

Prétraitement de données en R

Sophie Baillargeon, Université Laval

2019-01-25

Table des matières

Transformation de variables	2
Modification ou ajout d'une variable dans un data frame	3
Variable conditionnelle à une autre avec la fonction <code>ifelse</code>	4
Catégorisation d'une variable numérique	5
Fonction <code>cut</code>	5
Fonction <code>ave</code>	6
Normalisation de valeurs numériques	6
Fonction <code>scale</code>	6
Manipulation de chaînes de caractères	8
Fonction <code>paste</code>	9
Fonction <code>nchar</code>	9
Fonctions <code>toupper</code> , <code>tolower</code> et <code>casefold</code>	9
Fonction <code>strsplit</code>	9
Fonction <code>substr</code>	10
Fonctions <code>sub</code> et <code>gsub</code>	10
Fonctions <code>grep</code> et <code>grepl</code>	11
Fonction <code>chartr</code>	11
Fonction <code>iconv</code>	12
Manipulation de dates	12
Fonctions <code>as.Date</code> et <code>Sys.getlocale</code>	12
Fonctions <code>difftime</code> et <code>Sys.Date</code>	13
Fonction <code>format</code>	14
Manipulation de jeux de données	14
Fusion de données	15
Concaténation d'observations (lignes)	15
Concaténation de variables (colonnes)	15
Fusion par association - la fonction <code>merge</code>	16
Modification de l'ordre des données	18
Fonction <code>rev</code>	18
Fonction <code>sort</code>	18
Fonction <code>order</code>	19
Fonction <code>arrange</code> du package <code>dplyr</code>	21
Changement de mise en forme de jeux de données	22
Transposition avec la fonction <code>t</code>	22
Comparaison des mises en forme large et longue	23
Transformation entre les mises en forme large et longue avec les fonctions <code>stack</code> et <code>unstack</code>	27
Transformation entre les mises en forme large et longue avec la fonction <code>reshape</code>	28
Transformation entre les mises en forme large et longue avec les fonctions <code>gather</code> et <code>spread</code> du package <code>tidyr</code>	33
Exemples supplémentaires de changement de mise en forme	34
Synthèse	39
Références	41

Maintenant que nous savons comment créer un objet contenant les données que nous voulons analyser, il nous reste un volet à couvrir dans la manipulation de données en R : le prétraitement des données. Ce prétraitement est défini ici comme étant toutes les étapes nécessaires pour rendre des données prêtes à être fournies en entrée à une certaine fonction, par exemple une fonction pour produire un graphique ou ajuster un modèle. Dans la grande majorité des cas, les données importées doivent être adaptées afin d'être utilisables. Préparer des données peut impliquer :

- nettoyer les données (repérer et corriger des erreurs) ;
- modifier les observations d'une variable ;
- créer de nouvelles variables ;
- mettre en commun des jeux de données ;
- réordonner des observations ;
- modifier la mise en forme d'un jeu de données ;
- etc.

Voici comment réaliser certains de ces prétraitements en R.

Transformation de variables

Un grand nombre de transformations de variables usuelles peuvent être facilement réalisées en R, par exemple les transformations suivantes :

- conversion de format de données,
- modification de chaînes de caractères (p. ex. remplacement ou retrait des caractères, modification de la casse (majuscule ou minuscule))
- extraction d'une sous-chaîne de caractères,
- changement de libellés pour les niveaux d'un facteur,
- regroupement des niveaux d'un facteur,
- transformation mathématique de variables numériques (p. ex. normalisation de valeurs),
- arrondissement ou troncature de valeurs numériques,
- etc.

Une transformation de variables peut résulter en une modification des variables dans un jeu de données ou en la création de nouvelles variables à partir de celles existantes.

Supposons ici que nous travaillons avec un jeu de données stocké dans un data frame R. Les colonnes du data frame représentent des variables. Pour illustrer les transformations présentées, nous utiliserons le jeu de données `cars` du package `dataset`. Il s'agit d'un petit jeu de données à 50 observations et 2 variables.

```
str(cars)
```

```
## 'data.frame':   50 obs. of  2 variables:
## $ speed: num  4 4 7 7 8 9 10 10 10 11 ...
## $ dist : num  2 10 4 22 16 10 18 26 34 17 ...
```

Il s'agit d'observations prises sur des voitures. La distance parcourue (variable `dist`) par ces voitures entre le moment d'un freinage et l'atteinte d'une immobilisation complète a été mesurée. La vitesse de déplacement (variable `speed`) des voitures tout juste avant de freiner a aussi été mesurée.

Nous allons travailler sur une copie de ce jeu de données afin de ne pas modifier le jeu de données original.

```
cars2 <- cars
```

Modification ou ajout d'une variable dans un data frame

Pour remplacer une variable dans un data frame par une transformation de celle-ci, il suffit d'assigner le nouveau vecteur ou facteur contenant les observations de la variable transformée à la colonne contenant la variable à modifier.

Par exemple, dans le jeu de données `cars`, la distance est exprimée en pieds. Supposons que nous voulons transformer l'échelle de mesure de cette variable pour des mètres. Cette transformation est effectuée par une simple opération mathématique : la multiplication par un facteur de conversion. Ce facteur de conversion est ici 0.3048, car un pied est l'équivalent de 0.3048 mètre.

```
cars2$dist * 0.3048
```

Pour modifier la variable sans en créer une nouvelle, il suffit d'assigner le vecteur contenant les données pour la variable transformée à la colonne qui contenait la variable d'origine.

```
cars2$dist <- cars2$dist * 0.3048
```

Nous savons que l'instruction `cars2$dist`, servant à identifier la colonne à modifier, aurait pu être écrite de bien d'autres façons, notamment :

- `cars2[["dist"]]`
- `cars2[, "dist"]`
- `cars2[, 2]`
- etc.

Je considère cependant que l'opérateur `$` procure la syntaxe la plus simple pour un remplacement de variable dans un data frame. Le jeu de données `cars2` a été modifié comme suit.

```
str(cars2)
```

```
## 'data.frame':   50 obs. of  2 variables:
## $ speed: num  4 4 7 7 8 9 10 10 10 11 ...
## $ dist : num  0.61 3.05 1.22 6.71 4.88 ...
```

La fonction `transform` offre une autre façon d'écrire ce genre d'assignation. La commande suivante produit le même résultat que `cars2$dist <- cars2$dist * 0.3048`.

```
cars2 <- transform(cars2, dist = dist * 0.3048)
```

Il faut donner comme premier argument à la fonction `transform` le data frame dans lequel nous souhaitons effectuer une transformation de variable. Les arguments suivants sont des assignations de nouvelles valeurs. Dans ces assignations, il est possible de référer aux colonnes du data frame par leur nom sans les précéder de `nom_du_data_frame$` et sans les encadrer de guillemets, comme avec la fonction `subset`, vue dans le [cours sur les structures de données en R](#).

Plutôt que d'écraser une colonne d'un data frame en la remplaçant par un autre objet, comme nous venons de le faire, il sera souvent préférable d'ajouter les nouvelles variables créées tout en conservant les anciennes. Dans ce cas, il suffit de concaténer les nouvelles variables au data frame avec la fonction `cbind` ou `data.frame`, ou d'assigner les nouvelles variables à des colonnes portant des noms qui ne sont pas initialement présents dans le data frame. Par exemple, pour ajouter une variable contenant la distance exprimée en mètres au data frame `cars2`, nous pourrions procéder comme suit.

```
cars2 <- cars # pour repartir avec la version originale du jeu de données
cars2$dist_m <- cars2$dist * 0.3048
head(cars2)
```

```
##   speed dist dist_m
## 1     4    2 0.6096
## 2     4   10 3.0480
## 3     7    4 1.2192
```

```
## 4      7      22 6.7056
## 5      8      16 4.8768
## 6      9      10 3.0480
```

Étant donné que le jeu de données `cars2` ne contenait pas à l'origine de colonne nommée `dist_m`, une nouvelle colonne portant ce nom a été créée et ajoutée après les colonnes déjà présentes dans le data frame. La même tâche aurait pu être effectuée en utilisant la commande `data.frame` comme suit.

```
cars2 <- cars # encore pour repartir avec la version originale du jeu de données
cars2 <- data.frame(cars2, dist_m = cars2$dist * 0.3048)
```

La fonction `transform` permet aussi d'ajouter de nouvelles variables par une assignation à un nouveau nom de colonne.

Il y a souvent plusieurs façons de réaliser une même tâche en R. Personnellement, je tends à adopter la façon qui présente le meilleur compromis entre un code le plus facile à comprendre possible et un code le plus court possible.

Variable conditionnelle à une autre avec la fonction `ifelse`

La fonction `ifelse` est souvent utile pour transformer des variables. Cette fonction prend comme premier argument un vecteur de valeurs logiques. Cet argument est nommé `test`. Elle retourne en sortie un vecteur de même longueur que `test`. Ce vecteur est formé des valeurs dans le deuxième argument (`yes`) aux positions pour lesquelles `test` comporte une valeur `TRUE`, et des valeurs dans le troisième argument (`no`) aux autres positions. Voici un petit exemple pour comprendre le fonctionnement de cette fonction.

```
ifelse(test = c(TRUE, FALSE, FALSE, TRUE),
       yes = c(1, 2, 3, 4),
       no = c(-1, -2, -3, -4))
```

```
## [1] 1 -2 -3 4
```

Par exemple, supposons que nous souhaitons créer une nouvelle variable dans le jeu de données `cars2` contenant la vitesse en km/heure plutôt qu'en miles/heure, mais dans laquelle les valeurs de distance en miles à l'heure inférieures à 10 sont ramenées à des valeurs manquantes. Nous pourrions créer cette variable comme suit (1.60934 est le facteur de conversion entre un mile à l'heure et un kilomètre à l'heure).

```
cars2$speed_km <- ifelse(test = cars$speed < 10, yes = NA, no = cars$speed * 1.60934)
head(cars2, n = 10)
```

```
##      speed dist  dist_m speed_km
## 1      4      2  0.6096      NA
## 2      4     10  3.0480      NA
## 3      7      4  1.2192      NA
## 4      7     22  6.7056      NA
## 5      8     16  4.8768      NA
## 6      9     10  3.0480      NA
## 7     10     18  5.4864 16.09340
## 8     10     26  7.9248 16.09340
## 9     10     34 10.3632 16.09340
## 10    11     17  5.1816 17.70274
```

Dans cet exemple, la valeur fournie au deuxième argument (`yes`) n'est pas un vecteur de même longueur que les valeurs fournies aux arguments `test` et `no`. R applique donc la règle de recyclage pour transformer `NA`, qui est en fait un vecteur de longueur 1, en un vecteur contenant `length(cars$speed) = 50` valeurs `NA`.

Il est même possible d'imbriquer plusieurs appels à la fonction `ifelse`, comme dans cet exemple.

```
cars2$speed_catego <- ifelse(
  test = cars2$speed < 10,
  yes = "moins de 10",
  no = ifelse(
    test = cars2$speed >= 20,
    yes = "20 ou plus",
    no = "entre 10 et 20"
  )
)
str(cars2)
```

```
## 'data.frame': 50 obs. of 5 variables:
## $ speed : num 4 4 7 7 8 9 10 10 10 11 ...
## $ dist : num 2 10 4 22 16 10 18 26 34 17 ...
## $ dist_m : num 0.61 3.05 1.22 6.71 4.88 ...
## $ speed_km : num NA NA NA NA NA ...
## $ speed_catego: chr "moins de 10" "moins de 10" "moins de 10" "moins de 10" ...
```

Voici un extrait du jeu de données pour mieux comprendre de quoi a l'air la nouvelle variable créée.

```
cars2[c(5:7,38:40), c("speed", "speed_catego")]
```

```
##   speed speed_catego
## 5     8   moins de 10
## 6     9   moins de 10
## 7    10  entre 10 et 20
## 38    19  entre 10 et 20
## 39    20    20 ou plus
## 40    20    20 ou plus
```

Nous avons catégorisé la variable `speed` en 3 catégories. Il existe cependant de meilleures façons de faire ce genre de tâche, présentées dans la section suivante.

Catégorisation d'une variable numérique

Parfois, nous avons besoin de rendre catégorique une variable numérique. Les agences de statistiques officielles, par exemple Statistique Canada, font souvent de telles catégorisations avant de rendre des données publiques afin qu'il ne soit plus possible d'identifier les individus auxquels réfèrent les données. Il s'agit d'une technique courante d'anonymisation de données. Ainsi, ils peuvent donner accès à des données tout en respectant les règlements concernant la confidentialité des données.

Fonction `cut` :

La principale fonction pour catégoriser une variable numérique en R est la fonction `cut`. En voici un exemple d'utilisation.

```
cars2$speed_catego2 <- cut(cars$speed, breaks = c(-Inf, 10, 20, Inf), right = FALSE)
str(cars2)
```

```
## 'data.frame': 50 obs. of 6 variables:
## $ speed : num 4 4 7 7 8 9 10 10 10 11 ...
## $ dist : num 2 10 4 22 16 10 18 26 34 17 ...
## $ dist_m : num 0.61 3.05 1.22 6.71 4.88 ...
## $ speed_km : num NA NA NA NA NA ...
## $ speed_catego : chr "moins de 10" "moins de 10" "moins de 10" "moins de 10" ...
## $ speed_catego2: Factor w/ 3 levels "[-Inf,10)","[10,20)","...: 1 1 1 1 1 1 2 2 2 2 ...
```

La nouvelle variable obtenue est un facteur, avec des niveaux aux libellés différents de ceux de la variable `speed_catego`.

```
cars2[c(5:7,38:40), c("speed", "speed_catego" , "speed_catego2")]
```

```
##      speed  speed_catego speed_catego2
## 5      8      moins de 10      [-Inf,10)
## 6      9      moins de 10      [-Inf,10)
## 7     10     entre 10 et 20      [10,20)
## 38     19     entre 10 et 20      [10,20)
## 39     20      20 ou plus      [20, Inf)
## 40     20      20 ou plus      [20, Inf)
```

Nous aurions pu contrôler ces libellés avec l'argument `labels` de la fonction `cut`.

Fonction `ave` :

Si nous souhaitons remplacer les identifiants des nouvelles catégories par une statistique calculée sur les valeurs observées dans chaque catégorie, nous utiliserons la [fonction `ave`](#), comme dans cet exemple.

```
cars2$speed_catego3 <- ave(cars2$speed, cars2$speed_catego, FUN = mean)
```

```
cars2[c(5:7,38:40),c("speed", "speed_catego", "speed_catego2", "speed_catego3")]
```

```
##      speed  speed_catego speed_catego2 speed_catego3
## 5      8      moins de 10      [-Inf,10)      6.50000
## 6      9      moins de 10      [-Inf,10)      6.50000
## 7     10     entre 10 et 20      [10,20)      14.53125
## 38     19     entre 10 et 20      [10,20)      14.53125
## 39     20      20 ou plus      [20, Inf)      22.16667
## 40     20      20 ou plus      [20, Inf)      22.16667
```

La valeur de la nouvelle variable `speed_catego3` est la moyenne des vitesses observées dans chacune des catégories définies par la variable `speed_catego`.

Normalisation de valeurs numériques

La normalisation permet de ramener les mesures de différentes variables sur une même échelle.

Fonction `scale`

La [fonction `scale`](#) permet de normaliser par colonne les valeurs numériques dans une matrice ou un data frame.

Par exemple, revenons au jeu de données `cars` d'origine. Les valeurs pour les deux variables dans le jeu de données `cars` ne sont pas sur la même échelle. En effet, leurs moyennes et écarts-types empiriques ne sont pas similaires. Utilisons la fonction `apply`, vue dans le cours sur les [calculs de base en R](#), pour calculer ces statistiques par variable (= par colonne).

```
# moyennes par variable
apply(cars, MARGIN = 2, FUN = mean)
```

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

```
# écarts-types par variable
apply(cars, MARGIN = 2, FUN = sd)
```

```
##      speed      dist
## 5.287644 25.769377
```

Une normalisation couramment employée consiste à soustraire de valeurs leur moyenne, puis à diviser ces différences par l'écart-type des valeurs d'origine. Cette normalisation est parfois appelée standardisation. Les valeurs sont centrées et réduites. En notation statistique, la formule pour obtenir cette normalisation est la suivante :

$$z_i = \frac{x_i - \bar{x}}{\sigma_x}$$

Standardisons les valeurs du jeu de données `cars` avec la fonction `scale`.

```
cars_standard <- scale(cars)
```

Les valeurs dans chacune des colonnes de `cars_standard` ont une moyenne de 0 et un écart-type de 1. Elles sont donc sur la même échelle.

```
# moyennes par variable
round(apply(cars_standard, MARGIN = 2, FUN = mean), digits = 5)
```

```
## speed dist
##      0      0
```

```
# écarts-types par variable
apply(cars_standard, MARGIN = 2, FUN = sd)
```

```
## speed dist
##      1      1
```

Les arguments `center` et `scale` de la fonction `scale` permettent de contrôler, respectivement, les valeurs soustraites par colonnes et les valeurs par lesquelles les colonnes sont divisées. Il est possible, par exemple, de centrer sans réduire avec `center = TRUE` et `scale = FALSE`.

Un autre exemple de normalisation possible avec `scale` est de ramener les mesures de toutes les variables entre 0 et 1. En notation statistique, la formule pour obtenir cette normalisation est la suivante :

$$y_i = \frac{x_i - \min(x)}{\max(x) - \min(x)}$$

Effectuons cette normalisation sur le jeu de données `cars`.

```
minimums <- apply(cars, MARGIN = 2, FUN = min) # obtention des minimums par variable
maximums <- apply(cars, MARGIN = 2, FUN = max) # obtention des maximums par variable
cars_01norm <- scale(cars, center = minimums, scale = maximums - minimums)
summary(cars_01norm)
```

```
##      speed      dist
## Min.   :0.0000  Min.   :0.0000
## 1st Qu.:0.3810  1st Qu.:0.2034
## Median :0.5238  Median :0.2881
## Mean   :0.5429  Mean   :0.3473
## 3rd Qu.:0.7143  3rd Qu.:0.4576
## Max.   :1.0000  Max.   :1.0000
```

Note : La fonction `sweep` permet aussi de faire simultanément une transformation pour chacune des colonnes d'une matrice. Elle généralise en fait `scale` en permettant d'effectuer la transformation sur n'importe laquelle des dimensions d'un array. Cette fonction ne sera cependant pas approfondie ici.

Manipulation de chaînes de caractères

D'un langage de programmation à l'autre, les chaînes de caractères ne sont pas manipulées de la même façon. Par exemple, si un objet R contient une chaîne de caractères telle la suivante,

```
chaine <- "Bonjour"
```

il n'est pas possible d'extraire disons la deuxième lettre de la chaîne avec la commande suivante.

```
chaine[2]
```

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

Cette commande aurait fonctionné en langage Python. Elle ne fonctionne pas en R, parce que l'objet `chaine` est un vecteur de longueur 1. L'opérateur d'indexage `[]` réfère à des éléments d'un objet. L'objet `chaine` ne contient qu'un seul élément, qui est une chaîne de caractères.

```
str(chaine)
```

```
## chr "Bonjour"
```

```
length(chaine)
```

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

```
chaine[1]
```

```
## [1] "Bonjour"
```

Cette section initie à quelques fonctions du R de base pour manipuler des vecteurs dont les éléments sont des chaînes de caractères. Toutes ces fonctions travaillent de façon vectorielle. Elles effectuent d'un coup une opération sur tous les éléments d'un vecteur.

Ces fonctions seront illustrées en utilisant le data frame `boxoffice` créé dans les notes sur [la lecture et l'écriture dans des fichiers externes à partir de R](#). Ce data frame est composé de colonnes contenant toutes des éléments de type caractère. Il avait été créé en important des données d'une table HTML sur le web à l'aide des commandes suivantes.

```
library(rvest)
url <- "http://pro.boxoffice.com/numbers/all_time"
doc <- read_html(url)
tables <- html_table(doc)
boxoffice <- tables[[1]]
```

Pour m'assurer que tous aient accès à ces données, j'ai déposé une copie des données datant du 25 janvier 2019 dans le haut de [la page web des notes que vous êtes en train de lire](#). Le fichier contenant les données est en format `.rds`, il se nomme `boxoffice_20190125.rds`. Il se trouve dans le fichier compressé `pretraitement_donnees_r_2019.zip`. Après avoir téléchargé ce fichier sur votre ordinateur, et après l'avoir décompressé pour en extraire `boxoffice_20190125.rds`, vous pouvez charger les données en R avec le code suivant (en spécifiant le bon chemin d'accès) :

```
boxoffice <- readRDS("C:/coursR/boxoffice_20190125.rds")
```

Rappelons-nous qu'il s'agit de données concernant les recettes des films ayant générés le plus de revenus dans les cinémas américains.

```
head(boxoffice, n = 3)
```

```
##      X1                                     X2          X3          X4
## 1  1 Star Wars: The Force Awakens (Disney) Dec 18, 2015 $936,662,225
## 2  2                                     Avatar (Fox) Dec 18, 2009 $760,507,625
## 3  3                                     Black Panther (Disney) Feb 16, 2018 $700,059,566
```


Les noms des colonnes du data frame ne sont pas informatifs. Renommons les colonnes du data frame en respectant les noms de colonnes dans la page web d'où il provient.

```
colnames(boxoffice) <- c("Ranking", "Movie (Distributor)", "Release Date", "Gross")
```

Fonction paste :

La fonction `paste` sert à concaténer des éléments de vecteurs. Elle retourne toujours un résultat sous le format caractère. Elle n'est pas illustrée ici, car elle a été présentée dans les notes sur les [structures de données en R](#).

Notons que la fonction `paste0` est en quelque sorte une version de la fonction `paste` avec l'argument `sep = ""` (aucun caractère comme séparation entre les chaînes de caractères à concaténer).

Fonction nchar :

La fonction `nchar` permet de compter le nombre de caractères dans des chaînes de caractères. Elle a aussi été illustrée dans les notes sur les [structures de données en R](#).

Fonctions toupper, tolower et casefold :

Les fonctions `toupper` et `tolower` transforment tous les caractères alphabétiques en majuscules et en minuscules respectivement.

```
extrait <- boxoffice$"Movie (Distributor)"[1:2]
extrait
```

```
## [1] "Star Wars: The Force Awakens (Disney)" "Avatar (Fox)"
```

```
toupper(extrait)
```

```
## [1] "STAR WARS: THE FORCE AWAKENS (DISNEY)" "AVATAR (FOX)"
```

```
tolower(extrait)
```

```
## [1] "star wars: the force awakens (disney)" "avatar (fox)"
```

La fonction `casefold`, utilisée comme exemple de fonction dans les notes sur les [concepts de base en R](#), permet de faire les deux transformations de casse, soit transformer tous les caractères alphabétiques en majuscules (avec l'argument `upper = TRUE`) ou transformer tous les caractères alphabétiques en minuscules (avec l'argument `upper = FALSE`).

```
casefold(extrait, upper = TRUE)
```

```
## [1] "STAR WARS: THE FORCE AWAKENS (DISNEY)" "AVATAR (FOX)"
```

```
casefold(extrait, upper = FALSE)
```

```
## [1] "star wars: the force awakens (disney)" "avatar (fox)"
```

Fonction strsplit :

La fonction `strsplit` sépare des chaînes de caractères en sous-chaînes de caractères en coupant lors de la rencontre d'une certaine sous-chaîne de caractères. Tentons d'utiliser cette fonction pour séparer le nom du film du nom du distributeur, qui se retrouvent tous deux dans la colonne "Movie (Distributor)" du data frame `boxoffice`. Demandons à `strsplit` de briser le contenu de cette colonne en deux parties : ce qui se trouve avant la parenthèse ouvrante et ce qui se trouve après. Étant donné que le nom du distributeur est toujours placé à la fin de la chaîne de caractères, entre parenthèses, nous réussirons ainsi à isoler les deux éléments.

```
resSplit <- strsplit(x = boxoffice$"Movie (Distributeur)", split = "(", fixed = TRUE)
resSplit[1:2]
```

```
## [[1]]
## [1] "Star Wars: The Force Awakens " "Disney)"
##
## [[2]]
## [1] "Avatar " "Fox)"
```

Par défaut, la fonction considère que la valeur donnée à l'argument `split` est une [expression régulière](#). Les expressions régulières ne seront pas vues ici. Elles constituent un moyen puissant pour effectuer des recherches dans des chaînes de caractères. L'argument `fixed = TRUE` demande à `strsplit` de ne pas considérer l'argument `split` comme une expression régulière, mais bien de traiter la chaîne de caractères telle qu'elle a été fournie.

Le résultat de la fonction `strsplit` est toujours une liste de la même longueur que le vecteur assigné à l'argument `x`. Chaque élément de la liste est un vecteur comprenant les sous-chaînes de caractères obtenues de la séparation. Ici, ces vecteurs ont le plus souvent une longueur de 2. Par exemple pour le premier film du jeu de données, Star Wars : The Force Awakens (Disney), on obtient le vecteur `["Star Wars: The Force Awakens ", "Disney)"]`. Par contre, certains titres de film comprennent un élément entre parenthèses de plus, par exemple "The Lion King (1994) (Disney)". Pour eux, la longueur du vecteur est 3.

La commande suivante permet d'extraire le dernier élément de chacun des vecteurs se trouvant dans `resSplit`. Elle utilise la fonction `sapply` vue dans le cours sur les [calculs de base en R](#).

```
distributeur <- sapply(resSplit, tail, n = 1)
distributeur[1:2]
```

```
## [1] "Disney)" "Fox)"
```

Fonction `substr` :

Afin d'avoir vraiment isolé le nom du distributeur, il ne reste plus qu'à retirer la parenthèse à la fin des chaînes de caractères dans `distributeur`. La [fonction `substr`](#) permet d'extraire une partie de chaînes de caractères en spécifiant les positions, dans les chaînes, des caractères à conserver.

```
distributeur <- substr(x = distributeur, start = 1, stop = nchar(distributeur) - 1)
distributeur[1:2]
```

```
## [1] "Disney" "Fox"
```

Ici nous voulons extraire les caractères de la première position jusqu'à l'avant-dernière position. Celle-ci varie d'une observation à l'autre, c'est pourquoi nous utilisons l'expression `nchar(distributeur) - 1` pour la calculer.

Fonctions `sub` et `gsub` :

Voyons maintenant comment il est possible de remplacer une sous-chaîne de caractères par une autre. Dans les données `boxoffice`, les recettes des films comprennent un signe de dollar au début, ainsi que des virgules pour séparer les milliers. À cause de ces caractères, cette colonne ne peut pas être transformée sous un format numérique directement. Nous allons donc retirer ces caractères en les remplaçant par des chaînes de caractères vides. Ensuite, nous pourrions convertir le vecteur au format numérique plutôt que caractère.

```
recettes <- boxoffice$Gross
str(recettes)
```

```
## chr [1:1000] "$936,662,225" "$760,507,625" "$700,059,566" ...
```

```
recettes <- sub(pattern = "$", replacement = "", x = recettes, fixed = TRUE)
str(recettes)

## chr [1:1000] "936,662,225" "760,507,625" "700,059,566" "678,815,482" ...

recettes <- gsub(pattern = ",", replacement = "", x = recettes, fixed = TRUE)
str(recettes)

## chr [1:1000] "936662225" "760507625" "700059566" "678815482" ...

recettes <- as.numeric(recettes)
str(recettes)

## num [1:1000] 9.37e+08 7.61e+08 7.00e+08 6.79e+08 6.59e+08 ...
```

Il est maintenant possible de faire des calculs numériques sur les recettes des films, puisqu'elles sont stockées sous un format numérique.

Les fonctions `sub` et `gsub` servent donc à chercher les occurrences d'un « motif » (argument `pattern`) dans des chaînes de caractères (argument `x`), puis à remplacer ces occurrences par un autre motif (argument `replacement`). La fonction `sub` remplace seulement la première occurrence de `pattern`, alors que la fonction `gsub` remplace toutes les occurrences. Tout comme la fonction `strsplit`, les fonctions `sub` et `gsub` travaillent par défaut avec des motifs exprimés sous forme d'expressions régulières. L'argument `fixed = TRUE` empêche ce comportement et implique le traitement des motifs comme des chaînes de caractères.

Fonctions `grep` et `grepl` :

Supposons que nous désirions avoir une variable logique prenant la valeur `TRUE` si la compagnie Disney a distribué le film, `FALSE` sinon. Nous pourrions utiliser la fonction `grepl` comme suit pour obtenir cette variable.

```
Disney <- grepl(pattern = "Disney", x = distributeur, fixed = TRUE)
sum(Disney)

## [1] 161
```

Ainsi, 161 des 1000 films du jeu de données `boxoffice` ont été distribués par Disney.

Les fonctions `grep` et `grepl` identifient les éléments d'un vecteur de chaînes de caractères (argument `x`) contenant un certain « motif » (argument `pattern`). Pour ne pas que ce motif soit traité comme une expression régulière, il faut fournir la valeur `TRUE` à l'argument `fixed`. La fonction `grep` retourne un vecteur numérique contenant les positions dans `x` des éléments possédant le motif. Comme nous l'avons constaté dans l'exemple précédent, la fonction `grepl` retourne pour sa part un vecteur logique aussi long que `x`, contenant la valeur `TRUE` pour les éléments possédant le motif, `FALSE` sinon.

Les fonctions `sub`, `gsub`, `grep` et `grepl` sont toutes documentées dans la même fiche d'aide : [celle de la fonction `grep`](#). Quelques autres fonctions de manipulation de chaînes de caractères sont documentées dans cette fiche, mais ne sont pas présentées ici.

Fonction `chartr` :

Lorsque nous avons besoin de remplacer une série de caractères par d'autres (chaque caractère de la série ayant une valeur de remplacement spécifique), il est possible de procéder en appelant la fonction `gsub` séparément pour chaque caractère de la série. Il existe cependant une façon de faire plus rapide : utiliser la fonction `chartr`. Il faut fournir à cette fonction une chaîne de caractères comprenant tous les caractères à traduire (argument `old`). Il faut aussi lui fournir une chaîne de caractères comprenant tous les caractères de remplacement (argument `new`), en respectant l'ordre des caractères dans le premier vecteur. Finalement, il faut fournir à `chartr` le vecteur des chaînes de caractères dans lesquelles effectuer les remplacements.

Par exemple, la commande suivante permet de retirer les accents français d'un vecteur de chaînes de caractères.

Une fois s'être assuré d'avoir un paramètre "LC_TIME" en anglais, nous pouvons appeler la fonction `as.Date` comme suit.

```
boxoffice$Date <- as.Date(boxoffice$"Release Date", format = "%b %d, %Y")
str(boxoffice[, c("Release Date", "Date")])

## 'data.frame': 1000 obs. of 2 variables:
## $ Release Date: chr "Dec 18, 2015" "Dec 18, 2009" "Feb 16, 2018" "Apr 27, 2018" ...
## $ Date : Date, format: "2015-12-18" "2009-12-18" ...

head(boxoffice[, -c(1,4)])

##           Movie (Distributor) Release Date      Date
## 1 Star Wars: The Force Awakens (Disney) Dec 18, 2015 2015-12-18
## 2           Avatar (Fox) Dec 18, 2009 2009-12-18
## 3           Black Panther (Disney) Feb 16, 2018 2018-02-16
## 4 Avengers: Infinity War (Disney) Apr 27, 2018 2018-04-27
## 5           Titanic (Paramount) Dec 19, 1997 1997-12-19
## 6           Jurassic World (Universal) Jun 12, 2015 2015-06-12
```

Fonctions `difftime` et `Sys.Date`

Pour calculer le nombre d'années entre aujourd'hui et la date de sortie des films, tentons d'utiliser la fonction `difftime`. D'abord, obtenons la date courante avec la fonction `Sys.Date`.

```
aujourd'hui <- Sys.Date()
aujourd'hui
```

```
## [1] "2019-01-25"
```

Ensuite, demandons à `difftime` de calculer la différence, en années, entre aujourd'hui et les dates de sortie des films.

```
boxoffice$Age <- difftime(aujourd'hui, boxoffice$Date, units = "years")
```

```
## Error in match.arg(units) :
## 'arg' should be one of "auto", "secs", "mins", "hours", "days", "weeks"
```

La fonction `difftime` ne calcule pas de différence entre des dates en années, d'où l'erreur obtenue. Elle peut cependant calculer cette différence en jours ou en semaines (aussi en secondes, minutes ou heures si les dates comprennent aussi une heure). Pour faire un calcul approximatif, mais presque exact, du nombre d'années écoulées entre aujourd'hui et la sortie des films, nous pourrions d'abord calculer le temps écoulé en jours. Ensuite, nous pourrions diviser ce nombre de jours par 365.25 jours, soit le nombre moyen de jours dans une année.

```
boxoffice$Age <- as.numeric(difftime(aujourd'hui, boxoffice$Date, units = "days")) / 365.25
head(boxoffice[, -c(1,4)])
```

```
##           Movie (Distributor) Release Date      Date      Age
## 1 Star Wars: The Force Awakens (Disney) Dec 18, 2015 2015-12-18 3.1047228
## 2           Avatar (Fox) Dec 18, 2009 2009-12-18 9.1033539
## 3           Black Panther (Disney) Feb 16, 2018 2018-02-16 0.9390828
## 4 Avengers: Infinity War (Disney) Apr 27, 2018 2018-04-27 0.7474333
## 5           Titanic (Paramount) Dec 19, 1997 1997-12-19 21.1006160
## 6           Jurassic World (Universal) Jun 12, 2015 2015-06-12 3.6221766
```

Cet âge est presque exact. Cependant, il y a une petite erreur due au fait que nous ne savons pas combien il y a eu d'années bissextiles entre aujourd'hui et la date de sortie des films. Une année ne comporte pas 365.25 jours, mais bien 365 ou 366 jours.

Fonction format

Une date peut se présenter sous plusieurs formats, explicités dans la [fiche d'aide nommée `strptime`](#).

```
help(strptime)
```

Grâce à ces différents formats, nous pourrions par exemple extraire le mois de sortie d'un film grâce à la [fonction `format`](#) comme suit.

```
boxoffice$Month <- as.integer(format(boxoffice$Date, "%m"))
```

Voyons ce que ça donne.

```
head(boxoffice[, -c(1,4)])
```

##	Movie (Distributor)	Release Date	Date	Age	Month
## 1	Star Wars: The Force Awakens (Disney)	Dec 18, 2015	2015-12-18	3.1047228	12
## 2	Avatar (Fox)	Dec 18, 2009	2009-12-18	9.1033539	12
## 3	Black Panther (Disney)	Feb 16, 2018	2018-02-16	0.9390828	2
## 4	Avengers: Infinity War (Disney)	Apr 27, 2018	2018-04-27	0.7474333	4
## 5	Titanic (Paramount)	Dec 19, 1997	1997-12-19	21.1006160	12
## 6	Jurassic World (Universal)	Jun 12, 2015	2015-06-12	3.6221766	6

Note : Il existe d'autres formats de date en R permettant d'inclure l'heure : `"POSIXlt"` et `"POSIXct"`, mais nous n'approfondirons pas ces formats. Seule une petite partie des possibilités de manipulation de dates en R est abordée ici.

Manipulation de jeux de données

Nous venons de voir différentes façons de manipuler des variables dans un jeu de données sous forme de data frame. Voyons maintenant différentes manipulations usuelles de jeux de données entiers, soit :

- la fusion de plusieurs jeux de données,
- la modification de l'ordre des observations ou des variables dans un jeu de données,
- la modification de la mise en forme de jeux de données.

Pour illustrer des manipulations, reprenons le data frame `dataEx` créé dans le cours [Lecture et écriture dans des fichiers externes à partir de R](#) afin d'illustrer quelques mises en forme de jeux de données.

```
de1 <- c(2,3,4,1,2,3,5,6,5,4)
de2 <- c(1,4,2,3,5,4,6,2,5,3)
lanceur <- rep(c("Luc", "Kim"), each = 5)
dataEx <- data.frame(de1, de2, lanceur)
# Introduction de valeurs manquantes
dataEx$de2[7] <- NA
dataEx$lanceur[3] <- NA
# Affichage du data frame
dataEx
```

##	de1	de2	lanceur
## 1	2	1	Luc
## 2	3	4	Luc
## 3	4	2	<NA>
## 4	1	3	Luc
## 5	2	5	Luc

```
## 6    3    4    Kim
## 7    5   NA    Kim
## 8    6    2    Kim
## 9    5    5    Kim
## 10   4    3    Kim
```

Fusion de données

Fusionner des données signifie ici de mettre en commun deux jeux de données ou plus en un seul jeu de données. Une fusion peut être une simple concaténation, c'est-à-dire une mise bout à bout de lignes ou de colonnes. Elle peut aussi être réalisée par association en tenant compte des observations dans des variables en commun dans les jeux de données à fusionner.

Concaténation d'observations (lignes)

Supposons que nous avons en main deux jeux de données contenant différentes observations pour les mêmes variables. Par exemple, en plus du data frame `dataEx`, disons que nous avons les observations supplémentaires suivantes.

```
supp <- data.frame(de1 = c(2,5), de2 = c(5, 6), lanceur = c("Luc", "Kim"))
supp
```

```
##   de1 de2 lanceur
## 1    2    5     Luc
## 2    5    6     Kim
```

Pour concaténer les jeux de données `dataEx` et `supp` en un seul jeu de données, nous pourrions utiliser la fonction `rbind` comme suit :

```
rbind(dataEx, supp)
```

```
##   de1 de2 lanceur
## 1    2    1     Luc
## 2    3    4     Luc
## 3    4    2  <NA>
## 4    1    3     Luc
## 5    2    5     Luc
## 6    3    4     Kim
## 7    5   NA     Kim
## 8    6    2     Kim
## 9    5    5     Kim
## 10   4    3     Kim
## 11   2    5     Luc
## 12   5    6     Kim
```

Il serait maintenant judicieux d'assigner cette concaténation à un nom afin de conserver le résultat obtenu dans un objet. Si nous assignions cet appel à `rbind` au nom `dataEx`, nous nous retrouverions à ajouter des observations (soit des lignes) à `dataEx`. Nous pourrions aussi créer un nouvel objet pour stocker le résultat de la concaténation.

Attention : Lors d'une concaténation d'observations, il faut manuellement s'assurer que les colonnes des data frames concaténés correspondent (mêmes variables, dans le même ordre).

Concaténation de variables (colonnes)

Lorsque nous avons ajouté des variables à un jeu de données dans la section précédente, nous avons réalisé une forme de concaténation de jeux de données en colonnes. Voici une copie de l'exemple réalisé

```
cars2 <- cars # pour repartir avec la version originale du jeu de données
cars2 <- data.frame(cars2, dist_m = cars2$dist * 0.3048)
```

Dans cette forme de concaténation, le deuxième jeu de données fusionné est constitué d'une seule variable, stockée dans un vecteur, et le résultat de la concaténation remplace le premier jeu de données fusionné.

La concaténation de variables peut être plus générale que ça. Nous pourrions concaténer deux jeux de données contenant chacun plus d'une variable et stocker le résultat de la concaténation dans un nouvel objet.

Par exemple, disons que nous avons un jeu de données contenant des informations sur les lanceurs de dés du jeu de données `dataEx`.

```
dataExLanceurs <- data.frame(lateralite = rep(c("gaucher", "droitier"), each = 5),
                             age = rep(c(22, 51), each = 5))
dataExLanceurs
```

```
##      lateralite age
## 1      gaucher  22
## 2      gaucher  22
## 3      gaucher  22
## 4      gaucher  22
## 5      gaucher  22
## 6     droitier  51
## 7     droitier  51
## 8     droitier  51
## 9     droitier  51
## 10    droitier  51
```

Pour concaténer en colonnes les jeux de données `dataEx` et `dataExLanceurs`, nous pourrions utiliser la fonction `cbind` ou `data.frame` comme suit.

```
cbind(dataEx, dataExLanceurs)
```

```
data.frame(dataEx, dataExLanceurs)
```

```
##      de1 de2 lanceur lateralite age
## 1      2  1      Luc    gaucher  22
## 2      3  4      Luc    gaucher  22
## 3      4  2    <NA>    gaucher  22
## 4      1  3      Luc    gaucher  22
## 5      2  5      Luc    gaucher  22
## 6      3  4      Kim    droitier  51
## 7      5 NA      Kim    droitier  51
## 8      6  2      Kim    droitier  51
## 9      5  5      Kim    droitier  51
## 10     4  3      Kim    droitier  51
```

Le résultat de la concaténation pourrait être assigné à un nom d'objet R existant ou à un nouveau nom, créant du coup un nouvel objet.

Fusion par association - la fonction `merge`

Une concaténation de variables crée un jeu de données utile seulement si les lignes dans les jeux de données concaténés sont associées aux mêmes individus, dans le même ordre. Par exemple, le résultat que nous venons d'obtenir nous porte à croire que Luc est gaucher et qu'il a 22 ans, et que Kim est droitier et à 51 ans. Mais rien dans le jeu de données `dataExLanceurs` ne nous disait quelles caractéristiques correspondaient à quels lanceurs. Et la ligne 3 de ce jeu de données est surprenante. Le nom du lanceur est manquant, mais la latéralité et l'âge du lanceur sont connus.

Avec une fusion par concaténation de variables, les risques d'erreurs sont grands. Il est plus prudent de fusionner des jeux de données en s'assurant de spécifier à quel individu (ou unité statistique) réfère chaque ligne, et ce, pour chaque jeu de données. Une fusion par association peut s'effectuer en R avec la [fonction merge](#).

Fusionner des jeux de données par association signifie de mettre en commun des observations de certaines variables en associant correctement les observations selon l'individu auquel elles réfèrent.

Dans l'exemple précédent, nous aurions préféré fusionner le jeu de données `dataEx` au jeu de données suivant.

```
lanceurs <- data.frame(lanceur = c("Luc", "Kim"),
                      lateralite = c("gaucher", "droitier"),
                      age = c(22, 51))

lanceurs
```

```
##   lanceur lateralite age
## 1     Luc   gaucher  22
## 2     Kim  droitier  51
```

Ce jeu de données indique clairement à quel lanceur sont associées les différentes caractéristiques.

Un fusion par association selon la valeur de la variable `lanceur` entre les data frames `dataEx` et `lanceurs` permettrait d'ajouter des colonnes pour la latéralité et l'âge du lanceur à `dataEx` de façon plus intelligente et moins risquée que précédemment.

```
merge(dataEx, lanceurs, by = "lanceur", all = TRUE, sort = FALSE)
```

```
##   lanceur de1 de2 lateralite age
## 1     Luc  2  1   gaucher  22
## 2     Luc  3  4   gaucher  22
## 3     Luc  1  3   gaucher  22
## 4     Luc  2  5   gaucher  22
## 5     Kim  3  4  droitier  51
## 6     Kim  5 NA  droitier  51
## 7     Kim  6  2  droitier  51
## 8     Kim  5  5  droitier  51
## 9     Kim  4  3  droitier  51
## 10    <NA>  4  2    <NA>  NA
```

Ici, les deux jeux de données fusionnés contenaient une seule colonne de même nom. La fonction `merge` aurait donc pu identifier par défaut cette colonne comme étant celle à considérer dans l'association et l'argument `by` n'aurait pas été nécessaire. Si la ou les colonnes identifiants les individus n'avaient pas porté les mêmes noms dans les deux jeux de données, nous aurions dû spécifier les arguments `by.x` et `by.y`.

L'argument `sort` permet de spécifier à `merge` si les observations doivent être ordonnées dans la sortie (par défaut elles le sont) selon les valeurs des variables spécifiées dans `by` (ou `by.x` et `by.y`). Dans l'exemple, malgré l'emploi de `sort = FALSE`, nous constatons que l'observation avec un nom de lanceur manquant a été placée à la fin. Cependant, les observations avec un nom de lanceur non manquant n'ont bel et bien pas été ordonnées.

L'argument `all` permet quant à lui de spécifier à `merge` quelles observations doivent être conservées dans la sortie. Par défaut, seules les observations présentes dans les deux jeux de données, ou ne contenant pas de valeurs manquantes pour les variables spécifiées dans `by`, sont conservées. Selon la [terminologie des bases de données relationnelles](#), la fonction `merge` effectue par défaut une jointure interne (en anglais *inner join*). Dans l'exemple, nous avons plutôt spécifié `all = TRUE` afin de conserver toutes les observations des deux jeux de données. Nous avons donc effectué une jointure externe complète (en anglais *full outer join*). Avec les arguments `all.x` et `all.y`, nous aurions aussi pu effectuer des jointure externe gauche ou droite (en anglais *left or right outer join*).

Modification de l'ordre des données

Il existe différentes fonctions en R pour ordonner les éléments d'un objet.

Fonction `rev` :

Une des fonctions les plus simples pour effectuer un réordonnement d'éléments est la [fonction `rev`](#), qui renverse l'ordre des éléments d'un vecteur (ou un facteur).

```
dataEx$de1

## [1] 2 3 4 1 2 3 5 6 5 4
rev(dataEx$de1)

## [1] 4 5 6 5 3 2 1 4 3 2
```

Fonction `sort` :

Pour effectuer le réordonnement des éléments d'un vecteur (ou un facteur) selon les valeurs des données en éléments, de façon à replacer les éléments en ordre croissant ou décroissant de valeur, il faut plutôt utiliser la [fonction `sort`](#). Par exemple, ordonnons des valeurs numériques en ordre décroissant.

```
dataEx$de1

## [1] 2 3 4 1 2 3 5 6 5 4
sort(dataEx$de1, decreasing = TRUE)

## [1] 6 5 5 4 4 3 3 2 2 1
```

Par défaut, l'ordre croissant est retourné. Alors pour obtenir un ordre décroissant, il faut spécifier `decreasing = TRUE`.

Nous avons vu dans les notes sur les [calculs de base en R](#) comment connaître l'ordre utilisé par notre session R pour classer des caractères. Il suffit de fournir à la fonction `sort` un vecteur contenant tous les caractères possibles, par exemple comme suit (des caractères sont potentiellement manquants ici).

```
caracteres_speciaux <-
  c("!", "\"", "#", "$", "%", "&", "'", "(", ")", "*", "+", ",", "-", ".", "/", ":", ";",
    "<", "=", ">", "?", "@", "[", "\\\"", "]", "^", "_", "{", "|", "}", "~")
lettres_accentuees <- c("â", "ã", "ä", "å", "æ", "ç", "è", "é", "ê", "ë", "î", "ï", "ô", "ù", "ü", "û", "ü", "ç")
catacteres_ordonnes <- sort(c(caracteres_speciaux, 0:9, letters, LETTERS,
                             lettres_accentuees, toupper(lettres_accentuees)))
paste(catacteres_ordonnes, collapse = "")
```

Dans cette commande, l'appel à la fonction `paste` sert uniquement à raccourcir la sortie en mettant bout à bout les caractères ordonnés dans une seule chaîne de caractères. J'obtiens le résultat suivant, qui sera peut-être différent sur votre ordinateur si vous n'avez pas les mêmes paramètres régionaux que moi.

```
"' - ! \" $ % & ( ) * , . / : ; ? @ [ \\ \" ] ^ _ { | } ~
0 1 2 3 4 5 6 7 8 9 a A à Â â ã B b C c Ç ç D d E e É é Ê ê Ë ë F f G g H h I i Î î Ï ï J j K k L l M m
n N o O ô Ô p P q Q r R s S t T u U û Ü ü V v W w X x Y y Z z "
```

Ordonnons en ordre alphabétique croissant les noms des lanceurs dans le jeu de données `dataEx`.

```
dataEx$lanceur

## [1] Luc  Luc  <NA> Luc  Luc  Kim  Kim  Kim  Kim  Kim
## Levels: Kim Luc
```

```
sort(dataEx$lanceur)
```

```
## [1] Kim Kim Kim Kim Kim Luc Luc Luc Luc  
## Levels: Kim Luc
```

Remarquons que la valeur manquante est omise dans la sortie. C'est le comportement par défaut de la fonction `sort`. Pour forcer `sort` à inclure les valeurs manquantes dans la sortie, il faut lui dire où les placer avec l'argument `na.last` comme suit.

```
sort(dataEx$lanceur, na.last = TRUE)
```

```
## [1] Kim Kim Kim Kim Kim Luc Luc Luc Luc <NA>  
## Levels: Kim Luc
```

Fonction `order` :

Pour modifier l'ordre des observations ou des variables dans un jeu de données R, la fonction `order` est utile. Elle permet d'obtenir un vecteur d'indices des observations dans l'ordre désiré.

Par exemple, pour ordonner les observations du jeu de données `dataEx` en ordre croissant de la valeur de la variable `de1`,

```
dataEx$de1
```

```
## [1] 2 3 4 1 2 3 5 6 5 4
```

nous devons d'abord obtenir le vecteur d'indices suivant avec la fonction `order` :

```
permutation <- order(dataEx$de1)  
permutation
```

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

Ce vecteur s'interprète comme suit : la plus petite valeur de `dataEx$de1` se trouve en position 4 (valeur 1), les deuxièmes plus petites (valeur 2 qui revient 2 fois) se trouvent dans les positions 1 et 5, ..., la plus grande valeur (6) se trouve en position 8. Ensuite, il suffit de fournir ce vecteur à un l'opérateur d'indexage `[` pour réordonner les lignes de `dataEx`.

```
dataEx[permutation, ]
```

```
##      de1 de2 lanceur  
## 4      1   3      Luc  
## 1      2   1      Luc  
## 5      2   5      Luc  
## 2      3   4      Luc  
## 6      3   4      Kim  
## 3      4   2      <NA>  
## 10     4   3      Kim  
## 7      5  NA      Kim  
## 9      5   5      Kim  
## 8      6   2      Kim
```

Notons que nous aurions pu jumeler l'appel à `order` à l'appel à `[` comme suit :

```
dataEx[order(dataEx$de1), ]
```

La fonction `order` accepte plus d'un vecteur de valeurs pour déterminer le nouvel ordre. Par exemple, la commande suivante permet d'ordonner les observations de `dataEx` en ordre croissant de valeur de la variable `de1`, en résolvant les égalités en `de1` en considérant l'ordre alphabétique croissant de la variable `lanceur`.

```
dataEx[order(dataEx$de1, dataEx$lanceur), ]
```

```
##      de1 de2 lanceur
## 4      1  3      Luc
## 1      2  1      Luc
## 5      2  5      Luc
## 6      3  4      Kim
## 2      3  4      Luc
## 10     4  3      Kim
## 3      4  2      <NA>
## 7      5 NA      Kim
## 9      5  5      Kim
## 8      6  2      Kim
```

La fonction `order` a un argument `decreasing`, tout comme `sort`, pour demander un ordre décroissant plutôt que croissant. Par exemple, pour modifier l'ordre des colonnes de `dataEx` selon l'ordre alphabétique décroissant des noms de colonnes, nous pourrions procéder comme suit.

```
dataEx[, order(names(dataEx), decreasing = TRUE)]
```

```
##      lanceur de2 de1
## 1      Luc   1  2
## 2      Luc   4  3
## 3      <NA>  2  4
## 4      Luc   3  1
## 5      Luc   5  2
## 6      Kim   4  3
## 7      Kim  NA  5
## 8      Kim   2  6
## 9      Kim   5  5
## 10     Kim   3  4
```

Ainsi, la fonction `order` fournit la permutation selon l'ordre voulue, mais c'est l'opérateur `[` qui effectue l'opération de réordonner des lignes ou des colonnes. Une permutation fournie comme premier argument à `[` (avant la virgule) réordonne les lignes et une permutation fournie comme deuxième argument à `[` (après la virgule) réordonne les colonnes. Les lignes et les colonnes peuvent être réordonner simultanément.

Nous pourrions fournir à l'opérateur `[` des permutations arbitraires. Par exemple, pour déplacer en une commande la dernière observation du data frame `dataEx` en première position et la variable `lanceur` en premier position nous pouvons procéder ainsi.

```
dataEx[c(nrow(dataEx), 1:(nrow(dataEx) - 1)), c("lanceur", "de1", "de2")]
```

```
##      lanceur de1 de2
## 10     Kim   4  3
## 1      Luc   2  1
## 2      Luc   3  4
## 3      <NA>  4  2
## 4      Luc   1  3
## 5      Luc   2  5
## 6      Kim   3  4
## 7      Kim   5 NA
## 8      Kim   6  2
## 9      Kim   5  5
```

Notons finalement que ce genre de manipulation fonctionne aussi avec une matrice plutôt qu'un data frame.

Fonction `arrange` du package `dplyr`

Si l'utilisation de la fonction `order` vous paraît peu conviviale, vous pouvez vous tourner vers la [fonction `arrange` du package `dplyr`](#) pour le réordonnement d'observations dans un data frame. Voici un exemple d'utilisation de cette fonction, produisant presque le même résultat que la commande suivante.

```
dataEx[order(dataEx$de1, dataEx$lanceur), ]
```

```
##    de1 de2 lanceur
## 4    1  3      Luc
## 1    2  1      Luc
## 5    2  5      Luc
## 6    3  4      Kim
## 2    3  4      Luc
## 10   4  3      Kim
## 3    4  2      <NA>
## 7    5 NA      Kim
## 9    5  5      Kim
## 8    6  2      Kim
```

```
library(dplyr)
arrange(dataEx, de1, lanceur)
```

```
##    de1 de2 lanceur
## 1    1  3      Luc
## 2    2  1      Luc
## 3    2  5      Luc
## 4    3  4      Kim
## 5    3  4      Luc
## 6    4  3      Kim
## 7    4  2      <NA>
## 8    5 NA      Kim
## 9    5  5      Kim
## 10   6  2      Kim
```

La seule différence entre les deux sorties est les noms des lignes. La fonction `arrange` renomme les lignes 1 au nombre total de lignes.

Avec `arrange`, pour demander un ordre décroissant par rapport à une certaine variable, il faut envelopper le nom de la variable par un appel à la [fonction `desc`](#) comme dans cet exemple.

```
arrange(dataEx, desc(de1), lanceur)
```

```
##    de1 de2 lanceur
## 1    6  2      Kim
## 2    5 NA      Kim
## 3    5  5      Kim
## 4    4  3      Kim
## 5    4  2      <NA>
## 6    3  4      Kim
## 7    3  4      Luc
## 8    2  1      Luc
## 9    2  5      Luc
## 10   1  3      Luc
```

La fonction `arrange` s'avère donc plus souple que `order`, qui applique le même type d'ordre (ascendant ou descendant) à toutes les variables.

```
dataEx[order(dataEx$de1, dataEx$lanceur, decreasing = TRUE), ]
```

```
##      de1 de2 lanceur
## 8      6  2      Kim
## 7      5 NA      Kim
## 9      5  5      Kim
## 10     4  3      Kim
## 3      4  2     <NA>
## 2      3  4      Luc
## 6      3  4      Kim
## 1      2  1      Luc
## 5      2  5      Luc
## 4      1  3      Luc
```

Cependant, la fonction `arrange` permet seulement de réordonner les lignes d'un data frame (ou d'un `tibble`), alors que la fonction `order` utilisée avec l'opérateur `[` permet en fait de réordonner les éléments de n'importe quel type de structure de données et selon n'importe laquelle de ses dimensions.

Changement de mise en forme de jeux de données

La mise en forme de jeux de données est définie ici, de façon générale, par le choix de positionnement des données dans un tableau de données.

Par exemple, en statistique, les variables sont habituellement positionnées en colonnes et les observations en lignes dans un tableau de données. Que faire avec un tableau de données dans lequel les variables sont en lignes et les observations en colonnes ? Il serait judicieux de transposer le tableau (inverser les lignes et les colonnes), afin de retomber sur une mise en forme plus standard.

Mais par mise en forme de jeux de données en statistique, on réfère le plus souvent à la distinction entre deux types de mise en forme, que nous appellerons ici les mises en forme « large » et « longue ». Ces mises en forme sont définies un peu plus bas.

Transposition avec la fonction `t`

La fonction `t` sert à transposer (inverser les lignes et les colonnes) une matrice ou un data frame.

```
dataEx
```

```
##      de1 de2 lanceur
## 1      2  1      Luc
## 2      3  4      Luc
## 3      4  2     <NA>
## 4      1  3      Luc
## 5      2  5      Luc
## 6      3  4      Kim
## 7      5 NA      Kim
## 8      6  2      Kim
## 9      5  5      Kim
## 10     4  3      Kim
```

```
dataEx_transpo <- t(dataEx)
dataEx_transpo
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## de1    "2"  "3"  "4"  "1"  "2"  "3"  "5"  "6"  "5"  "4"
## de2    "1"  "4"  "2"  "3"  "5"  "4" NA   "2"  "5"  "3"
## lanceur "Luc" "Luc" NA   "Luc" "Luc" "Kim" "Kim" "Kim" "Kim" "Kim"
```

```
str(dataEx_transpo)
```

```
## chr [1:3, 1:10] "2" " 1" "Luc" "3" " 4" "Luc" "4" " 2" NA "1" " 3" ...  
## - attr(*, "dimnames")=List of 2  
## ..$ : chr [1:3] "de1" "de2" "lanceur"  
## ..$ : NULL
```

Cependant, la fonction `t` est en fait conçue pour effectuer des calculs d'algèbre linéaire et elle retourne toujours une matrice.

Comparaison des mises en forme large et longue

Tout d'abord, introduisons un peu de terminologie informelle qui nous aidera à décrire les mises en forme.

- Individus (ou unités statistiques ou sujets) :
 - unités sur lesquelles des mesures sont prises ;
 - possèdent des identifiants.
- Variables :
 - ce qui est mesuré ;
 - possèdent aussi des identifiants.
- Valeurs :
 - mesures obtenues ;
 - chaque valeur est associée à un individu et une variable.

Mise en forme large (en anglais *wide*) :

Lorsqu'un jeu de données comprend une seule ligne par individu et que les différentes variables mesurées se trouvent dans différentes colonnes, la mise en forme du jeu de données est dite « large ».

Exemple de notes d'étudiants - Mise en forme large

- Individus = étudiants (au nombre de 4),
- Variables = notes pour deux travaux pratiques : TP1 et TP2

```
notes_large
```

```
##   etudiant TP1 TP2  
## 1      a  90  95  
## 2      b  80  74  
## 3      c  70  89  
## 4      d  60  85
```

Mise en forme longue (en anglais *long*) :

Si un jeu de données comporte plutôt plus d'une ligne par individu, sa mise en forme est dite « longue ».

Exemple de notes d'étudiants - Mise en forme long

```
notes_long
```

```
##   etudiant note evaluation  
## 1      a    90         TP1  
## 2      b    80         TP1  
## 3      c    70         TP1  
## 4      d    60         TP1  
## 5      a    95         TP2
```

```
## 6      b      74      TP2
## 7      c      89      TP2
## 8      d      85      TP2
```

Le data frame `notes_long` contient exactement les mêmes données que celles dans le data frame `notes_large`, mais disposées différemment. Dans `notes_long`, il y a deux lignes par individu. Le tableau de mise en forme longue contient plus de lignes, donc est plus long, que le tableau de mise en forme large, d'où les qualificatifs « large » et « longue » pour les mises en forme.

Dans un jeu de données avec une mise en forme longue, les mesures pour plus d'une variable sont combinées dans une même colonne. Cependant, il doit aussi nécessairement y avoir une ou des colonnes servant à identifier à quelle variable est associée chaque ligne (colonne `evaluation` dans l'exemple).

Lors d'une analyse statistique de données stockées sous une mise en forme longue, l'analyste doit être vigilant. Il est important d'être conscient que certaines lignes concernent les mêmes individus, donc qu'il s'agit de *mesures répétées* sur les mêmes individus et non d'observations indépendantes.

Mise en forme intermédiaire :

Il peut arriver qu'un jeu de données ait une mise en forme mi-large, mi-longue. Voici un exemple pour illustrer cette mise en forme intermédiaire.

Exemple de mesures sur des plants

- Individus = plants (au nombre de 3),
- Variables = Hauteur_Temps1, Hauteur_Temps2, Diametre_Temps1, Diametre_Temps2

Format large pur : une ligne par individu, une colonne par variable

Dimension de la sous-matrice contenant les valeurs : 3 x 4

```
plants_large
```

```
##   Plant Hauteur_Temps1 Hauteur_Temps2 Diametre_Temps1 Diametre_Temps2
## 1     a              67              69              15              15
## 2     b              59              65              16              17
## 3     c              62              66              13              14
```

Format intermédiaire : plus d'une ligne par individu, mais valeurs réparties dans plus d'une colonne

Dimension de la sous-matrice contenant les valeurs : 6 x 2

```
plants_inter
```

```
##   Plant Temps Hauteur Diametre
## 1     a     1     67         15
## 2     b     1     59         16
## 3     c     1     62         13
## 4     a     2     69         15
## 5     b     2     65         17
## 6     c     2     66         14
```

Format long pur : toutes les valeurs sont empilées dans une seule colonne

Dimension de la sous-matrice contenant les valeurs : 12 x 1

```
plants_long
```



```
##      Plant Temps Variable Valeur
## 1      a      1  Hauteur    67
## 2      b      1  Hauteur    59
## 3      c      1  Hauteur    62
## 4      a      2  Hauteur    69
## 5      b      2  Hauteur    65
## 6      c      2  Hauteur    66
## 7      a      1 Diametre    15
## 8      b      1 Diametre    16
## 9      c      1 Diametre    13
## 10     a      2 Diametre    15
## 11     b      2 Diametre    17
## 12     c      2 Diametre    14
```

ou encore

```
plants_long2
```

```
##      Plant      Variable Valeur
## 1      a Hauteur_Temps1    67
## 2      b Hauteur_Temps1    59
## 3      c Hauteur_Temps1    62
## 4      a Hauteur_Temps2    69
## 5      b Hauteur_Temps2    65
## 6      c Hauteur_Temps2    66
## 7      a Diametre_Temps1    15
## 8      b Diametre_Temps1    16
## 9      c Diametre_Temps1    13
## 10     a Diametre_Temps2    15
## 11     b Diametre_Temps2    17
## 12     c Diametre_Temps2    14
```

Utilisation des deux mises en forme

Certaines fonctions R requièrent un type particulier de mise en forme du jeu de données.

Dans l'exemple des notes d'étudiants, nous pourrions vouloir comparer les notes des deux travaux. Étant donné que nous sommes en présence de données prises sur les mêmes individus (qualifiées en statistique de *mesures répétées* ou encore de *données paires* dans le cas particulier de deux variables), un test approprié à réaliser serait un test de comparaison de moyennes pour données paires. La [fonction `t.test`](#) réalise ce type de test. Elle accepte les jeux de données de mise en forme large,

```
t.test(notes_large$TP1, notes_large$TP2, paired = TRUE)
```

```
##
## Paired t-test
##
## data: notes_large$TP1 and notes_large$TP2
## t = -1.54, df = 3, p-value = 0.2212
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
## -32.96547 11.46547
## sample estimates:
## mean of the differences
## -10.75
```

ou de mise en forme longue (qui permet l'utilisation d'une formule R).

```
t.test(note ~ evaluation, data = notes_long, paired = TRUE)
```

```
##
## Paired t-test
##
## data: note by evaluation
## t = -1.54, df = 3, p-value = 0.2212
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
## -32.96547 11.46547
## sample estimates:
## mean of the differences
## -10.75
```

Nous pourrions aussi ajuster un modèle linéaire mixte pour réaliser un test équivalent. La [fonction lme](#) du [package nlme](#) ajuste des modèles linéaires mixtes. Elle accepte cependant seulement des jeux de données de mise en forme longue.

```
library(nlme)
mixed <- lme(fixed = note ~ evaluation, data = notes_long, random = ~ 1 | etudiant)
summary(mixed)
```

```
## Linear mixed-effects model fit by REML
## Data: notes_long
##      AIC      BIC    logLik
## 56.51876 55.6858 -24.25938
##
## Random effects:
## Formula: ~1 | etudiant
##      (Intercept) Residual
## StdDev:      5.000001 9.872098
##
## Fixed effects: note ~ evaluation
##              Value Std.Error DF   t-value p-value
## (Intercept)  75.00  5.533045  3 13.554924  0.0009
## evaluationTP2 10.75  6.980628  3  1.539976  0.2212
## Correlation:
##              (Intr)
## evaluationTP2 -0.631
##
## Standardized Within-Group Residuals:
##      Min      Q1      Med      Q3      Max
## -1.2489496 -0.6258934  0.2768885  0.5459932  1.1029740
##
## Number of Observations: 8
## Number of Groups: 4
```

Nous avons utilisé ici un modèle statistique puissant pour réaliser en fait un test simple. Le test sur le terme `evaluationTP2` dans le modèle linéaire mixte revient au test de comparaison de moyennes pour données paires réalisé avec la fonction `t.test`. Nous pouvons constater que le seuil observé des deux tests est le même, soit 0.2212.

Le but de cet exemple était de montrer qu'il est utile en R de savoir convertir un jeu de données d'une mise à forme à l'autre puisque certaines fonctions exigent une mise en forme spécifique.

Transformation entre les mises en forme large et longue avec les fonctions `stack` et `unstack`

Revenons maintenant au jeu de données `dataEx`. Celui-ci est sous une mise en forme large, car il contient une seule ligne par individu. Dans ce jeu de données, un individu est un lancer de dés.

```
dataEx
```

```
##      de1 de2 lanceur
## 1      2   1      Luc
## 2      3   4      Luc
## 3      4   2    <NA>
## 4      1   3      Luc
## 5      2   5      Luc
## 6      3   4      Kim
## 7      5  NA      Kim
## 8      6   2      Kim
## 9      5   5      Kim
## 10     4   3      Kim
```

Il est possible de faire passer ce jeu de données à une mise en forme longue avec la [fonction `stack`](#) comme suit

```
dataEx_long <- stack(dataEx)
```

```
## Warning in stack.data.frame(dataEx): non-vector columns will be ignored
```

```
dataEx_long
```

```
##      values ind
## 1         2 de1
## 2         3 de1
## 3         4 de1
## 4         1 de1
## 5         2 de1
## 6         3 de1
## 7         5 de1
## 8         6 de1
## 9         5 de1
## 10        4 de1
## 11        1 de2
## 12        4 de2
## 13        2 de2
## 14        3 de2
## 15        5 de2
## 16        4 de2
## 17       NA de2
## 18        2 de2
## 19        5 de2
## 20        3 de2
```

Cette fonction n'est pas très puissante. Elle n'a pas su traiter la colonne `lanceur` et l'a laissé tomber.

Nous pourrions réaliser la transformation inverse avec la [fonction `unstack`](#), pour revenir à la mise en forme originale.

```
dataEx_large <- unstack(dataEx_long)
```

```
dataEx_large
```

```
##      de1 de2
## 1      2   1
```

```
## 2    3    4
## 3    4    2
## 4    1    3
## 5    2    5
## 6    3    4
## 7    5   NA
## 8    6    2
## 9    5    5
## 10   4    3
```

Voyons maintenant une autre fonction qui nous permettra de réaliser les mêmes transformations, sans mettre de côté la variable `lanceur`.

Transformation entre les mises en forme large et longue avec la fonction `reshape`

Le package `stats`, chargé en R par défaut, comprend une fonction nommée `reshape` dont l'utilité est de faire passer un jeu de donnée d'une mise en forme large vers une mise en forme longue, et aussi de réaliser la transformation inverse, d'une mise en forme longue vers une mise en forme large. Cette fonction est puissante, mais un peu difficile à utiliser, il faut l'avouer. Il est facile de se perdre dans ses nombreux arguments.

Exemple de transformation vers une mise en forme longue :

Pour faire passer le data frame `dataEx` vers une mise en forme longue, nous pouvons procéder comme suit.

```
dataEx_long2 <- reshape(data = dataEx,
                        direction = "long",
                        varying = c("de1", "de2"),
                        v.names = "resultat")
dataEx_long2
```

```
##      lanceur time resultat id
## 1.1      Luc   1         2   1
## 2.1      Luc   1         3   2
## 3.1     <NA>   1         4   3
## 4.1      Luc   1         1   4
## 5.1      Luc   1         2   5
## 6.1      Kim   1         3   6
## 7.1      Kim   1         5   7
## 8.1      Kim   1         6   8
## 9.1      Kim   1         5   9
## 10.1     Kim   1         4  10
## 1.2      Luc   2         1   1
## 2.2      Luc   2         4   2
## 3.2     <NA>   2         2   3
## 4.2      Luc   2         3   4
## 5.2      Luc   2         5   5
## 6.2      Kim   2         4   6
## 7.2      Kim   2        NA   7
## 8.2      Kim   2         2   8
## 9.2      Kim   2         5   9
## 10.2     Kim   2         3  10
```

Il faut toujours fournir à la fonction `reshape` les arguments :

- `data` : le data frame dont à mise en forme est à transformer ;
- `direction` : la direction de la transformation, `long` pour une transformation vers une mise en forme longue, `wide` pour une transformation vers une mise en forme large.

Lors d'une transformation vers une mise en forme longue, il faut aussi spécifier l'argument :

- **varying** : les noms ou les indices des colonnes dans le jeu de données d'origine sous la mise en forme large (**data**) contenant les données à combiner dans des colonnes uniques sous la mise en forme longue.

La fonction **reshape** va alors tenter de deviner le ou les noms à donner à cette ou ces nouvelles colonnes, mais il est préférable de lui spécifier ce ou ces noms avec l'argument :

- **v.names** : nom(s) de la ou des colonnes dans la mise en forme longue correspondant à plusieurs colonnes sous la mise en forme large.

Par défaut, **reshape** a nommé **time** la colonne contenant les identifiants des variables empilées. Aussi, **reshape** a utilisé comme identifiants dans cette colonne les entiers allant de 1 jusqu'au nombre de variables empilées. Il est possible de contrôler cette colonne avec les arguments :

- **timevar** : nom de la colonne dans la mise en forme longue contenant l'identifiant de la variable à laquelle correspond la ou les données qui se trouvent dans la ou les colonnes identifiées par **v.names**.
- **times** : les valeurs d'identifiants à utiliser dans la colonne spécifiée par **timevar**.

Les noms de ces arguments contiennent le mot **time** en référence à un cas classique de données pour lesquelles nous sommes susceptibles de vouloir modifier la mise en forme : des mesures répétées dans le temps.

```
dataEx_long2 <- reshape(data = dataEx,
                        direction = "long",
                        varying = c("de1", "de2"),
                        v.names = "resultat",
                        timevar = "de",
                        times = c("de1", "de2"))
```

dataEx_long2

```
##      lanceur  de resultat id
## 1.de1      Luc de1        2  1
## 2.de1      Luc de1        3  2
## 3.de1    <NA> de1        4  3
## 4.de1      Luc de1        1  4
## 5.de1      Luc de1        2  5
## 6.de1      Kim de1        3  6
## 7.de1      Kim de1        5  7
## 8.de1      Kim de1        6  8
## 9.de1      Kim de1        5  9
## 10.de1     Kim de1        4 10
## 1.de2      Luc de2        1  1
## 2.de2      Luc de2        4  2
## 3.de2    <NA> de2        2  3
## 4.de2      Luc de2        3  4
## 5.de2      Luc de2        5  5
## 6.de2      Kim de2        4  6
## 7.de2      Kim de2       NA  7
## 8.de2      Kim de2        2  8
## 9.de2      Kim de2        5  9
## 10.de2     Kim de2        3 10
```

Sous une mise en forme longue, **reshape** inclut aussi toujours au moins une colonne pour contenir un identifiant des individus. Il nomme par défaut cette colonne **id** et utilise comme identifiants les entiers allant de 1 jusqu'au nombre de lignes dans le jeu de données d'origine sous la mise en forme large (**data**). Pour contrôler cette colonne, il faut utiliser les arguments :

- **idvar** : nom(s) de la ou des colonnes pour identifier les individus sous la mise en forme longue.
- **ids** : les valeurs d'identifiants à utiliser dans la ou les colonnes spécifiées par **idvar**.

```
dataEx_long2 <- reshape(data = dataEx,
                        direction = "long",
                        varying = c("de1", "de2"),
                        v.names = "resultat",
                        timevar = "de",
                        times = c("de1", "de2"),
                        idvar = "noLancer",
                        ids = paste0("l", 1:10))

dataEx_long2
```

```
##      lanceur  de resultat noLancer
## 11.de1      Luc de1        2       11
## 12.de1      Luc de1        3       12
## 13.de1      <NA> de1        4       13
## 14.de1      Luc de1        1       14
## 15.de1      Luc de1        2       15
## 16.de1      Kim de1        3       16
## 17.de1      Kim de1        5       17
## 18.de1      Kim de1        6       18
## 19.de1      Kim de1        5       19
## 110.de1     Kim de1        4      110
## 11.de2      Luc de2        1       11
## 12.de2      Luc de2        4       12
## 13.de2      <NA> de2        2       13
## 14.de2      Luc de2        3       14
## 15.de2      Luc de2        5       15
## 16.de2      Kim de2        4       16
## 17.de2      Kim de2        NA       17
## 18.de2      Kim de2        2       18
## 19.de2      Kim de2        5       19
## 110.de2     Kim de2        3      110
```

Dans le jeu de données de mise en forme longue obtenu en sortie de `reshape`, les données pour les variables (colonnes) ne contenant pas de mesures répétées ont simplement été recopiées autant de fois que nécessaire. Il serait possible de laisser tomber ces variables grâce à l'argument `drop`. Il serait aussi possible de contrôler les noms des lignes du data frame obtenu avec l'argument `new.row.names`.

Exemple de transformation vers une mise en forme large :

Lorsque nous avons obtenu un jeu de données avec la fonction `reshape`, il est toujours possible de retourner à la mise en forme d'origine simplement avec la commande

```
reshape(dataEx_long2)
```

```
##      lanceur noLancer de1 de2
## 11.de1      Luc      11   2   1
## 12.de1      Luc      12   3   4
## 13.de1      <NA>     13   4   2
## 14.de1      Luc      14   1   3
## 15.de1      Luc      15   2   5
## 16.de1      Kim      16   3   4
## 17.de1      Kim      17   5  NA
## 18.de1      Kim      18   6   2
## 19.de1      Kim      19   5   5
## 110.de1     Kim     110   4   3
```

en raison d'un attribut inséré par `reshape` dans sa sortie.

```
str(dataEx_long2)

## 'data.frame':    20 obs. of  4 variables:
## $ lanceur : Factor w/ 2 levels "Kim","Luc": 2 2 NA 2 2 1 1 1 1 1 ...
## $ de      : chr  "de1" "de1" "de1" "de1" ...
## $ resultat: num  2 3 4 1 2 3 5 6 5 4 ...
## $ noLancer: chr  "11" "12" "13" "14" ...
## - attr(*, "reshapeLong")=List of 4
## ..$ varying:List of 1
## .. ..$ resultat: chr  "de1" "de2"
## .. ..- attr(*, "v.names")= chr "resultat"
## .. ..- attr(*, "times")= chr  "de1" "de2"
## ..$ v.names: chr "resultat"
## ..$ idvar : chr "noLancer"
## ..$ timevar: chr "de"
```

Mais supposons que nous n'avons pas créé `dataEx_long2` avec la fonction `reshape` et que nous voulons transformer ce jeu de données vers une mise en forme large. Afin de réellement simuler une telle situation, nous allons effacer de `dataEx_long2` son attribut nommé `reshapeLong` comme suit.

```
attr(dataEx_long2, "reshapeLong") <- NULL
str(dataEx_long2)

## 'data.frame':    20 obs. of  4 variables:
## $ lanceur : Factor w/ 2 levels "Kim","Luc": 2 2 NA 2 2 1 1 1 1 1 ...
## $ de      : chr  "de1" "de1" "de1" "de1" ...
## $ resultat: num  2 3 4 1 2 3 5 6 5 4 ...
## $ noLancer: chr  "11" "12" "13" "14" ...
```

Nous pourrions procéder ainsi pour effectuer la transformation de mise en forme.

```
dataEx_large2 <- reshape(data = dataEx_long2,
                        direction = "wide",
                        timevar = "de",
                        idvar = "noLancer")

dataEx_large2

##           noLancer lanceur.de1 resultat.de1 lanceur.de2 resultat.de2
## 11.de1           11         Luc           2         Luc           1
## 12.de1           12         Luc           3         Luc           4
## 13.de1           13        <NA>           4        <NA>           2
## 14.de1           14         Luc           1         Luc           3
## 15.de1           15         Luc           2         Luc           5
## 16.de1           16         Kim           3         Kim           4
## 17.de1           17         Kim           5         Kim          NA
## 18.de1           18         Kim           6         Kim           2
## 19.de1           19         Kim           5         Kim           5
## 110.de1          110         Kim           4         Kim           3
```

Les seuls arguments obligatoires pour réaliser cette transformation vers une mise en forme large sont `data`, `direction`, `timevar` et `idvar`, dont les descriptions ont été fournies ci-dessus. Cependant, `reshape` a par défaut considéré que toutes les variables autres que celles identifiées avec `timevar` et `idvar` contiennent des mesures répétées. Nous nous retrouvons donc maintenant avec deux colonnes identiques contenant le nom du lanceur. Pour spécifier à `reshape` la ou les colonnes pour lesquelles les données doivent être réparties en plusieurs colonnes, il faut aussi spécifier l'argument `v.names`.

```
dataEx_large2 <- reshape(dataEx_long2,
  direction = "wide",
  timevar = "de",
  idvar = "noLancer",
  v.names = "resultat")
dataEx_large2
```

```
##      lanceur noLancer resultat.de1 resultat.de2
## 11.de1      Luc      11           2           1
## 12.de1      Luc      12           3           4
## 13.de1    <NA>      13           4           2
## 14.de1      Luc      14           1           3
## 15.de1      Luc      15           2           5
## 16.de1      Kim      16           3           4
## 17.de1      Kim      17           5          NA
## 18.de1      Kim      18           6           2
## 19.de1      Kim      19           5           5
## 110.de1     Kim     110           4           3
```

Et voilà !

Afin de reconstruire un data frame vraiment identique à `dataEx`,

```
dataEx
```

```
##    de1 de2 lanceur
## 1    2   1      Luc
## 2    3   4      Luc
## 3    4   2    <NA>
## 4    1   3      Luc
## 5    2   5      Luc
## 6    3   4      Kim
## 7    5  NA      Kim
## 8    6   2      Kim
## 9    5   5      Kim
## 10   4   3      Kim
```

il suffirait de retirer la colonne `noLancer`, réordonner les colonnes, renommer les colonnes contenant les résultats des lancers de dés et renommer les lignes.

```
dataEx_large2$noLancer <- NULL # retirer une colonne
dataEx_large2 <- dataEx_large2[, c(2,3,1)] # réordonner les colonnes
colnames(dataEx_large2)[1:2] <- c("de1", "de2") # renommer certaines colonnes
rownames(dataEx_large2) <- 1:10 # renommer les lignes
dataEx_large2
```

```
##    de1 de2 lanceur
## 1    2   1      Luc
## 2    3   4      Luc
## 3    4   2    <NA>
## 4    1   3      Luc
## 5    2   5      Luc
## 6    3   4      Kim
## 7    5  NA      Kim
## 8    6   2      Kim
## 9    5   5      Kim
## 10   4   3      Kim
```


Transformation entre les mises en forme large et longue avec les fonctions `gather` et `spread` du package `tidyr`

Des alternatives à la fonction `reshape` ont été créées et sont distribuées dans des packages R. Notamment, le package `tidyr` offre les fonctions suivantes, permettant d'effectuer des transformations de mise en forme.

Transformation	Package <code>stats</code>	Package <code>tidyr</code>
large vers long	<code>reshape</code> (avec <code>direction = "long"</code>)	<code>gather</code>
long vers large	<code>reshape</code> (avec <code>direction = "wide"</code>)	<code>spread</code>

Illustrons l'utilisation des fonctions `gather` et `spread` de `tidyr`.

```
library(tidyr)
```

Exemple de transformation vers une mise en forme longue :

Voici une commande pour transformer le data frame `dataEx` de sa mise en forme d'origine, large, à la mise en forme longue.

```
dataEx_long3 <- gather(data = dataEx, key = de, value = resultat, de1, de2)
dataEx_long3
```

```
##   lanceur  de resultat
## 1      Luc de1        2
## 2      Luc de1        3
## 3    <NA> de1        4
## 4      Luc de1        1
## 5      Luc de1        2
## 6      Kim de1        3
## 7      Kim de1        5
## 8      Kim de1        6
## 9      Kim de1        5
## 10     Kim de1        4
## 11     Luc de2        1
## 12     Luc de2        4
## 13    <NA> de2        2
## 14     Luc de2        3
## 15     Luc de2        5
## 16     Kim de2        4
## 17     Kim de2       NA
## 18     Kim de2        2
## 19     Kim de2        5
## 20     Kim de2        3
```

Il faut fournir à `gather` les arguments :

- **data** : le data frame à transformer de mise en forme.
- **key** : le nom de la colonne dans la mise en forme longue pour contenir l'identifiant de la variable à laquelle correspond la donnée qui se retrouvera dans la colonne identifiée par **value**.
- **value** : le nom de la colonne dans la mise en forme longue correspondant à plusieurs colonnes sous la mise en forme large.

Notons que les noms fournis aux arguments **key** et **value** peuvent être encadrés de guillemets (donc sous forme de chaîne de caractères) ou non.

Après avoir fourni les trois premiers arguments nommés obligatoires, il faut fournir les noms (sans les encadrer

de guillemets) des colonnes dans le jeu de données d'origine sous la mise en forme large (**data**) contenant les données à combiner dans des colonnes uniques sous la mise en forme longue.

Remarquons que la fonction **gather** n'ajoute pas une colonne pour identifier les individus comme le fait la fonction **reshape**. Il aurait fallu que le jeu de données d'origine contienne déjà une colonne avec ces identifiants. Dans **dataEx**, ce sont plutôt les noms de lignes qui identifient en quelque sorte les individus.

Exemple de transformation vers une mise en forme large :

Afin d'effectuer une transformation d'une mise en forme longue vers une mise en forme large avec la fonction **spread** de **tidyr**, le jeu de données fourni en entrée à la fonction doit contenir une colonne pour identifier les individus. Voici donc un exemple d'utilisation de la fonction **spread**, réalisé en utilisant comme jeu de données d'origine **dataEx_long2**.

```
dataEx_large3 <- spread(data = dataEx_long2, key = de, value = resultat)
dataEx_large3
```

```
##   lanceur noLancer de1 de2
## 1      Kim      110   4   3
## 2      Kim       16   3   4
## 3      Kim       17   5  NA
## 4      Kim       18   6   2
## 5      Kim       19   5   5
## 6      Luc       11   2   1
## 7      Luc       12   3   4
## 8      Luc       14   1   3
## 9      Luc       15   2   5
## 10     <NA>       13   4   2
```

La fonction a seulement besoin des arguments **data**, **key**, **value**, qui ont la même signification que pour la fonction **gather**.

Exemples supplémentaires de changement de mise en forme

Format large pur vers format intermédiaire

```
# Avec reshape
reshape(data = plants_large,
        direction = "long",
        idvar = "Plant",
        timevar = "Temps",
        times = 1:2,
        v.names = c("Hauteur", "Diametre"),
        varying = c("Diametre_Temps1", "Hauteur_Temps1", "Diametre_Temps2", "Hauteur_Temps2"))
```

```
##   Plant Temps Hauteur Diametre
## a.1    a     1      67      15
## b.1    b     1      59      16
## c.1    c     1      62      13
## a.2    a     2      69      15
## b.2    b     2      65      17
## c.2    c     2      66      14
```

L'ordre à donner aux noms de colonnes dans **varying** pour arriver au bon résultat est contre-intuitif.

```
# Avec gather
# Il faut réaliser la transformation en deux temps,
# parce que l'argument value accepte seulement un nom.
```

```
gather(data = plants_large,
       key = Time,
       value = Hauteur,
       Hauteur_Temps1, Hauteur_Temps2)
```

```
##   Plant Diametre_Temps1 Diametre_Temps2      Time Hauteur
## 1    a                15                15 Hauteur_Temps1    67
## 2    b                16                17 Hauteur_Temps1    59
## 3    c                13                14 Hauteur_Temps1    62
## 4    a                15                15 Hauteur_Temps2    69
## 5    b                16                17 Hauteur_Temps2    65
## 6    c                13                14 Hauteur_Temps2    66
```

```
gather(data = plants_large,
       key = Time,
       value = Diametre,
       Diametre_Temps1, Diametre_Temps2)
```

```
##   Plant Hauteur_Temps1 Hauteur_Temps2      Time Diametre
## 1    a                67                69 Diametre_Temps1    15
## 2    b                59                65 Diametre_Temps1    16
## 3    c                62                66 Diametre_Temps1    13
## 4    a                67                69 Diametre_Temps2    15
## 5    b                59                65 Diametre_Temps2    17
## 6    c                62                66 Diametre_Temps2    14
```

Il resterait à fusionner les deux data frames et à éliminer des variables.

Format large pur vers format long pur

```
# Avec reshape
reshape(data = plants_large,
       direction = "long",
       idvar = "Plant",
       timevar = "Variable",
       times = c("Hauteur_Temps1", "Hauteur_Temps2", "Diametre_Temps1", "Diametre_Temps2"),
       v.names = "Valeur",
       varying = c("Hauteur_Temps1", "Hauteur_Temps2", "Diametre_Temps1", "Diametre_Temps2"))
```

```
##           Plant      Variable Valeur
## a.Hauteur_Temps1    a Hauteur_Temps1    67
## b.Hauteur_Temps1    b Hauteur_Temps1    59
## c.Hauteur_Temps1    c Hauteur_Temps1    62
## a.Hauteur_Temps2    a Hauteur_Temps2    69
## b.Hauteur_Temps2    b Hauteur_Temps2    65
## c.Hauteur_Temps2    c Hauteur_Temps2    66
## a.Diametre_Temps1    a Diametre_Temps1    15
## b.Diametre_Temps1    b Diametre_Temps1    16
## c.Diametre_Temps1    c Diametre_Temps1    13
## a.Diametre_Temps2    a Diametre_Temps2    15
## b.Diametre_Temps2    b Diametre_Temps2    17
## c.Diametre_Temps2    c Diametre_Temps2    14
```

L'ordre à donner aux noms de colonnes dans `varying` pour arriver au bon résultat est contre-intuitif.

```
# Avec gather
gather(data = plants_large,
       key = Variable,
       value = Valeur,
       Hauteur_Temps1, Hauteur_Temps2, Diametre_Temps1, Diametre_Temps2)
```

```
##      Plant      Variable Valeur
## 1      a Hauteur_Temps1    67
## 2      b Hauteur_Temps1    59
## 3      c Hauteur_Temps1    62
## 4      a Hauteur_Temps2    69
## 5      b Hauteur_Temps2    65
## 6      c Hauteur_Temps2    66
## 7      a Diametre_Temps1    15
## 8      b Diametre_Temps1    16
## 9      c Diametre_Temps1    13
## 10     a Diametre_Temps2    15
## 11     b Diametre_Temps2    17
## 12     c Diametre_Temps2    14
```

Format intermédiaire vers format long pur

```
# Avec reshape
reshape(data = plants_inter,
        direction = "long",
        idvar = c("Plant", "Temps"), # n'accepte pas "Plant" seul, car la colonne
        timevar = "Variable",       # contient des combinaisons répétées
        times = c("Hauteur", "Diametre"),
        v.names = "Valeur",
        varying = c("Hauteur", "Diametre"))
```

```
##      Plant Temps Variable Valeur
## a.1.Hauteur      a      1 Hauteur    67
## b.1.Hauteur      b      1 Hauteur    59
## c.1.Hauteur      c      1 Hauteur    62
## a.2.Hauteur      a      2 Hauteur    69
## b.2.Hauteur      b      2 Hauteur    65
## c.2.Hauteur      c      2 Hauteur    66
## a.1.Diametre     a      1 Diametre    15
## b.1.Diametre     b      1 Diametre    16
## c.1.Diametre     c      1 Diametre    13
## a.2.Diametre     a      2 Diametre    15
## b.2.Diametre     b      2 Diametre    17
## c.2.Diametre     c      2 Diametre    14
```

```
# Avec gather
gather(data = plants_inter,
       key = Variable,
       value = Valeur,
       Hauteur, Diametre)
```

```
##      Plant Temps Variable Valeur
## 1      a      1 Hauteur    67
## 2      b      1 Hauteur    59
## 3      c      1 Hauteur    62
```

```
## 4      a      2 Hauteur      69
## 5      b      2 Hauteur      65
## 6      c      2 Hauteur      66
## 7      a      1 Diametre     15
## 8      b      1 Diametre     16
## 9      c      1 Diametre     13
## 10     a      2 Diametre     15
## 11     b      2 Diametre     17
## 12     c      2 Diametre     14
```

Format long pur vers format intermédiaire

Il faut partir de `plants_long`.

```
# Avec reshape
reshape(data = plants_long,
        direction = "wide",
        idvar = c("Plant", "Temps"),
        timevar = "Variable",
        v.names = "Valeur")
```

```
##   Plant Temps Valeur.Hauteur Valeur.Diametre
## 1      a      1          67          15
## 2      b      1          59          16
## 3      c      1          62          13
## 4      a      2          69          15
## 5      b      2          65          17
## 6      c      2          66          14
```

```
# Avec spread
spread(data = plants_long,
        key = Variable,
        value = Valeur)
```

```
##   Plant Temps Diametre Hauteur
## 1      a      1       15       67
## 2      a      2       15       69
## 3      b      1       16       59
## 4      b      2       17       65
## 5      c      1       13       62
## 6      c      2       14       66
```

Format long pur vers large pur

Il faut partir de `plants_long2`.

```
# Avec reshape
reshape(data = plants_long2,
        direction = "wide",
        idvar = "Plant",
        timevar = "Variable",
        v.names = "Valeur")
```

```
##   Plant Valeur.Hauteur_Temps1 Valeur.Hauteur_Temps2 Valeur.Diametre_Temps1
## 1      a          67          69          15
## 2      b          59          65          16
## 3      c          62          66          13
```

```
##   Valeur.Diametre_Temps2
## 1             15
## 2             17
## 3             14
```

```
# Avec spread
spread(data = plants_long2,
       key = Variable,
       value = Valeur)
```

```
##   Plant Diametre_Temps1 Diametre_Temps2 Hauteur_Temps1 Hauteur_Temps2
## 1     a             15             15             67             69
## 2     b             16             17             59             65
## 3     c             13             14             62             66
```

Format intermédiaire pur vers large pur

```
# Avec reshape
reshape(data = plants_inter,
       direction = "wide",
       idvar = "Plant",
       timevar = "Temps",
       v.names = c("Hauteur", "Diametre"))
```

```
##   Plant Hauteur.1 Diametre.1 Hauteur.2 Diametre.2
## 1     a         67         15         69         15
## 2     b         59         16         65         17
## 3     c         62         13         66         14
```

```
# Avec spread
# Il faut réaliser la transformation en deux temps,
# parce que l'argument value accepte seulement un nom.
spread(data = subset(plants_inter, select = - Diametre),
       key = Temps,
       value = Hauteur)
```

```
##   Plant  1  2
## 1     a 67 69
## 2     b 59 65
## 3     c 62 66
```

```
spread(data = subset(plants_inter, select = - Hauteur),
       key = Temps,
       value = Diametre)
```

```
##   Plant  1  2
## 1     a 15 15
## 2     b 16 17
## 3     c 13 14
```

```
# Il faut retirer la variable non traitée, sinon elle est considérée comme participant à
# l'identification des individus. Il resterait à renommer les colonnes et à fusionner les
# deux data frames
```

Synthèse

Transformation de variables

- Ajout ou modification de variables dans un data frame :
 - assignation à une colonne, nouvelle pour un ajout,
 - concaténation de colonnes avec `cbind` ou `data.frame`,
 - fonction `transform`;
- Variable conditionnelle à une autre : `ifelse`;
- Catégorisation d'une variable numérique : `cut`, `ave`;
- Normalisation de valeurs : `scale`;
- Manipulation de chaînes de caractères :
 - `paste` : concatène des chaînes de caractères,
 - `nchar` : calcule le nombre de caractères,
 - `toupper` : transforme toutes les lettres en majuscules,
 - `tolower` : transforme toutes les lettres en minuscules,
 - `strsplit` : brise des chaînes de caractères, en coupant lors de la rencontre d'une certaine sous-chaîne de caractères,
 - `substr` : extrait les caractères entre deux positions,
 - `sub` ou `gsub` : remplacent la première occurrence ou toutes les occurrences d'une certaine sous-chaîne de caractères par une autre,
 - `grep` et `grepl` : testent la présence d'une certaine sous-chaîne de caractères dans des chaînes de caractères,
 - `chartr` : remplace des caractères par d'autres,
 - `iconv` : convertit l'encodage de chaîne de caractères;
- Manipulation de dates :
 - `as.Date` : convertit une chaîne de caractère en format date,
 - `Sys.setlocale` : permet de modifier les paramètres régionaux d'une session R (utile lorsque nous voulons convertir en format `Date` des chaînes de caractères contenant des noms de mois dans une langue autre que celle du système d'exploitation de notre ordinateur),
 - `difftime` : calcule l'intervalle de temps entre deux dates,
 - `Sys.Date` : retourne la date courante,
 - `format` : permet de modifier le format d'une date.

Manipulation de jeux de données

Fusion de données

- Concaténer des vecteurs, matrices ou data frames en lignes (lignes = observations dans un data frame) : `rbind`.
- Concaténer des vecteurs, matrices ou data frames en colonnes (lignes = variables dans un data frame) : `cbind`, `data.frame`.
- Fusionner par association (ajout de colonnes en identifiant les lignes correspondantes) : `merge`
 - argument `by`, ou arguments `by.x` et `by.y`, pour spécifier les variables identifiant les unités (ou individus) à faire correspondre
 - argument `all`, ou arguments `all.x` et `all.y`, pour spécifier les lignes à conserver.

Modification de l'ordre des données

- **rev** : renverse l'ordre des éléments ;
- **sort** : ordonne les éléments d'un vecteur atomique
 - argument **decreasing** = TRUE pour ordre décroissant ;
- **order** : retourne la permutation des données requise pour réordonner les données selon l'ordre spécifié
→ pour effectuer le réordonnement il faut fournir le résultat retourné par **order** en argument à l'opérateur d'extraction [
 - argument **decreasing** = TRUE pour ordre décroissant ;
- **arrange** du package **dplyr** : réordonne les observations dans un data frame
 - encadrer les noms de variable d'un appel à la fonction **desc** pour obtenir un ordre décroissant.

Changement de mise en forme de jeux de données

- **t** : transpose des matrices
- **Mise en forme « large »** :
 - une seule ligne par individu (lorsque la largeur est maximale),
 - les vecteurs d'observations de certaines variables sont placés dans différentes colonnes.
- **Mise en forme « longue »** :
 - plus d'une ligne par individu,
 - les vecteurs d'observations de certaines variables sont empilés dans une même colonne, les uns en dessous des autres,
 - comprend une ou des colonnes ayant pour but d'identifier à quelle variable est associée chaque observation.

Transformation	Package utils	Package stats	Package tidyr
large vers long	stack	reshape (direction = "long")	gather
long vers large	unstack	reshape (direction = "wide")	spread

Arguments de la fonction **reshape**

- En lien avec le contexte :
 - **data** : data frame à remettre en forme ;
 - **direction** : direction de la remise en forme.
- En lien avec les individus :
 - **idvar** : nom(s) colonne(s) contenant les identifiants des individus ;
 - **ids** : identifiants des individus.
- En lien avec les variables :
 - **timevar** : nom colonne contenant les identifiants des variables ;
 - **times** : identifiants des variables.
- En lien avec les valeurs :
 - **v.names** : nom(s) colonne(s) contenant les valeurs dans le format long ou intermédiaire ;
 - **varying** : noms colonnes contenant les valeurs dans le format large ou intermédiaire.

Les noms de colonnes doivent être fournis sous forme de chaînes de caractères.

Arguments des fonctions **gather** et **spread** du package **tidyr**

gather : transformation vers un format plus long (rassembler valeurs dans une même colonne)

spread : transformation vers un format plus large (étalement des valeurs dans plusieurs colonnes)

- En lien avec le contexte :
 - **data** : data frame à remettre en forme ;

- En lien avec les individus :
 - aucun argument : colonne(s) contenant les identifiants des individus déduites de façon automatique = colonnes non identifiées dans les autres arguments.
- En lien avec les variables :
 - **key** : nom colonne contenant les identifiants des variables ; de vouloir modifier la mise en forme : des mesures répétées dans le temps.
- En lien avec les valeurs :
 - **value** : nom colonne contenant les valeurs dans le format long.
 - ... : noms colonnes contenant les valeurs dans le format large ;

Les noms de colonnes peuvent être fournis directement (ou sous forme de chaînes de caractères).

Références

- Spector, P. (2008). Data Manipulation with R. Springer, New York.

Outils alternatifs pour la manipulation de données :

- Package **dplyr** :
 - <https://dplyr.tidyverse.org/>
 - Tutoriel d'étudiants : [Manipulations de base sur un jeu de données](#)
 - alternative à **cbind**, **data.frame**, **transform** ou une assignation à une nouvelle colonne :
 - * ajouter de nouvelles variables avec **mutate** ou **transmute** ;
 - alternatives à **merge** :
 - * fusionner des jeux de données avec les fonctions **left_join**, **right_join**, **inner_join**, **full_join**, **semi_join**, **nest_join** ou **anti_join** ;
- Package **tidyr** :
 - <https://tidyr.tidyverse.org/>
 - Tutoriel d'étudiants : [Réorganisation des données par concaténation et séparation d'observations](#)

Pour aller plus loin en manipulation de chaînes de caractères :

Le package **stringr** contient des fonctions pour la manipulation de chaînes de caractères alternatives aux fonctions du R de base. Ces fonctions portent des noms plus faciles à retenir et ont des noms d'arguments unifiés.

- <https://stringr.tidyverse.org/>
- Vignette : [Introduction to stringr](#)
- Tutoriel d'étudiants : [Faciliter la manipulation de chaînes de caractères avec le package stringr](#)

Pour en savoir plus sur les expressions régulières :

- **help(regexp)**
- http://fr.wikipedia.org/wiki/Expression_rationnelle

Autres références en manipulation de chaînes de caractères en R :

- http://gastonsanchez.com/Handling_and_Processing_Strings_in_R.pdf
- <https://jozefhajnal.gitlab.io/r/r007-string-manipulation/>

Pour aller plus loin en manipulation de dates :

- documentation sur tous les format possibles : **help(strptime)**,
- package **lubridate**, qui vise à faciliter le travail avec les dates en R :
 - <https://CRAN.R-project.org/package=lubridate>
 - Vignette : [Do more with dates and times in R with lubridate 1.3.0](#)

- Tutoriel d'étudiants : [Manipuler facilement vos formats date avec lubridate](#)
- Article de blogue : [Using Dates and Times in R](#)

Pour aller plus loin en mise en forme de jeux de données :

- Fonction `arrange` du package `dplyr` ou du package `plyr` pour ordonner les observations d'un data frame :
 - <https://CRAN.R-project.org/package=dplyr>
 - Tutoriel d'étudiants : [Ordonner un data frame](#)
- Fonctions `melt` et `dcast` du package `reshape2`, qui offre une alternative à la fonction `reshape` :
 - <https://CRAN.R-project.org/package=reshape2>
 - Tutoriel d'étudiants : [Transformer un jeu de données d'une mise en forme large vers une mise en forme longue](#)
 - Tutoriel d'étudiants : [Transformer un jeu de données d'une mise en forme longue vers une mise en forme large en R](#)
- Fonctions `gather` et `spread` du package `tidyr`, qui offre une alternative à la fonction `reshape` :
 - <https://CRAN.R-project.org/package=tidyr>
 - Tutoriel d'étudiants : [Transformer un jeu de données en forme large ou longue : base R vs le package tidyr](#)

Pour manipuler des data frames R en utilisant le langage SQL :

- Package `sqldf` : <https://CRAN.R-project.org/package=sqldf>