une autre façon de procéder consiste à décomposer un vecteur en liste avec la fonction `split()` :

```
poitai <- split(t3var[, 2:3], t3var$sexe)
```

On peut alors utiliser la fonction `lapply()` pour appliquer une fonction à tous les éléments de la liste :

```
lapply(poitai, mean)
```

```
$f
   poi    tai
 53.40 165.64
$h
      poi       tai
 71.29268 179.19512
```

On peut passer des arguments à la fonction que l'on applique ainsi :

```
lapply(poitai, mean, na.rm = TRUE)
```

```
$f
   poi    tai
 53.40 165.64
$h
      poi       tai
 71.29268 179.19512
```

Exercice.

1. Calculer la variance des poids et tailles des hommes et des femmes :

```
lapply(poitai, var)
```

```
$f
          poi      tai
poi 30.75000 19.73333
tai 19.73333 40.32333
$h
          poi      tai
poi 58.06220 30.64146
tai 30.64146 45.41098
```

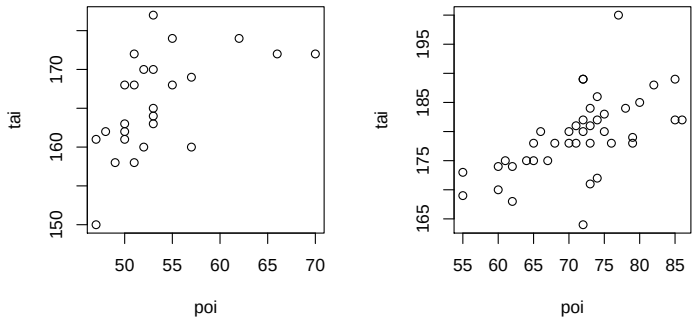
2. Calculer l'écart-type des poids et tailles des hommes et des femmes :

```
lapply(poitai, sd)
```

```
$f
     poi      tai
5.545268 6.350066
$h
     poi      tai
7.619855 6.738767
```

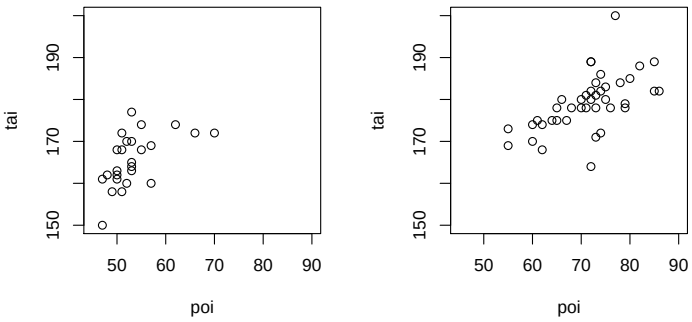
3. Appliquer la fonction `plot()` à la liste `poitai` :

```
par(mfrow = c(1, 2))
lapply(poitai, plot)
```



Modifier les valeurs des paramètres `xlim` et `ylim` de la fonction `plot` pour avoir des échelles cohérentes :

```
par(mfrow = c(1, 2))
lapply(poitai, plot, ylim = c(150, 200), xlim = c(45, 90))
```



ou encore, pour introduire à la fiche suivante :

```
par(mfrow = c(1, 1))
plot(t3var[, c("tai", "poi")], main = paste("Poids et taille de",
    nrow(t3var), "individus"), xlab = "Taille", ylab = "Poids",
    col = c("red", "blue")[as.numeric(t3var$sexe)], pch = c(17,
        20)[as.numeric(t3var$sexe)], cex = 2, las = 1)
legend(185, 60, c("Femmes", "Hommes"), pch = c(17, 20), col = c("red",
    "blue"))
```