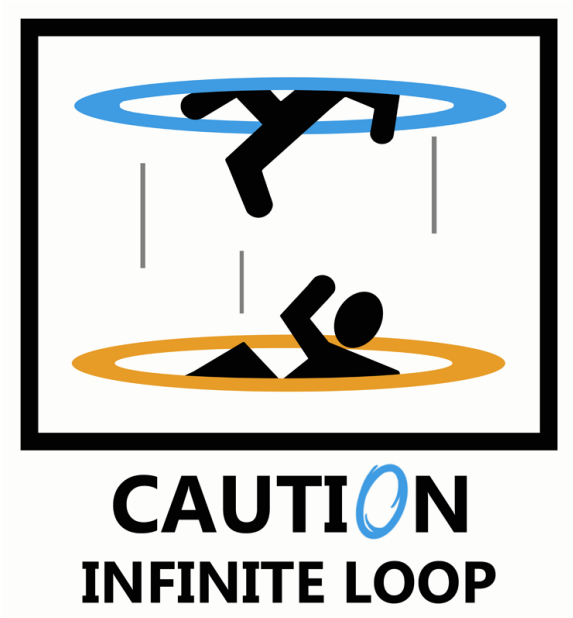
Logiciel R et programmation

Boucles et calculs vectoriels

Ewen Gallic
Université de Rennes 1, 2014 - 2015

Les boucles



Source : Portal - Caution Infinite Loop by caycowa, on Deviantart

Les boucles avec `while()`

- Répéter une instruction tant qu'une condition d'arrêt n'est pas honorée ;
- Nombre de répétitions inconnu *a priori*.

```
while(condition) instruction
```

- `condition` doit être un logique ;
- `instruction` peut contenir plusieurs instructions, si entouré par des accolades.

Les boucles avec `while()`

```
x <- 100
while(x/3 > 1){
  x <- x/3
}
x/3 > 1
```

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

```
x
```

```
## [1] 1.234568
```

- Attention à bien s'assurer d'avoir une condition qui pourra être honorée ;
- R éprouve des difficultés à arrêter le calcul d'une boucle infinie...

Exercice

- Créer une boucle qui permet de calculer la factorielle d'un nombre.
- Comparer le résultat avec la fonction `factorial()`.

(Aide : on peut s'appuyer sur la valeur d'un compteur...)

Les boucles avec `for()`

- Répéter une instruction un nombre de fois défini *a priori*.

```
for(variable in vector) instruction
```

- `variable` : nom de variable locale à la fonction `for`, qui prendra une valeur différente à chaque itération ;
- `vector` : le vecteur contenant les valeurs que va prendre `variable` ;
- `instruction`: l'instruction à répéter à chaque itération (si entre accolades, permet d'évaluer plusieurs instructions).

Les boucles avec `for()`

Dans l'exemple suivant, on affiche dans la console, un par un, les noms du vecteur.

```
for(nom in  c("Sonia", "Anne-Hélène", "Julien-Yacine")) print(nom)
```

```
## [1] "Sonia"
## [1] "Anne-Hélène"
## [1] "Julien-Yacine"
```

Les boucles avec `for()`

- Les boucles avec `for()` sont pratiques pour remplir les éléments d'une liste ou d'un vecteur ;
- La création *a priori* de l'objet à remplir, à la taille souhaitée est recommandée ;
- En effet, l'allocation en mémoire est plus efficace de la sorte, plutôt que de laisser `R` créer un objet plus long à chaque itération.

```
# Manière peu efficace
resultat <- NULL
for(i in seq_len(3)) {
  resultat[i] <- i
}
resultat
```

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

Les boucles avec `for()`

```
# Manière plus économique
resultat <- rep(NA, 3)
for(i in seq_len(3)) {
  resultat[i] <- i
}
resultat
```

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

Exercice

1. Construire une boucle avec `for()` pour calculer la factorielle d'un nombre.

Les conditions

- Les conditions sont primordiales en programmation ;
- Elles permettent d'exécuter du code, ou de retourner des objets, à condition de remplir certains critères.

Les conditions : instruction `if`... `else`

- Les instructions `if` et `else` fournissent un moyen d'exécuter du code si une condition est respectée ou non.
- Deux syntaxes possibles :

```
# Première forme (pas de code si condition == FALSE)
if (condition) instruction

# Seconde forme
if (condition) instruction si vrai else instruction si faux
```

- `condition` : un logique ;
- `instruction` : du code à évaluer en fonction de la condition.

Les conditions : instruction `if`... `else`

- Exemple avec une condition simple

```
x <- 2
if(x == 2) print("Hello")
```

```
## [1] "Hello"
```

```
x <- 3
if(x == 2) print("Hello")
```

Les conditions : instruction `if`... `else`

- Exemple avec des instructions dans le cas contraire

```
if(x == 2) print("Hello") else print("x est différent de 2")
```

```
## [1] "x est différent de 2"
```

```
if(x == 2){
  print("Hello")
} else {# x != 2
  x <- x-1
  print(paste0("La nouvelle valeur de x : ", x))
}
```

```
## [1] "La nouvelle valeur de x : 2"
```

Exercice

1. Créer une fonction qui retourne `TRUE` si un nombre `x` est divisible par une autre nombre `diviseur`, et `FALSE` sinon.

Les conditions : la fonction `switch()`

- La fonction `switch()` permet d'éxecuter du code en fonction d'une valeur prise par une variable.

```
switch(valeur_test,
       cas_1 = {
         instruction_cas_1
       },
       cas_2 = {
         instruction_cas_2
       },
       ...
  )
```

- `valeur_test` : nombre ou chaîne de caractère ;
- Si `valeur_test` vaut `cas_1`, évaluer `instruction_cas_1` ;
- Si `valeur_test` vaut `cas_2`, évaluer `instruction_cas_2`.

Les conditions : la fonction `switch()`

```
centre <- function(x, type) {
  switch(type,
         mean = mean(x),
         median = median(x))
}
x <- rcauchy(10)
centre(x, "mean")
```

```
## [1] -1.082508
```

```
centre(x, "median")
```

```
## [1] -0.1342556
```

L'intstruction `repeat`... `break`

- `repeat`... `break` permet de répéter une expression ;
- La présence d'un test d'arrêt est nécessaire, à l'aide de l'instruction `break`.

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

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

L'intstruction `next`... `break`

- `next`... `break` permet de passer immédiatement à l'itération suivante d'une boucle `for`, `while` ou `repeat`.

```
resul <- rep(NA, 10)
for(i in 1:10) {
  if(i == 5) next
  resul[i] <- i
}
# Le 5e élément de resul est resté non-disponible
resul
```

```
##  [1]  1  2  3  4 NA  6  7  8  9 10
```

La vectorisation

Source : U.S. Navy National Museum of Naval Aviation

La vectorisation

- Les boucles sont lentes en `R` ;
- De nombreux cas où elles peuvent être évitées ;
- Emploi des calculs vectoriels.

La vectorisation

- Voici un exemple d'un calcul en boucle que l'on peut vectoriser :

```
# Somme des logarithmes des 10 premiers entiers
somme_log <- 0
for(i in seq_len(10)){
  somme_log <- somme_log + log(i)
}
somme_log
```

```
## [1] 15.10441
```

- À chaque itération, on modifie la valeur de `somme_log` pour lui ajouter la valeur du logarithme d'un entier `i`.

La vectorisation

- La même chose en vectorisant :

```
# En vectorisant le calcul
sum(log(seq_len(10)))
```

```
## [1] 15.10441
```

- La fonction `log()` est appliquée à tous les éléments de `seq_len(10)` ;
- Puis la fonction `sum()` se charge d'additionner tous les éléments entre eux.

La vectorisation

- Nous allons considérer des fonctions qui prennent une fonction en `input`, et retournent un vecteur en `output` ;
- Deux sources :
  - les fonctions issues du *package* `plyr`,
  - certains équivalents présents dans le *package* `base`.

Les fonctions du *package* `plyr`

- Charger le *package* : `library(plyr)` ;
- Des noms faciles à retenir :
  - première lettre : format d'entrée des données ;
  - seconde lettre : format de sortie souhaité ;
  - suffixe `ply`.

TYPE	ARRAY	DATA.FRAME	LIST
array	`aaply()`	`adply()`	`alply()`
data.frame	`daply()`	`ddply()`	`dlply()`
list	`laply()`	`ldply()`	`llply()`

- Les paramètres de ces fonctions commencent par un point `.`.

Les fonctions du *package* `plyr`: `array` en input

- `aaply()`, `adply()` ou `alply()` ;
- Application d'une fonction à chaque portion d'un `array` ;
- Puis joignent le résultat sous forme de : `array`, `data.frame` ou `list` ;
- Les paramètres principaux sont :
  - `.data` : les données d'*input*,
  - `.margins` : la manière de découper le tableau ;
  - `.fun` : la fonction à appliquer à chaque portion du tableau.

Les fonctions du *package* `plyr`: `array` en input

Le paramètre `.margins` peut prendre les valeurs suivantes :

- `.margins = 1` : par lignes ;
- `.margins = 2` : par colonnes ;
- `.margins = c(1,2)` : par cellule ;
- `.margins = c()` : ne pas faire de découpement ;

Les fonctions du *package* `plyr` : `data.frame` en input

- Format très fréquent en entrée ;
- `daply()`, `ddply()`, `dlply()` ;
- Application d'une fonction à chaque partie d'un tableau ;
- Puis rassemblement des résultats obtenus dans un `array`, un `data.frame` ou une `list` ;
- Les paramètres principaux sont :
  - `.data` : les données d'*input*,
  - `.variables` : la ou les variables servant à découper le tableau pour faire des sous-tableaux,
  - `.fun` : la fonction à appliquer à chaque portion du tableau.

Les fonctions du *package* `plyr` : `data.frame` en input

```
chomage <- data.frame(region = rep(c(rep("Bretagne", 4), rep("Corse", 2)), 2),
                      departement = rep(c("Cotes-d'Armor", "Finistere",
                                          "Ille-et-Vilaine", "Morbihan",
                                          "Corse-du-Sud", "Haute-Corse"), 2),
                      annee = rep(c(2011, 2010), each = 6),
                      ouvriers = c(8738, 12701, 11390, 10228, 975, 1297,
                                   8113, 12258, 10897, 9617, 936, 1220),
                      ingenieurs = c(1420, 2530, 3986, 2025, 259, 254,
                                     1334, 2401, 3776, 1979, 253, 241))
```

Les fonctions du *package* `plyr` : `data.frame` en input

- Total des chômeurs en Bretagne et en Corse pour les années 2010 et 2011 ;
- Résultat sous forme de `data.frame` :

```
ddply(chomage, .(annee), summarise, total_chomeurs = sum(ouvriers + ingenieurs))
```

```
##   annee total_chomeurs
## 1  2010          53025
## 2  2011          55803
```

Les fonctions du *package* `plyr` : `data.frame` en input

- Total des chômeurs en Bretagne et en Corse pour les années 2010 et 2011 ;
- Résultat sous forme de `tableau` :

```
daply(chomage, .(annee), summarise, total_chomeurs = sum(ouvriers + ingenieurs))
```

```
## $`2010`
## [1] 53025
##
## $`2011`
## [1] 55803
```

Les fonctions du *package* `plyr` : `data.frame` en input

- Total des chômeurs en Bretagne et en Corse pour les années 2010 et 2011 ;
- Résultat sous forme de `list` :

```
dlply(chomage, .(annee), summarise, total_chomeurs = sum(ouvriers + ingenieurs))
```

```
## $`2010`
##   total_chomeurs
## 1          53025
##
## $`2011`
##   total_chomeurs
## 1          55803
##
## attr(,"split_type")
## [1] "data.frame"
## attr(,"split_labels")
##   annee
## 1  2010
## 2  2011
```

Les fonctions du *package* `plyr` : `data.frame` en input

- Total des chômeurs pour les années 2010 et 2011, par région :

```
ddply(chomage, .(annee, region), summarise, total_chomeurs = sum(ouvriers + ingenieurs))
```

```
##   annee   region total_chomeurs
## 1  2010 Bretagne          50375
## 2  2010    Corse           2650
## 3  2011 Bretagne          53018
## 4  2011    Corse           2785
```

Les fonctions du *package* `plyr` : `data.frame` en input

- Nombre d'observations pour chaque groupe :

```
ddply(chomage, .(annee, region), nrow)
```

```
##   annee   region V1
## 1  2010 Bretagne  4
## 2  2010    Corse  2
## 3  2011 Bretagne  4
## 4  2011    Corse  2
```

Les fonctions du *package* `plyr` : `data.frame` en input

- En utilisant une fonction définie par l'utilisateur

```
ddply(chomage, .(annee, region), function(x){
  moy_ouvriers <- mean(x$ouvriers)
  moy_ingenieurs <- mean(x$ingenieurs)
  data.frame(moy_ouvriers = moy_ouvriers, moy_ingenieurs = moy_ingenieurs)
})
```

```
##   annee   region moy_ouvriers moy_ingenieurs
## 1  2010 Bretagne     10221.25        2372.50
## 2  2010    Corse      1078.00         247.00
## 3  2011 Bretagne     10764.25        2490.25
## 4  2011    Corse      1136.00         256.50
```

Exercice

1. Importer dans `R` les données sur les salaires de professeurs d'Universités depuis la source suivante : http://data.princeton.edu/wws509/datasets/salary.dat (la première ligne contient les noms des variables) ;
2. Calculer le salaire (`sl`) moyen par sexe (`sx`) ;
3. Calculer le salaire (`sl`) moyen par sexe (`sx`) et niveau d'études (`dg`) ;
4. Même question, mais ajouter également le salaire médian.

Les fonctions du *package* `plyr` : `list` en input

- Format très fréquent en entrée ;
- `laply()`, `ldply()`, `llply()` ;
- Application d'une fonction à chaque élément d'une liste ;
- Puis rassemblement des résultats obtenus dans un `array`, un `data.frame` ou une `list` ;
- Les paramètres principaux sont :
  - `.data` : les données d'*input*,
  - `.fun` : la fonction à appliquer à chaque portion du tableau.
- Pas de paramétrage à effectuer pour le découpage !

Les fonctions du *package* `plyr` : `list` en input

```
set.seed(1)
liste <-list(normale =rnorm(10),
             logiques =c(TRUE, TRUE, FALSE), x =c(0,NA, 3))
```

- Obtenir la longueur de chaque élément de la liste

```
laply(liste, length)
```

```
## [1] 10  3  3
```

Les fonctions du *package* `plyr` : `list` en input

```
ldply(liste, length)
```

```
##         .id V1
## 1   normale 10
## 2 logiques  3
## 3         x  3
```

```
llply(liste, length)
```

```
## $normale
## [1] 10
##
## $logiques
## [1] 3
##
## $x
## [1] 3
```

Les fonctions du *package* `plyr` : `list` en input

- La moyenne pour chaque élément :

```
unlist(llply(liste, mean, na.rm = TRUE))
```

```
##   normale  logiques         x
## 0.1322028 0.6666667 1.5000000
```

Les fonctions du *package* `plyr` : `list` en input

- Appliquer une fonction définie par l'utilisateur

```
llply(liste,function(x, y) x /mean(x, na.rm = TRUE) + y, y = 2)
```

```
## $normale
##  [1] -2.7385827  3.3891033 -4.3208096 14.0669232  4.4924421 -4.2061356
##  [7]  5.6869803  7.5847895  6.3552892 -0.3099997
##
## $logiques
## [1] 3.5 3.5 2.0
##
## $x
## [1]  2 NA  4
```

Exercice

1. Créer une liste de longueur 5 dans laquelle chaque élément doit être composé d'un échantillon d'observations issues d'une loi Normale centrée réduite ;
2. Sur chaque échantillon de l'objet créé, calculer la moyenne et l'écart-type. Le résultat doit être sous forme de `data.frame`.

Les fonctions du *package* `plyr` : calculs parallèles

- Il existe des méthodes en R pour effectuer les calculs parallèles ;
- Ces calculs peuvent être lancés sur plusieurs coeurs du processeur de la machine ;
- Mais également sur des postes distants, ou sur le cloud ;
- Les fonctions du *package* `plyr` proposent d'effectuer les calculs sur différents coeurs de processeur, en donnant la valeur `TRUE` au paramètre `.parallel` ;
- Il est cependant nécessaire de charger au préalable `doMC` (ou `doSMP` pour les utilisateurs de Window) ;
- Voir le poly pour un exemple.

Les fonctions de la famille `apply` du *package* `base`

- Il existe des fonctions du même type que celles du *package* `plyr`, dans le *package* `base` ;
- Elles sont cependant moins uniformisées dans leur syntaxe ;

FONCTION	INPUT	OUTPUT
`apply()`	Matrice ou tableau	Vecteur ou tableau ou liste
`lapply()`	Liste ou vecteur	Liste
`sapply()`	Liste ou vecteur	Vecteur ou matrice ou liste
`vapply()`	Liste ou vecteur	Vecteur ou matrice ou liste
`tapply()`	Tableau et facteurs	Tableau ou liste
`mapply()`	Listes et/ou vecteurs	Vecteur ou matrice ou liste

Les fonctions de la famille `apply` du *package* `base` : `lapply()`

- Applique une fonction à chaque élément de la liste ou du vecteur fourni en paramètre ;
- Retourne le résultat sous une forme de liste.

```
lapply(X, FUN, ...)
```

- `X` : vecteur ou liste donné en paramètres ;
- `FUN` : fonction à appliquer à chaque élément ;
- `...` : permet de fournir des paramètres à des fonctions imbriquées dans `FUN`.

Les fonctions de la famille `apply` du *package* `base` : `lapply()`

```
liste <- list(normale = rnorm(10), logiques = c(TRUE, TRUE, FALSE), x = c(0, NA, 3))
```

- Liste contenant la longueur des éléments de `liste` :

```
lapply(liste, length)
```

```
## $normale
## [1] 10
##
## $logiques
## [1] 3
##
## $x
## [1] 3
```

Les fonctions de la famille `apply` du *package* `base` : `lapply()`

- Calcul de la moyenne pour chaque élément

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

```
## $normale
## [1] 0.248845
##
## $logiques
## [1] 0.6666667
##
## $x
## [1] 1.5
```

Les fonctions de la famille `apply` du *package* `base` : `lapply()`

- Utilisation d'une fonction définie par l'utilisateur :

```
lapply(liste, function(x) x / mean(x, na.rm = TRUE))
```

```
## $normale
##  [1]  6.07519277  1.56661087 -2.49649643 -8.89991820  4.52060941 -0.18056868
##  [7] -0.06506164  3.79286833  3.30013177  2.38663180
##
## $logiques
## [1] 1.5 1.5 0.0
##
## $x
## [1]  0 NA  2
```

Les fonctions de la famille `apply` du *package* `base` : `lapply()`

- La fonction définie peut utiliser plusieurs paramètres :

```
lapply(liste, function(x, y) x / mean(x, na.rm = TRUE) + y, y = 2)
```

```
## $normale
##  [1]  8.0751928  3.5666109 -0.4964964 -6.8999182  6.5206094  1.8194313
##  [7]  1.9349384  5.7928683  5.3001318  4.3866318
##
## $logiques
## [1] 3.5 3.5 2.0
##
## $x
## [1]  2 NA  4
```

Les fonctions de la famille `apply` du *package* `base` : `lapply()`

- Appliquer `lapply()` sur un `data.frame` (qui est une liste), permet d'appliquer une fonction sur chaque colonne de ce `data.frame` :
- Par exemple, pour afficher le mode de chaque colonne du `data.frame` `cars` :

```
unlist(lapply(cars, class))
```

```
##     speed      dist
## "numeric" "numeric"
```

- Ou encore pour calculer la moyenne pour chaque colonne :

```
unlist(lapply(cars, mean))
```

```
## speed  dist
## 15.40 42.98
```

Les fonctions de la famille `apply` du *package* `base` : `lapply()`

- Il arrive de vouloir rassembler le résultat fourni par `lapply()` dans un `data.frame` ;
- La fonction `do.call()` permet de réaliser ce tour de passe-passe :

```
l <- lapply(1:3, function(x) cbind(valeur = x, lettre = LETTERS[x]))
data.frame(do.call("rbind", l))
```

```
##   valeur lettre
## 1      1      A
## 2      2      B
## 3      3      C
```

- L'appel de `do.call("rbind", x)` revient à faire `rbind(x[1], x[2], ..., x[n])` avec `x` un vecteur de taile `n`.

Exercice

1. Créer une liste de longueur 5 dans laquelle chaque élément doit être composé d'un échantillon d'observations issues d'une loi Normale centrée réduite ;
2. Sur chaque échantillon de l'objet créé, calculer la moyenne et l'écart-type. Le résultat doit être sous forme de `data.frame`.

Les fonctions de la famille `apply` du *package* `base` : `sapply()`

- Applique une fonction à chaque élément de la liste ou du vecteur fourni en paramètre ;
- Retourne le résultat sous une forme de liste, de vecteur ou de matrice.

```
sapply(X, FUN, ..., simplify, USE.NAMES)
```

- `X` : vecteur ou liste donné en paramètres ;
- `FUN` : fonction à appliquer à chaque élément ;
- `...` : permet de fournir des paramètres à des fonctions imbriquées dans `FUN` ;
- `simplify` : si `FALSE`, même résultat que `lapply()` ; sinon (par défaut), `R` tente de retourner le résultat sous une forme simplifiée :
  - si les résultats de la fonction `FUN` sont des scalaires : vecteur,
  - si les éléments retournés par la fonction `FUN` sont de même taille : matrice ;
- `USE.NAMES` : si `TRUE`, et si `X` est de type `character`, utilise `X` comme nom pour le résultat (à moins que le résultat possède déjà des noms).

Les fonctions de la famille `apply` du *package* `base` : `sapply()`

```
(x <- list(a = 1:10, beta = exp(-3:3), logic = c(TRUE,FALSE,FALSE,TRUE)))
```

```
## $a
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $beta
## [1]  0.04978707  0.13533528  0.36787944  1.00000000  2.71828183  7.38905610
## [7] 20.08553692
##
## $logic
## [1]  TRUE FALSE FALSE  TRUE
```

Les fonctions de la famille `apply` du *package* `base` : `sapply()`

- Application de la fonction `quantile()` à chaque élément ;
- Retourne la médiane et les quartiles ;
- Avec `lapply()` :

```
lapply(x, quantile)
```

```
## $a
##    0%   25%   50%   75%  100%
##  1.00  3.25  5.50  7.75 10.00
##
## $beta
##          0%         25%         50%         75%        100%
##  0.04978707  0.25160736  1.00000000  5.05366896 20.08553692
##
## $logic
##   0%  25%  50%  75% 100%
##  0.0  0.0  0.5  1.0  1.0
```

Les fonctions de la famille `apply` du *package* `base` : `sapply()`

- Avec `sapply()` :

```
sapply(x, quantile)
```

```
##          a        beta logic
## 0%    1.00  0.04978707   0.0
## 25%   3.25  0.25160736   0.0
## 50%   5.50  1.00000000   0.5
## 75%   7.75  5.05366896   1.0
## 100% 10.00 20.08553692   1.0
```

Les fonctions de la famille `apply` du *package* `base` : `sapply()`

- Un exemple d'utilisation de `USE.NAMES` :

```
sapply(LETTERS[1:3], nchar)
```

```
## A B C
## 1 1 1
```

```
sapply(LETTERS[1:3], nchar, USE.NAMES = FALSE)
```

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

Les fonctions de la famille `apply` du *package* `base` : `vapply()`

- `vapply()` similaire à `sapply()` ;
- Possède un type de valeurs spécifiées ;
- Utilisation plus sûre et parfois plus rapide ;
- Si `data.frame` en entrée, même résultat que `sapply()` ;
- Si liste vide en entrée, retourne un logique de longueur nulle.

```
vapply(X, FUN, FUN.VALUE, ..., USE.NAMES)
```

- `X` : vecteur ou liste donné en paramètres ;
- `FUN` : fonction à appliquer à chaque élément ;
- `FUN.VALUE` : vecteur masque pour la valeur retournée par `FUN` ;
- `...` : permet de fournir des paramètres à des fonctions imbriquées dans `FUN` ;
- `USE.NAMES` : si `TRUE`, et si `X` est de type `character`, utilise `X` comme nom pour le résultat (à moins que le résultat possède déjà des noms).

Les fonctions de la famille `apply` du *package* `base` : `vapply()`

```
sapply(cars, is.numeric)
```

```
## speed  dist
##  TRUE  TRUE
```

```
vapply(cars, is.numeric, FUN.VALUE = logical(1))
```

```
## speed  dist
##  TRUE  TRUE
```

Les fonctions de la famille `apply` du *package* `base` : `vapply()`

- Avec la liste vide :

```
sapply(list(), is.numeric)
```

```
## list()
```

```
vapply(list(), is.numeric, FUN.VALUE = logical(1))
```

```
## logical(0)
```

Les fonctions de la famille `apply` du *package* `base` : `apply()`

- Appliquer une fonction à une partie d'un `data.frame` ;
- Le résultat est un vecteur ou un tableau, ou encore une liste.

```
apply(X, MARGIN, FUN, ...)
```

- `X` : matrice ou `data.frame` donné en paramètre ;
- `MARGIN` : si `1`, `FUN` est appliquée aux lignes, si `2`, `FUN` est appliqué aux colonnes ;
- `FUN` : fonction à appliquer à chaque élément ;
- `...` : permet de fournir des paramètres à des fonctions imbriquées dans `FUN`.

Les fonctions de la famille `apply` du *package* `base` : `apply()`

```
(X <- matrix(1:9, ncol = 3))
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

- Somme par ligne

```
apply(X, MARGIN = 1, sum)
```

```
## [1] 12 15 18
```

Les fonctions de la famille `apply` du *package* `base` : `apply()`

- Somme par colonne

```
apply(X, MARGIN = 2, sum)
```

```
## [1]  6 15 24
```

- Fonction définie par l'utilisateur

```
apply(X, MARGIN = 1, function(x) sum(x) / sum(X))
```

```
## [1] 0.2666667 0.3333333 0.4000000
```

Exercice

1. Importer dans `R` les données sur les salaires de professeurs d'Universités depuis la source suivante : http://data.princeton.edu/wws509/datasets/salary.dat (la première ligne contient les noms des variables) ;
2. Calculer le salaire (`sl`) moyen par sexe (`sx`) ;
3. Calculer le salaire (`sl`) moyen par sexe (`sx`) et niveau d'études (`dg`) ;
4. Même question, mais ajouter également le salaire médian.

Les fonctions de la famille `apply` du *package* `base` : `tapply()`

- `tapply()` s'applique à chaque cellule d'un tableau ;
- La fonction s'applique à des regroupements faits selon une liste de facteurs.

```
tapply(X,INDEX, FUN, ..., simplify)
```

- `X` : `data.frame` donné en paramètre ;
- `INDEX` : liste d'un ou plusieurs facteurs, de même longueur que `X` ;
- `FUN` : fonction à appliquer à chaque élément ;
- `...` : permet de fournir des paramètres à des fonctions imbriquées dans `FUN` ;
- `simplify` : si `TRUE` (par défaut), retourne un tableau de scalaires, si `FALSE`, le résultat est un tableau de mode `list`.

Les fonctions de la famille `apply` du *package* `base` : `tapply()`

```
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

Les fonctions de la famille `apply` du *package* `base` : `tapply()`

- Moyenne de la longueur des sépales par espèce :

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

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

Les fonctions de la famille `apply` du *package* `base` : `tapply()`

- Moyenne de la longueur des sépales par espèce, résultat sous forme de liste :

```
tapply(iris$Sepal.Length, iris$Species, mean, simplify = FALSE)
```

```
## $setosa
## [1] 5.006
##
## $versicolor
## [1] 5.936
##
## $virginica
## [1] 6.588
```

Les fonctions de la famille `apply` du *package* `base` : `mapply()`

- `mapply()` applique une fonction à plusieurs listes ou vecteurs.

```
mapply(FUN, ..., MoreArgs, SIMPLIFY, USE.NAMES)
```

- `FUN` : fonction à appliquer aux vecteurs ou listes fournies (*via* `...`) ;
- `MoreArgs` : liste de paramètres supplémentaires à fournir à la fonction à appliquer ;
- `SIMPLIFY` : même usage que pour `sapply()` ;
- `USE.NAMES` : même usage que pour `sapply()`.

Les fonctions de la famille `apply` du *package* `base` : `mapply()`

```
(l1 <- list(a = c(1:5), b = c(6:10)))
```

```
## $a
## [1] 1 2 3 4 5
##
## $b
## [1]  6  7  8  9 10
```

```
(l2 <- list(c = c(11:15), d = c(16:20)))
```

```
## $c
## [1] 11 12 13 14 15
##
## $d
## [1] 16 17 18 19 20
```

Les fonctions de la famille `apply` du *package* `base` : `mapply()`

- Somme des éléments correspondants de `l1` et `l2` :

```
mapply(sum, l1$a, l1$b, l2$c, l2$d)
```

```
## [1] 34 38 42 46 50
```

Les fonctions de la famille `apply` du *package* `base` : `mapply()`

- Attention au recyclage silencieux !

```
(l1 <- list(a = c(1:5), b = c(6:20)))
```

```
## $a
## [1] 1 2 3 4 5
##
## $b
##  [1]  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

```
mapply(sum, l1$a, l1$b, l2$c, l2$d)
```

```
##  [1] 34 38 42 46 50 39 43 47 51 55 44 48 52 56 60
```

La fonction `Vectorize()`

- `Vectorize()` permet de convertir une fonction scalaire en une fonction vectorielle ;
- N'améliore cependant pa la rapidité d'exécution du code...
- Mais son utilisation permet d'éviter des lignes de codes !

```
Vectorize(FUN, vectorize.args, SIMPLIFY, USE.NAMES)
```

- `FUN` : fonction à appliquer à chaque élément ;
- `vectorize.args` : vecteur de paramètres (de type `character`) qui devaient être vectorisés (par défaut, tous les paramètres de `FUN`) ;
- `SIMPLIFY` : même usage que pour `sapply()` ;
- `USE.NAMES` : même usage que pour `sapply()`.

La fonction `Vectorize()`

- Une fonction scalaire simple :

```
f <- function(x = 1:3, y) c(x, y)
```

- On la "vectorise" :

```
vf <- Vectorize(f, SIMPLIFY = FALSE)
```

La fonction `Vectorize()`

```
f(1:3, 1:3)
```

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

```
vf(1:3, 1:3)
```

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

La fonction `Vectorize()`

- Vectorise seulement y, pas x

```
vf(y = 1:3)
```

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