Prétraitement de données en R

Sophie Baillargeon, Université Laval

2020-01-27

Table des matières

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. Un jeu de données brut doit, dans la grande majorité des cas, être adapté afin d'être analysable. 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. La façon de procéder avec les packages chargés par défaut lors de l'ouverture d'une session R (principalement les packages `base` et `stats`) est présentée en détails. Des façons de réaliser certaines tâches avec des packages du `tidyverse` (principalement les packages `dplyr` et `tidyr`), ainsi qu'avec le package `data.table`, sont également mentionnées, sans être approfondies. Chargeons donc ces packages.

```
library(dplyr)
library(tidyr)
library(data.table)
```

De façon générale, les packages du `tidyverse` présentent l'avantage d'être cohérents entre eux et abondamment documentés sur le web (en partie en raison de leur popularité). Pour sa part, la grande force du package `data.table` est sa capacité à effectuer rapidement des traitements, même sur des jeux de données de grande taille. Les packages du `tidyverse` tendent aussi à être un peu plus rapide que le R de base, mais pas aussi rapide que le package `data.table`. Malgré des différences en ce qui concerne les temps d'exécution, les trois options présentées ici (R de base versus `tidyverse` versus `data.table`) produisent des résultats équivalents. Ainsi, dans un cas de manipulation de données dont la taille n'est pas trop grande, le choix entre ces outils est uniquement une question de goût.

Transformations 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.

Les transformations de variables peuvent mener à des modifications 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 premières transformations présentées, nous utiliserons le jeu de données `cars` du package `dataset`.

```
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'un petit jeu de données à 50 observations prises sur des voitures et 2 variables :

- `speed` : la vitesse de déplacement des voitures tout juste avant de freiner, mesurée en miles par heure ;
- `dist` : la distance parcourue par ces voitures entre le moment d'un freinage et l'atteinte d'une immobilisation complète, mesurée en pieds.

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

Opérateurs d'indiçage et fonction `transform`

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 `cars2`, 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 la dernière assignation.

```
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 fourni comme premier argument 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 en appelant la fonction `cbind` ou `data.frame` comme suit.

```
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.

Remarque : 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. Ici, l'assignation à un nouveau nom de colonne avec l'opérateur `$` me semble la meilleure option.

Fonction `mutate` du package `dplyr`

Pour les adeptes du `tidyverse`, le package `dplyr` propose, entre autres, la fonction `mutate` pour modifier ou créer de nouvelles variables. L'opération précédente aurait pu être effectuée comme suit avec cette fonction, dont l'utilisation ressemble à celle de la fonction `transform`.

```
cars2 <- mutate(cars2, dist_m = dist * 0.3048)
```

Opérateur `:=` du package `data.table`

Si le data frame contenant les données à manipuler est d'abord transformé en data table, déjà mentionné dans les notes sur les structures de données, l'opérateur d'assignation `:=` offert par le package `data.table` peut ensuite être utilisé pour ajouter, modifier ou même effacer des variables dans les données. Cet optérateur s'utilise en conjonction avec l'opérateur, `[` comme dans l'exemple suivant.

```
cars_dt <- as.data.table(cars)
cars_dt[, dist_m := dist * 0.3048]
str(cars_dt)
```

```
## Classes 'data.table' and 'data.frame':   50 obs. of  3 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 ...
##  - attr(*, ".internal.selfref")=<externalptr>
```

Avec `:=`, aucune copie de l'objet n'est effectuée pour ensuite être réassignée. L'objet est modifié par référence (donc « sur place »), ce qui n'est pas le cas pour toutes les autres façons de faire présentées précédemment. La modification d'objets par référence est une des techniques utilisées par le package `data.table` pour optimiser ses temps de traitement.

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 dist_m.1 speed_km
## 1      4    2  0.6096   0.6096       NA
## 2      4   10  3.0480   3.0480       NA
## 3      7    4  1.2192   1.2192       NA
## 4      7   22  6.7056   6.7056       NA
```

```
## 5      8   16  4.8768   4.8768       NA
## 6      9   10  3.0480   3.0480       NA
## 7     10   18  5.4864   5.4864 16.09340
## 8     10   26  7.9248   7.9248 16.09340
## 9     10   34 10.3632  10.3632 16.09340
## 10    11   17  5.1816   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  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 ...
##  $ dist_m.1    : 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 sous-section suivante.

Notons, avant de clore cette sous-section, que le package `dplyr` offre une fonction de remplacement à `ifelse`, nommée `if_else`, tout comme le package `data.table` avec la fonction `fifelse`. Ces fonctions ont le potentiel d'être plus rapides que `ifelse`.

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 auquels réfèrent les données. Il s'agit d'une technique courante d'anonymisation de données qui contribue à permettre l'accès à des données tout en respectant des 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  7 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 ...
##  $ dist_m.1     : 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 souhaitions remplacer les identifiants des nouvelles catégories par une statistique calculée sur les valeurs observées dans chaque catégorie, nous utiliserions 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
```

Pour une observation, la valeur de la nouvelle variable `speed_catego3` est la moyenne de toutes les vitesses observées dans la même catégorie que celle de l'observation en question.

Comparaison entre la fonction `ave` et les fonctions `tapply`, `by` et `aggregate` :

La fonction `ave` calcule une statistique selon des combinaisons de niveaux de facteurs, tout comme les fonctions `tapply`, `by` et `aggregate`. Cependant, elle retourne un objet de même dimension que le premier argument qu'elle reçoit en entrée plutôt que de retourner une valeur par combinaison distincte de niveaux de facteurs. Par exemple, voici l'appel à la fonction `aggregate` correspondant à l'appel à la fonction `ave` précédent.

```r
aggregate(x = cars2$speed, by = list(catego = cars2$speed_catego), FUN = mean)
```

```
##           catego        x
## 1     20 ou plus 22.16667
## 2 entre 10 et 20 14.53125
## 3    moins de 10  6.50000
```

Ici, le résultat produit possède seulement 3 lignes, pour les trois catégories distinctes. La sortie de l'appel à la fonction `ave` précédent était plutôt de longueur 50, soit la longueur de `cars2$speed`.

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).

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

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

```r
# 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 :

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

où $\bar{x}$ est la moyenne des observations x_i et σ_x leur écart-type. Standardisons les valeurs du jeu de donnes `cars` avec la fonction `scale`.

```r
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.

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

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

```r
# 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`.

```r
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,

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

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

```r
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'indiçage `[` réfère à des éléments d'un objet. L'objet `chaine` ne contient qu'un seul élément, qui est une chaîne de caractères.

```r
str(chaine)
```

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

```r
length(chaine)
```

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

```
chaine[1]
```

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

Cette section présente quelques fonctions 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 un data frame nommé `boxoffice` créé le 25 janvier 2019 en important le contenu de la page web http://pro.boxoffice.com/numbers/all_time. Cette page web n'existe plus aujourd'hui, car le site web Boxoffice Pro a subi une refonte. Utilisons tout de même ces données, même si elles ne sont plus à jour, puisque leur format est parfait pour illustrer la manipulation de chaînes de caractères et de dates en R.

Une copie des données se trouve 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")
```

Le data frame `boxoffice` utilisé ici ressemble à celui créé dans les notes sur la lecture et l'écriture dans des fichiers externes à partir de R, mais il contient des données moins récentes et présentées sous un format un peu différent.

```
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 reprennant les noms de colonnes qui apparaissaient dans la page web d'où ses données proviennent (en remplaçant les espaces par des tirets bas et en omettant les parenthèses).

```
colnames(boxoffice) <- c("Ranking", "Movie_Distributor", "Release_Date", "Gross")
```

Il s'agit de données concernant les recettes (`Gross`) des films ayant générés le plus de revenus dans les cinémas américains.

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.

```r
extrait <- head(boxoffice$Movie_Distributor, n = 2)
extrait
```

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

```r
toupper(extrait)
```

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

```r
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`).

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

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

```r
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 série 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.

```r
resSplit <- strsplit(x = boxoffice$Movie_Distributor, split = "(", fixed = TRUE)
head(resSplit, n = 2)
```

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

Par défaut, la fonction considère que la valeur donnée a l'argument `split` est une expression régulière. Les expressions régulières ne seront pas vues ici (mais des références pour en apprendre davantage à ce sujet sont fournies dans les références). 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.

```r
distributeur <- sapply(resSplit, tail, n = 1)
head(distributeur, n = 2)
```

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

Fonction `substr` :

Afin d'avoir vraiment isolé le nom du distributeur, il ne reste plus qu'à retirer la parenthèse fermante 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.

```r
distributeur <- substr(x = distributeur, start = 1, stop = nchar(distributeur) - 1)
head(distributeur, n = 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.

Notons que nous aurions aussi pu effectuer ce traitement avec la fonction `strsplit`, en utilisant `)` comme point de coupure, ou encore avec la fonction `sub`, comme dans l'exemple suivant.

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 pourrons convertir le vecteur au format numérique plutôt que caractère.

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

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

```r
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" ...
```

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

```
##  chr [1:1000] "936662225" "760507625" "700059566" "678815482" "658672302" ...
```

```r
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.

```
chartr(
  old = "ÀÂÇÈÉÊËÎÏÔÙÛÜàâçèéêëîïôùûü",
  new = "AACEEEEIIOUUUaaceeeeiiouuu",
  x = c("François va à l'École des Pionniers.", "Où est située cette école?")
)
```

```
## [1] "Francois va a l'Ecole des Pionniers." "Ou est situee cette ecole?"
```

Fonction `iconv` :

Pour la tâche spécifique de retirer des accents, il est en fait encore plus simple d'utiliser la fonction `iconv`. Celle-ci permet de convertir l'encodage de chaîne de caractères. Un bon truc pour retirer des accents de chaînes de caractères, peu importe la langue utilisée dans ces chaînes, est de convertir vers l'encodage `"ASCII//TRANSLIT"`. Voici un exemple.

```
iconv(
  x = c("François va à l'École des Pionniers.", "Où est situé cette école?"),
  to = "ASCII//TRANSLIT"
)
```

```
## [1] "Francois va a l'Ecole des Pionniers." "Ou est situe cette ecole?"
```

La sortie de cette commande peut varier d'un système d'exploitation à l'autre à l'autre.

Package `stringr`

Un package du `tidyverse` se spécialise en manipulation de chaînes de caractères. Il s'agit du package `stringr`, qui propose des solutions de rechange à à peu près tout ce qui a été présenté dans cette sous-section. Les noms des fonctions de ce package, et de leurs arguments, ont été choisis de façon à être cohérents entre eux, ce qui n'est pas vraiment le cas pour les fonctions de manipulation de chaînes de caractères dans le R de base. En outre, quelques fonctions du package `stringr` simplifient la réalisation de tâches courantes en manipulation de chaînes de caractères. Par exemple, la fonction `str_trim` permet de retirer tout espace blanc en début et fin de chaînes de caractères. Les espaces blancs (en anglais *whitespace*) les plus courants sont les espaces, les tabulations (`\t`), et les retour à la ligne (`\n`).

```
library(stringr)
str_trim(
  string = c("\tFrançois va à l'École des Pionniers. ", "Où est située cette école?\n"),
  side = "both"
)
```

```
## [1] "François va à l'École des Pionniers."
## [2] "Où est située cette école?"
```

Des références sont fournies à la fin de ce document pour en apprendre davantage sur le package `stringr`.

Manipulation de dates

Une des variables de `boxoffice` est une date : la variable `Release_Date` contenant la date de sortie des films. Elle est pour l'instant stockée sous format caractère.

```
str(boxoffice$Release_Date)
```

```
##  chr [1:1000] "Dec 18, 2015" "Dec 18, 2009" "Feb 16, 2018" "Apr 27, 2018" ...
```

Nous pourrions vouloir compter le nombre d'années écoulées depuis la sortie des films. Pour ce faire, le plus simple est d'utiliser une classe dédiée à la manipulation de dates proposée par R.

Fonctions `as.Date` et `Sys.getlocale`

Nous allons convertir les données dans la colonne `Release_Date` vers la classe `"Date"` de R avec la fonction `as.Date`. Cependant, cette conversion fonctionnera uniquement si les paramètres régionaux (en anglais *locale*) de notre session R sont en anglais, puisque les dates comportent des abréviations de noms de mois en anglais. Ainsi, commençons par vérifier quel paramètre régional pour le temps utilise notre session R avec la fonction `Sys.getlocale`.

```
Sys.getlocale(category = "LC_TIME")
```

```
## [1] "English_Canada.1252"
```

Dans mon cas, j'utilise un système d'exploitation en anglais. En conséquence, les paramètres régionaux de mes sessions R sont par défaut en anglais. Si ce n'est pas votre cas, il faut d'abord changer le paramètre `"LC_TIME"` de votre session R pour qu'il s'agisse de l'anglais. Sous Windows, vous pouvez procéder comme suit.

```
# Sous Windows
Sys.setlocale("LC_TIME", locale = "English")
```

Les valeurs comprises comme argument `locale` dépendent du système d'exploitation. Sous un système d'exploitation Mac OS X / OS X / macOS ou Unix / Linux, tentez une des valeurs suivantes : `"en_CA"`, `"en_CA.UTF-8"` ou `"en_CA.utf8"` (ou la version US de ces valeurs, soit `"en_US"`, `"en_US.UTF-8"` ou `"en_US.utf8"`).

```
# Sous Mac OS X / OS X / macOS
Sys.setlocale("LC_TIME", locale = "en_CA.UTF-8")
```

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(subset(boxoffice, select = - c(Ranking, Gross)))
```

```
##                       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.

```
aujourdui <- Sys.Date()
aujourdui
```

```
## [1] "2020-01-27"
```

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

```
boxoffice$Age <- difftime(aujourdui, boxoffice$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$Age <- as.numeric(difftime(aujourdui, boxoffice$Date, units = "days") / 365.25)
head(subset(boxoffice, select = - c(Ranking, Gross)))
```

```
##                       Movie_Distributor Release_Date       Date       Age
## 1 Star Wars: The Force Awakens (Disney) Dec 18, 2015 2015-12-18  4.109514
## 2                           Avatar (Fox) Dec 18, 2009 2009-12-18 10.108145
## 3                Black Panther (Disney) Feb 16, 2018 2018-02-16  1.943874
## 4       Avengers: Infinity War (Disney) Apr 27, 2018 2018-04-27  1.752225
## 5                    Titanic (Paramount) Dec 19, 1997 1997-12-19 22.105407
## 6            Jurassic World (Universal) Jun 12, 2015 2015-06-12  4.626968
```

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(subset(boxoffice, select = - c(Ranking, Gross)))
```

```
##                         Movie_Distributor Release_Date       Date       Age Month
## 1 Star Wars: The Force Awakens (Disney) Dec 18, 2015 2015-12-18  4.109514    12
## 2                            Avatar (Fox) Dec 18, 2009 2009-12-18 10.108145    12
## 3                 Black Panther (Disney) Feb 16, 2018 2018-02-16  1.943874     2
## 4       Avengers: Infinity War (Disney) Apr 27, 2018 2018-04-27  1.752225     4
## 5                    Titanic (Paramount) Dec 19, 1997 1997-12-19 22.105407    12
## 6             Jurassic World (Universal) Jun 12, 2015 2015-06-12  4.626968     6
```

Note : Il existe d'autres classes dédiées à la manipulation de dates en R qui permettent d'inclure l'heure dans la date (heure, minute, seconde) : `"POSIXlt"` et `"POSIXct"`. Avec ces classes, la prise en compte des fuseaux horaires est primordiale. Nous n'approfondirons pas l'utilisation de ces classes. Seule une petite partie des possibilités de manipulation de dates en R est abordée ici.

Package `lubridate`

Le `tidyverse` offre aussi un outil pour manipuler des dates en R : le package `lubridate`. Par exemple, le calcul approximatif de temps en années entre deux dates que nous avons effectué précédemment peut être fait de façon exacte avec `lubridate` comme suit.

```
library(lubridate)
boxoffice$Date2 <- mdy(boxoffice$Release_Date, locale = "English")
boxoffice$Age2 <- time_length(interval(start = boxoffice$Date2, end = now()), "years")
head(subset(boxoffice, select = c(Movie_Distributor, Date, Date2, Age, Age2)))
```

```
##                         Movie_Distributor       Date      Date2       Age      Age2
## 1 Star Wars: The Force Awakens (Disney) 2015-12-18 2015-12-18  4.109514  4.109290
## 2                            Avatar (Fox) 2009-12-18 2009-12-18 10.108145 10.109290
## 3                 Black Panther (Disney) 2018-02-16 2018-02-16  1.943874  1.945205
## 4       Avengers: Infinity War (Disney) 2018-04-27 2018-04-27  1.752225  1.751366
## 5                    Titanic (Paramount) 1997-12-19 1997-12-19 22.105407 22.106557
## 6             Jurassic World (Universal) 2015-06-12 2015-06-12  4.626968  4.625683
```

La fonction `time_length` tient compte des années bisextiles. Elles peut aussi calculer des durées en mois tout en considérant les durées variables de ceux-ci. Remarquons que l'argument `locale` de la fonction `mdy` nous a permis de spécifier un paramète régional s'appliquant uniquement à la conversion à effectuer. La modification d'un paramètre régional global de la session R n'est pas nécessaire avec les fonctions du packages `lubridate`. Des références sont fournies à la fin de ce document pour en apprendre davantage sur le package `lubridate`.

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 sélection de sous-ensembles du jeu de données
- 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 concernant la lecture et l'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 date 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
```

Sélection de sous-ensembles de données

Sélectionner un sous-ensemble de données signifie **extraire** d'un jeu de données une partie des observations (lignes) et/ou des variables (colonnes) qu'il contient. La sélection d'observations répondant à certains critères est aussi parfois appelée **filtrage** de données. La matière couverte dans les notes sur les calculs de base en R concernant les conditions logiques nous sera très utile pour tester des critères.

Opérateur `[` et fonction `select`

Les principaux outils permettant d'extraire des sous-ensembles de données stockées dans un data frame ont déjà été présentés dans les notes sur les structures de données en R. Il s'agit de l'opérateur d'indiçage `[` et de la fonction d'extraction `subset`. La fonction `subset` a d'ailleurs été utilisée dans les derniers exemples pour sélectionner seulement certaines variables dans l'affichage des résultats.

Pour illustrer l'utilisation de ces outils dans un contexte de sélection de sous-ensembles de données, voyons comment conserver du jeu de données `dataEx` seulement les résultats de lancers de dés effectués par Kim et ne comprenant aucune donnée manquante. Dans ce sous-ensemble, la variable `lanceur` prendra toujours la même valeur. Elle ne sera donc plus vraiment pertinente et sera retirée.

Tout d'abord, voyons comment effectuer cette tâche avec l'opérateur `[`, étape par étape. Pour limiter les observations à celles pour lesquelles la variable `lanceur` prend la valeur `Kim`, nous écrivons la condition logique représentant ce critère et la fournissons au premier argument de l'opérateur `[` à plusieurs dimensions.

```
dataEx[dataEx$lanceur == "Kim", ]
```

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

Remarquons que l'observation pour laquelle la valeur de la variable `lanceur` est manquante a été conservée. Ensuite, la façon la plus simple de retirer les lignes comprenant au moins une donnée manquante est d'utiliser la fonction `na.omit` déjà mentionnée dans les notes sur les calculs de base en R.

```
na.omit(dataEx[dataEx$lanceur == "Kim", ])
```

```
##    de1 de2 lanceur
## 6    3   4     Kim
## 8    6   2     Kim
## 9    5   5     Kim
## 10   4   3     Kim
```

Finalement, la variable lanceur peut être retirée en fournissant les noms des autres variables, celles à conserver, au deuxième argument de l'opérateur `[`. Dans un cas de manipulation de data frame, l'identification des variables par leur nom plutôt que par leur position permet d'éviter des erreurs.

```
na.omit(dataEx[dataEx$lanceur == "Kim", c("de1", "de2")])
```

```
##    de1 de2
## 6    3   4
## 8    6   2
## 9    5   5
## 10   4   3
```

Effectuons maintenant le même travail avec la fonction `subset`, encore une fois étape par étape, pour bien comprendre les traitements réalisés.

```
subset(dataEx, subset = lanceur == "Kim")
```

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

Tout d'abord, remarquons que le comportement de la fonction `subset` diffère un peu de celui de l'opérateur `[`. L'observation pour laquelle la valeur de la variable `lanceur` est manquante n'a pas été conservée. Nous pourrions retirer les observations contenant des valeurs manquantes pour les autres variables en combinant des conditions logiques comme suit.

```
subset(dataEx, subset = ! is.na(de1) & ! is.na(de2) & lanceur == "Kim")
```

```
##    de1 de2 lanceur
## 6    3   4     Kim
## 8    6   2     Kim
## 9    5   5     Kim
## 10   4   3     Kim
```

Cependant, il est plus simple d'utiliser encore une fois la fonction `na.omit`. Pour retirer la varaible `lanceur`, l'argument `select` de la fonction `subset` peut être spécifié comme dans la commande suivante, qui accompli la tâche souhaitée.

```
na.omit(subset(dataEx, subset = lanceur == "Kim", select = - lanceur))
```

```
##    de1 de2
## 6    3   4
## 8    6   2
## 9    5   5
## 10   4   3
```

La possibilité de pouvoir retirer des variables en utilisant l'opéreur `-` devant leur nom permet de simplifier des commandes. Pour conserver ou retirer plus d'une variable, il faut fournir leurs noms dans un vecteur, comme nous l'avons fait précédemment dans la commande suivante (non exécutée ici).

```
head(subset(boxoffice, select = - c(Ranking, Gross)))
```

Fonctions `filter` et `select` du package `dplyr`

Le package `dplyr` attribue les tâches de sélection d'observations et de variables à deux fonctions distinctes :

- `filter` pour la sélection d'observations (lignes),
- `select` pour la sélection de variables (colonnes).

Par exemple, nous pouvons sélectionner les observations de `dataEx` pour lesquelles la valeur de la variable `lancer` est `"Kim"` avec la fonction `filter` comme suit.

```
filter(dataEx, lanceur == "Kim")
```

```
##   de1 de2 lanceur
## 1   3   4     Kim
## 2   5  NA     Kim
## 3   6   2     Kim
## 4   5   5     Kim
## 5   4   3     Kim
```

Tout comme le fait la fonction `subset`, les lignes pour lesquelles la condition vaut `NA` n'ont pas été conservées. Pour retirer les observations contenant des données manquantes, nous pouvons utiliser `na.omit`, ou encore `drop_na`, une fonction de similaire contenue dans le package `tidyr`.

```
drop_na(filter(dataEx, lanceur == "Kim"))
```

```
##   de1 de2 lanceur
## 1   3   4     Kim
## 3   6   2     Kim
## 4   5   5     Kim
## 5   4   3     Kim
```

Notons que la fonction `drop_na` permet de limiter la recherche de valeurs manquantes à un sous-ensemble de variables, ce qui n'est pas possible avec `na.omit`. La fonction `complete.cases` du packages `stats` permet cependant de faire l'équivalent (non illustré ici[1]).

Ensuite, la variable `lanceur` peut être retirée à l'aide de la fonction `select` comme suit.

[1] voir https://stackoverflow.com/questions/4862178/remove-rows-with-all-or-some-nas-missing-values-in-data-frame pour un exemple d'utilisation de complete.cases

```
select(drop_na(filter(dataEx, lanceur == "Kim")), - lanceur)
```

```
##   de1 de2
## 1   3   4
## 3   6   2
## 4   5   5
## 5   4   3
```

Cette commande est un peu difficile à lire en raison du grand nombre d'appels de fonctions imbriqués qu'elle contient. Elle serait peut-être plus claire en ajoutant des retours à la ligne et des indentations comme suit.

```
select(
  drop_na(
    filter(
      dataEx,
      lanceur == "Kim"
    )
  ),
  - lanceur
)
```

```
##   de1 de2
## 1   3   4
## 3   6   2
## 4   5   5
## 5   4   3
```

La pratique courante avec les packages du `tidyverse` est cependant d'utiliser l'opérateur « pipe » `%>%` du package `magrittr` pour rendre les commandes plus lisibles. En utilisant cet opérateur, la commande aurait l'allure suivante.

```
dataEx %>% filter(lanceur == "Kim") %>% drop_na() %>% select(- lanceur)
```

```
##   de1 de2
## 1   3   4
## 3   6   2
## 4   5   5
## 5   4   3
```

Je ne fais que mentionner cet opérateur ici. Il est recommandé mais pas obligatoire de l'employer lors de l'utilisation de packages du `tidyverse`. J'y reviendrai cependant dans un prochain cours.

Opérateur `[` et fonction `select` du package `data.table`

Pour effectuer de la sélection de sous-ensembles de données avec la package `data.table`, nous pouvons exploiter son opérateur `[` et ses arguments `i` et `j` ou encore sa méthode pour la fonction `subset` (qui est en fait une fonction générique) spécifique aux data tables. Illustrons ici l'utilisation de l'opérateur `[`. La fonction `subset` s'utilise à peu près de la même façon avec un data table qu'avec un data frame.

Après avoir transformé le data frame `dataEx` en data table,

```
dataEx_dt <- as.data.table(dataEx)
```

nous pouvons l'indicer à l'aide de l'opérateur `[` de `data.table` en fournissant une condition logique à l'argument `i` pour sélectionner des lignes.

```
dataEx_dt[i = lanceur == "Kim"]
```

```
##    de1 de2 lanceur
```

```
## 1:   3   4     Kim
## 2:   5  NA     Kim
## 3:   6   2     Kim
## 4:   5   5     Kim
## 5:   4   3     Kim
```

Dans les commandes suivantes, je vais omettre l'assignation explicite à l'argument `i` puisque ce n'est vraiment pas une pratique courante d'inclure cette assignation dans un appel à `[`. Selon l'odre des arguments de l'opérateur dans sa définition (voir la section Usage de sa fiche d'aide), la première valeur fournie après `[` est assignée à l'argument `i` et la deuxième à l'argument `j` si elles ne sont pas explicitement assignées à un nom d'argument (et si aucune valeur fournie en troisième position ou plus loin n'est assignée à `i` ou `j`, selon les règles d'assignation implicite de valeurs aux arguments présentée dans les notes sur les concepts de base en R).

Le package `data.table` offre une méthode pour la fonction `na.omit` spécifique aux data tables.

```
na.omit(dataEx_dt[lanceur == "Kim"])
```

```
##    de1 de2 lanceur
## 1:   3   4     Kim
## 2:   6   2     Kim
## 3:   5   5     Kim
## 4:   4   3     Kim
```

La sélection de variables dans un data table s'effectue en fournissant une valeur à l'argument `j`. Par exemple, nous pouvons sélectionner les variables `de1` et `de2` de `dataEx_dt` comme suit (commande non évaluée),

```
na.omit(dataEx_dt[lanceur == "Kim", .(de1, de2)])
```

ou encore omettre la variable `lanceur`, comme suit.

```
na.omit(dataEx_dt[lanceur == "Kim", ! c("lanceur")])
```

```
##    de1 de2
## 1:   3   4
## 2:   6   2
## 3:   5   5
## 4:   4   3
```

Combinaison de données

Combiner des données signifie de mettre en commun deux jeux de données ou plus. Ils peuvent être mis en commun par une simple concaténation, c'est-à-dire une mise bout à bout, de lignes ou de colonnes. Ils peuvent aussi être fusionné par association en tenant compte des valeurs prises par des variables communes aux jeux de données.

Concaténation d'observations (lignes) - Fonction `rbind`

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), lanceur = c("Luc", "Kim"), de2 = c(5, 6))
supp
```

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

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
```

Remarquons que `rbind` a su mettre dans les mêmes colonnes les observations des variables portant les mêmes noms, même si elles n'étaient pas dans le même ordre dans les deux jeux de données à combiner.

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`, le jeu de données `dataEx` d'origine serait remplacé par le résultat de la concaténation. Nous pourrions aussi créer un nouvel objet pour stocker le résultat.

Concaténation de variables (colonnes) - Fonction `cbind` ou `data.table`

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 d'un exemple réalisé.

```
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)
```

```
# ou encore
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 - Fonction `merge`

Une concaténation de variables crée un jeu de données utile seulement si les valeurs sur une même ligne réfèrent toutes à un même individu (ou unité statistique) et les valeurs sur une même colonne à une même variable. 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 droitière et a 51 ans. Mais rien dans le jeu de données `dataExLanceurs` ne dit à quels lanceurs correspondent les caractéristiques. Et la ligne 3 de ce jeu de données est surprenante. Le nom du lanceur est manquant, mais sa latéralité et son âge sont connus.

Avec une fusion par concaténation de variables, les risques d'erreurs sont présents. Il est plus prudent de fusionner des jeux de données en spécifiant par rapport à quelles variables établir les correspondances entre les lignes des deux jeux de données. Appelons ce type de combinaison de données jointure ou fusion par association. Une telle fusion peut s'effectuer en R avec la fonction `merge`.

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

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

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

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

Une fusion par association selon la valeur de la variable identifiant le lanceur entre les data frames `dataEx` et `lanceurs` peut s'effectuer comme suit.

```
merge(x = dataEx, y = lanceurs, by.x = "lanceur", by.y = "nom", 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, étant donné que la variable selon laquelle établir les correspondances ne porte pas le même nom dans les deux jeux de données, nous avons dû spécifier les arguments `by.x` et `by.y`. Ces arguments prennent comme valeur un vecteur contenant les noms des variables d'association dans le jeu de données fourni en entrée à l'argument `x` pour `by.x` et `y` pour `by.y`. Lorsque les variables d'association portent le même nom partout, seul l'argument `by` a besoin d'être spécifié. Si aucune valeur n'est fournie aux arguments `by.x` et `by.y` ou `by`, les variables utilisées pour associer les lignes sont celles portant le même nom dans les deux jeux de données.

L'argument `sort` permet d'indiquer à `merge` si les observations doivent être ordonnées dans la sortie (par défaut elles le sont) selon les valeurs des variables d'association. Si l'utilisateur demande à ce que les observations ne soient pas ordonnées, comme dans l'exemple précédent, les observations contenant des valeurs manquantes pour les variables d'association sont tout de même placées à la fin du data frame retourné en sortie.

L'argument `all` permet quant à lui de spécifier à `merge` quelles observations doivent être conservées dans la sortie. Par défaut, seules les observations ayant pu être associées à une observation de l'autre jeu de données 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 jointures externes gauches ou droites (en anglais *left or right outer join*).

Fonctions de concaténation et de jointure du package `dplyr`

Le package `dplyr` propose la fonction `bind_rows` en remplacement de `rbind` et `bind_cols` en remplacement de `cbind`. Une capacité intéressante de `bind_rows` est d'arriver à concaténer en lignes des jeux de données ne comportant pas toutes les mêmes variables. Des valeurs manquantes sont ajoutées dans les colonnes des variables non communes aux deux jeux de données.

```
supp2 <- data.frame(supp, lateralite = c("gaucher", "droitier"))
supp2
```

```
##   de1 lanceur de2 lateralite
## 1   2     Luc   5    gaucher
## 2   5     Kim   6   droitier
```

```
bind_rows(dataEx, supp2)
```

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

```
## 9    5   5     Kim       <NA>
## 10   4   3     Kim       <NA>
## 11   2   5     Luc    gaucher
## 12   5   6     Kim   droitier
```

Le fonction `bind_rows` peut aussi ajouter une colonne dans le résultat de la concaténation pour identifier la provenance des lignes grâce à son argument `.id`.

```
bind_rows(jeu1 = dataEx, jeu2 = supp2, .id = "origine")
```

```
##    origine de1 de2 lanceur lateralite
## 1     jeu1   2   1     Luc       <NA>
## 2     jeu1   3   4     Luc       <NA>
## 3     jeu1   4   2    <NA>       <NA>
## 4     jeu1   1   3     Luc       <NA>
## 5     jeu1   2   5     Luc       <NA>
## 6     jeu1   3   4     Kim       <NA>
## 7     jeu1   5  NA     Kim       <NA>
## 8     jeu1   6   2     Kim       <NA>
## 9     jeu1   5   5     Kim       <NA>
## 10    jeu1   4   3     Kim       <NA>
## 11    jeu2   2   5     Luc    gaucher
## 12    jeu2   5   6     Kim   droitier
```

De plus, au lieu d'offrir une seule fonction capable d'effectuer plusieurs types de fusions par association, le package `dplyr` comporte plusieurs fonctions de jointure effectuant chacunes un seul type de jointure, identifié par le nom de la fonction : `inner_join`, `full_join`, `left_join`, `right_join`, etc. L'exemple de fusion par association effectué avec `merge` se reproduit comme suit avec une fonction de `dplyr`.

```
full_join(x = dataEx, y = lanceurs, by = c("lanceur" = "nom"))
```

```
##    de1 de2 lanceur lateralite age
## 1    2   1     Luc    gaucher  22
## 2    3   4     Luc    gaucher  22
## 3    4   2    <NA>       <NA>  NA
## 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
```

Contrairement à `merge`, les fonctions de jointure de `dplyr` ne modifient aucunement l'ordre des variables et l'ordre des observations. Ainsi, la ou les variables d'association ne sont pas placées au début et les observations avec des valeurs d'association manquantes ne sont pas placées à la fin dans le jeu de données produit en sortie.

Fonctions `rbindlist` et `merge` du package `data.table`

Le package `data.table` contient la fonction `rbindlist`, qui est une solution de rechange à `rbind` du R de base ou `bind_rows` de `dplyr`. Elle a les mêmes capacité que cette dernière.

```
rbindlist(list(jeu1 = dataEx, jeu2 = supp2), fill = TRUE, idcol = "origine")
```

```
##     origine de1 de2 lanceur lateralite
##  1:    jeu1   2   1     Luc       <NA>
##  2:    jeu1   3   4     Luc       <NA>
##  3:    jeu1   4   2    <NA>       <NA>
```

```
##  4:    jeu1   1   3     Luc       <NA>
##  5:    jeu1   2   5     Luc       <NA>
##  6:    jeu1   3   4     Kim       <NA>
##  7:    jeu1   5  NA     Kim       <NA>
##  8:    jeu1   6   2     Kim       <NA>
##  9:    jeu1   5   5     Kim       <NA>
## 10:    jeu1   4   3     Kim       <NA>
## 11:    jeu2   2   5     Luc    gaucher
## 12:    jeu2   5   6     Kim   droitier
```

Le package `data.table` comporte aussi une méthode pour la fonction `merge` spécifique aux data tables qui s'utilise de la même façon que la méthode `merge` pour data frames (celle dont nous avons parlé précédemment).

```
lanceurs_dt <- as.data.table(lanceurs)
merge(x = dataEx_dt, y = lanceurs_dt, by.x = "lanceur", by.y = "nom",
      all = TRUE, sort = FALSE)
```

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

Il est aussi possible de fusionner des données avec l'opérateur `[` du package `data.table` et son argument `on`, mais cette fonctionnalité ne sera pas illustrée ici.

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 ordonnancement 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 ordonnancement 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. Rappeleons qu'il suffit de fournir à la fonction `sort` un vecteur contenant tous les caractères potentiels, par exemple.

```
caracteres_speciaux <-
  c("!", "\"", "#", "$", "%", "&", "'", "(", ")", "*", "+", ",", "-", ".", "/", ":", ";",
    "<", "=",">", "?", "@", "[", "\\", "]", "^", "_", "{", "|", "}", "~")
lettres_accentuees <- c("à", "â", "é", "è", "ê", "ë", "ï", "î", "ô", "ù", "ü", "û", "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.

```
"'-!\"#$%&()*,./:;?@[\\]^_{|}~+<=>0123456789aAàÀâÂbBcCçÇdDeEéÉèÈêÊëËfFgGhHiIïÏîÎjJkKlLmM
nNoOôÔpPqQrRsStTuUùÙûÛüÜvVwWxXyYzZ"
```

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'indice 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 l'opérateur d'indiçage `[` pour ordonner 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` fournie la permutation selon l'ordre voulue, mais c'est l'opérateur `[` qui effectue la tâche d'ordonner des lignes ou des colonnes. Une permutation fournie comme premier argument à `[` (avant la virgule) ordonne les lignes et une permutation fournie comme deuxième argument à `[` (après la virgule) ordonne les colonnes. Les lignes et les colonnes peuvent être ordonner 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 l'ordonnancement 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
```

```
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.

Dans l'appel à la fonction `arrange`, il suffit de fournir d'abord le data frame à traiter, puis les noms des variables selon lesquelles définir l'ordre. Par défaut, ces variables sont prises dans le data frame fourni comme premier argument. Si elles proviennent bien de celui-ci, nul besoin de précéder leur nom de `nom_du_data_frame$` ou de l'encadrer de guillemets.

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. Toutefois, pour des variables numériques, le type d'ordre peut toujours être inversé en transformant la variable par elle-même multiplitée par -1.

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

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

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

Fonctions `setorder` et `setcolorder` du package `data.table`

Les fonctions `setorder` et `setcolorder` du package `data.table` permettent d'ordonner, respectivement, les lignes et les colonnes d'un data table. Tout comme l'opérateur `:=`, ces fonctions modifient un data table par

référence (donc « sur place »), sans créer de copie.

Par exemple, reprenons la version data table du jeu de données `dataEx`.

```
dataEx_dt
```

```
##     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
```

Réordonnons ses lignes en ordre décroissant de valeur de la variable `de1` et en ordre alphabétique ascendant de valeur de la variable `lanceur` en cas d'égalité. Un type d'ordre descendant pour une variable s'obtient en précédent son nom de l'opérateur `-` dans l'appel à la fonction `setorder`. Mis à part la façon de demander un ordre descendant qui diffère, la fonction `setorder` s'utilise comme la fonction `arrange` de `dplyr`.

```
setorder(dataEx_dt, - de1, lanceur)
```

Remarquons que cette commande ne retourne rien. Elle a toutefois eu un effet (que nous pourrions qualifier d'effet de bord). Elle a modifié le data table `dataEx_dt`.

```
dataEx_dt
```

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

Pour changer l'ordre des variables d'un data table par référence, il suffit de fournir à la fonction `setcolorder` ce data table, puis un vecteur des noms de variables dans l'ordre souhaité.

```
setcolorder(dataEx_dt,  c("lanceur", "de1", "de2"))
dataEx_dt
```

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

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.

Toutefois, l'expression mise en forme de jeux de données réfère le plus souvent en statistique à la distinction entre les mises en forme « large » et « longue », qui 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" "Luc" ...
##  - 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 <- data.frame(
  etudiant = c("a", "b", "c", "d"),
  TP1 = c(90, 80, 70, 60),
  TP2 = c(95, 74, 89, 85)
)
```

```
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 longue

```
notes_long <- data.frame(
  etudiant = c("a", "b", "c", "d", "a", "b", "c", "d"),
  note = c(90, 80, 70, 60, 95, 74, 89, 85),
  evaluation = rep(c("TP1", "TP2"), each = 4)
)
```

```
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 valeur (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`

Mise en forme large pure : une ligne par individu, une colonne par variable

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

```
plants_large <- data.frame(
  Plant = c("a", "b", "c"),
  Hauteur_Temps1 = c(67, 59, 62),
  Hauteur_Temps2 = c(69, 65, 66),
  Diametre_Temps1 = c(15, 16, 13),
  Diametre_Temps2 = c(15, 17, 14),
  stringsAsFactors = FALSE
)
```

```
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
```

Mise en forme 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 <- data.frame(
  Plant = rep(c("a", "b", "c"), times = 2),
  Temps = rep(1:2, each = 3),
  Hauteur = c(67, 59, 62, 69, 65, 66),
  Diametre = c(15, 16, 13, 15, 17, 14),
  stringsAsFactors = FALSE
)
```

```
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
```

Mise en forme longue pure : toutes les valeurs sont empilées dans une seule colonne

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

```
plants_long <- data.frame(
  Plant = rep(c("a", "b", "c"), times = 4),
  Temps = rep(rep(1:2, each = 3), times = 2),
  Variable = rep(c("Hauteur", "Diametre"), each = 6),
  Valeur = c(67, 59, 62, 69, 65, 66, 15, 16, 13, 15, 17, 14),
  stringsAsFactors = FALSE
)
```

```
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 <- data.frame(
  Plant = rep(c("a", "b", "c"), times = 4),
  Variable = rep(
    c("Hauteur_Temps1", "Hauteur_Temps2", "Diametre_Temps1", "Diametre_Temps2"),
    each = 3
  ),
  Valeur = c(67, 59, 62, 69, 65, 66, 15, 16, 13, 15, 17, 14),
  stringsAsFactors = FALSE
)
```

```
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 pairées* dans le cas particulier de deux variables), un test approprié à réaliser serait un test de comparaison de moyennes pour données pairées. 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 pairées 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.

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
```

Les noms `stack` et `unstack` viennent du fait que pour passer d'une mise en forme large vers une mise en forme longue, des valeurs doivent être empilées (en anglais *stacked*) dans une même colonne, alors que pour réaliser la transformation inverse elles doivent être désempilées (en anglais *unstacked*).

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

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 avec `reshape`, nous pouvons procéder comme suit.

```r
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 la 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 (`data`) sous la mise en forme large contenant les valeurs 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 où sont empilées les valeurs provenant de plusieurs colonnes dans la mise en forme large.

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

- `timevar` : nom de la colonne dans la mise en forme longue contenant, pour chaque ligne, l'identifiant de la variable de laquelle la valeur qui se trouve dans la colonne identifiée par `v.names` est une mesure (ou de laquelle les valeurs qui se trouvent dans les colonnes identifiées par `v.names` sont des mesures).
- `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 (`data`) sous la mise en forme large. 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 = "IDlancer",
  ids = paste0("l", 1:10)
)
dataEx_long2
```

```
##         lanceur  de resultat IDlancer
## l1.de1      Luc de1        2       l1
## l2.de1      Luc de1        3       l2
```

```
## l3.de1     <NA> de1        4       l3
## l4.de1      Luc de1        1       l4
## l5.de1      Luc de1        2       l5
## l6.de1      Kim de1        3       l6
## l7.de1      Kim de1        5       l7
## l8.de1      Kim de1        6       l8
## l9.de1      Kim de1        5       l9
## l10.de1     Kim de1        4      l10
## l1.de2      Luc de2        1       l1
## l2.de2      Luc de2        4       l2
## l3.de2     <NA> de2        2       l3
## l4.de2      Luc de2        3       l4
## l5.de2      Luc de2        5       l5
## l6.de2      Kim de2        4       l6
## l7.de2      Kim de2       NA       l7
## l8.de2      Kim de2        2       l8
## l9.de2      Kim de2        5       l9
## l10.de2     Kim de2        3      l10
```

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 IDlancer de1 de2
## l1.de1      Luc       l1   2   1
## l2.de1      Luc       l2   3   4
## l3.de1     <NA>       l3   4   2
## l4.de1      Luc       l4   1   3
## l5.de1      Luc       l5   2   5
## l6.de1      Kim       l6   3   4
## l7.de1      Kim       l7   5  NA
## l8.de1      Kim       l8   6   2
## l9.de1      Kim       l9   5   5
## l10.de1     Kim      l10   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 ...
##  $ IDlancer: chr  "l1" "l2" "l3" "l4" ...
##  - 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 "IDlancer"
##   ..$ 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 ...
##  $ IDlancer: chr  "l1" "l2" "l3" "l4" ...
```

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

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

```
##         IDlancer lanceur.de1 resultat.de1 lanceur.de2 resultat.de2
## l1.de1        l1         Luc            2         Luc            1
## l2.de1        l2         Luc            3         Luc            4
## l3.de1        l3        <NA>            4        <NA>            2
## l4.de1        l4         Luc            1         Luc            3
## l5.de1        l5         Luc            2         Luc            5
## l6.de1        l6         Kim            3         Kim            4
## l7.de1        l7         Kim            5         Kim           NA
## l8.de1        l8         Kim            6         Kim            2
## l9.de1        l9         Kim            5         Kim            5
## l10.de1      l10         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 valeurs doivent être réparties en plusieurs colonnes, il faut aussi spécifier l'argument `v.names`.

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

```
##         lanceur IDlancer resultat.de1 resultat.de2
## l1.de1      Luc       l1            2            1
## l2.de1      Luc       l2            3            4
```

```
## l3.de1     <NA>       l3            4            2
## l4.de1      Luc       l4            1            3
## l5.de1      Luc       l5            2            5
## l6.de1      Kim       l6            3            4
## l7.de1      Kim       l7            5           NA
## l8.de1      Kim       l8            6            2
## l9.de1      Kim       l9            5            5
## l10.de1     Kim      l10            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 `IDlancer`, renommer les colonnes contenant les résultats des lancers de dés, réordonner les colonnes et renommer les lignes.

```
dataEx_large2$IDlancer <- NULL                                  # retirer une colonne
colnames(dataEx_large2)[2:3] <- c("de1", "de2")                 # renommer des colonnes
dataEx_large2 <- dataEx_large2[, c("de1", "de2", "lanceur")]    # réordonner les 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
```

Fonctions `pivot_longer` et `pivot_wider` du package `tidyr`

Le package `tidyr` offre les fonctions suivantes, permettant d'effectuer des transformations de mise en forme :

- mise en forme large vers longue : `pivot_longer`,
- mise en forme longue vers large : `pivot_wider`.

Ces fonctions remplacent les fonctions `gather` et `spread` depuis septembre 2019. Illustrons l'utilisation des fonctions `pivot_longer` et `pivot_wider`.

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, vers une mise en forme longue avec la fonction `pivot_longer`.

```
dataEx_long3 <- pivot_longer(
  data = dataEx,
  cols = c(de1, de2),
  names_to = "de",
  values_to = "resultat"
)
dataEx_long3
```

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

Il faut fournir à `pivot_longer` les arguments :

- `data` : le data frame (ou tibble) dont la mise en forme est à transformer ;
- `cols` : les noms des colonnes dans le jeu de données d'origine (`data`) sous la mise en forme large contenant les données à empiler dans une seule colonne sous la mise en forme longue.

Pour rendre le résultat plus facilement interprétable, il est ausi recommandé de fournir une valeur aux arguments suivants :

- `names_to` : le nom de la colonne dans la mise en forme longue servant à identifier les variables dont les valeurs dans la nouvelle colonne de valeurs empilées sont des mesures ;
- `values_to` : le nom de la colonne dans la mise en forme longue où seront empilées les valeurs provenant de plusieurs colonnes sous la mise en forme large.

Remarquons que la fonction `pivot_longer` retourne un tibble et n'empile pas les valeurs dans le même ordre que `reshape`. De plus, `pivot_longer` 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 `pivot_wider` 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 `pivot_wider`, réalisé en utilisant comme jeu de données d'origine `dataEx_long2`.

```
dataEx_large3 <- pivot_wider(
  data = dataEx_long2,
  names_from = de,
  values_from = resultat
)
dataEx_large3
```

```
## # A tibble: 10 x 4
##    lanceur IDlancer   de1   de2
##    <fct>   <chr>    <dbl> <dbl>
##  1 Luc     l1           2     1
##  2 Luc     l2           3     4
##  3 <NA>    l3           4     2
##  4 Luc     l4           1     3
##  5 Luc     l5           2     5
##  6 Kim     l6           3     4
##  7 Kim     l7           5    NA
##  8 Kim     l8           6     2
##  9 Kim     l9           5     5
## 10 Kim     l10          4     3
```

La fonction a réussi à réaliser la tâche souhaitée en fournissant seulement des valeurs aux arguments `data`, `names_from` et `values_from`, définis comme suit :

- `data` : le data frame (ou tibble) dont la mise en forme est à transformer ;
- `names_from` : le nom de la colonne dans la mise en forme longue contenant les identifiants des variables desquelles les valeurs empilées sont des mesures ;
- `values_from` : le nom de la ou des colonnes dans la mise en forme longue où sont empilées les valeurs qui doivent maintenant être réparties dans plusieurs colonnes.

Les références proposent de bonnes sources d'information pour en apprendre davantage à propos des fonctions `pivot_longer` et `pivot_wider`.

Fonctions `melt` et `dcast` du package `data.table`

Le package `data.table` contient les fonctions suivantes, permettant d'effectuer des transformations de mise en forme :

- mise en forme large vers longue : `melt`,
- mise en forme longue vers large : `dcast`.

Ces fonctions sont en fait des extensions aux data tables des fonctions portant les mêmes noms dans le package `reshape2`, qui n'est maintenant plus mis à jour. Illustrons l'utilisation des fonctions `melt` et `dcast`.

Exemple de transformation vers une mise en forme longue :

Voici une commande pour transformer le data table `dataEx_dt` de sa mise en forme d'origine, large, vers une mise en forme longue avec la fonction `melt` du package `data.table`.

```
dataEx_long4 <- melt(
  data = dataEx_dt,
  measure.vars = c("de1", "de2"),
  variable.name = "de",
  value.name = "resultat"
)
dataEx_long4
```

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

Les arguments de `melt` portent des noms différents de ceux de `pivot_longer` (mis à part `data`), mais ils peuvent être définis de façon similaire :

- `data` : le data table dont la mise en forme est à transformer ;
- `measure.vars` : les noms des colonnes dans le jeu de données d'origine (`data`) sous la mise en forme large contenant les données à empiler dans une seule colonne sous la mise en forme longue ;
- `variable.name` : le nom de la colonne dans la mise en forme longue servant à identifier les variables dont les valeurs dans la nouvelle colonne de valeurs empilées sont des valeurs ;
- `value.name` : le nom de la colonne dans la mise en forme longue où seront empilées les valeurs provenant de plusieurs colonnes sous la mise en forme large.

Exemple de transformation vers une mise en forme large :

Comme pour la fonction `pivot_wider`, la fonction `dcast` a besoin que le jeu de données à transformer contiennent une colonne pour identifier les individus. Voici donc un exemple d'utilisation de la fonction `dcast`, réalisé en utilisant comme jeu de données d'origine le data frame `dataEx_long2` converti en data table.

```
dataEx_long2_dt <-as.data.table(dataEx_long2)
dataEx_large4 <- dcast(
  data = dataEx_long2_dt,
  formula = lanceur + IDlancer ~ de,
  value.var = "resultat"
)
```

```
dataEx_large4
```

```
##     lanceur IDlancer de1 de2
##  1:    <NA>       l3   4   2
##  2:     Kim      l10   4   3
##  3:     Kim       l6   3   4
##  4:     Kim       l7   5  NA
##  5:     Kim       l8   6   2
##  6:     Kim       l9   5   5
##  7:     Luc       l1   2   1
##  8:     Luc       l2   3   4
##  9:     Luc       l4   1   3
## 10:     Luc       l5   2   5
```

Les arguments de `dcast` sont différents des arguments de toutes les fonctions de modification de mise en forme vues jusqu'à maintenant.

- `data` : le data table dont la mise en forme est à transformer ;
- `formula` : formule de la forme `LHS ~ RHS`, où
  - `LHS` = noms des colonnes identifiant les individus ou à conserver telles quelles dans la sortie (séparées par `+` s'il y en a plus d'une),
  - `RHS` = nom de la colonne identifiant les variables desquelles les valeurs empilées sont des mesures ;
- `value.var` = le nom de la colonne dans la mise en forme longue où sont empilées les valeurs qui doivent maintenant être réparties dans plusieurs colonnes.

La fonction `dcast` va plus loin que la modification de mise en forme de jeux de données. Elle peut aussi agréger les données référant à un même individu en appliquant la fonction fournie à son argument `fun.aggregate`. Pour plus d'informations, voir les références.

Résumé

R de base (principalement les packages `base` et `stats`)

Transformation de variables

- Ajout ou modification de variables dans un data frame :
  - assignation à une colonne, nouvelle pour un ajout,
  - fonction `transform`,
  - concaténation de colonnes avec `cbind` ou `data.frame` ;
- 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 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,
  - `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

Sélection de sous-ensembles de données

- Sélectionner des observations (lignes) et des variables (colonnes) : opérateur `[`, fonction `subset`.
- Sélectionner les observations (lignes) ne contenant aucune donnée manquante : `na.omit`.

Combinaison 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 deux jeux de données (jointure en identifiant les lignes correspondantes) : `merge`
  - argument `by`, ou arguments `by.x` et `by.y`, pour spécifier les variables d'association (celles à partir desquelles établir les correspondances entre les lignes) ;
  - argument `all`, ou arguments `all.x` et `all.y`, pour spécifier les lignes à conserver (donc le type de jointure à réaliser).

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 ordonner les données selon l'ordre spécifié
  → pour effectuer l'ordonnancement, il faut fournir le résultat retourné par `order` en argument à l'opérateur d'extraction `[`
  - argument `decreasing = TRUE` pour 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 donnée.
- `stack` et `unstack` : modifient la mise en forme, fonctions simplistes
- `reshape` : modifie la mise en forme ; les arguments de `reshape` sont les suivants
  - 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 mise en forme longue ou intermédiaire ;
    - `varying` : noms colonnes contenant les valeurs dans le mise en forme large ou intermédiaire.

`tidyverse` (principalement les packages `dplyr` et `tidyr`)

- Transformation de variables : `mutate` du package `dplyr`.
- Variable conditionnelle à une autre : `if_else` du package `dplyr`.
- Manipulation de chaînes de caractères : package `stringr`.
- Manipulation de dates : package `lubridate`.
- Sélection de sous-ensembles de données :
  - sélection de lignes : `filter` du package `dplyr` ;
  - sélection de colonnes : `select` du package `dplyr` ;
  - sélection des lignes ne contenant aucune donnée manquante : `drop_na` du package `tidyr`.
- Combinaison de données :
  - concaténation de lignes : `bind_rows` ;
  - concaténation de colonnes : `bind_cols` du package `dplyr` ;
  - fusion par association : `inner_join`, `full_join`, `left_join`, `right_join`, etc., du package `dplyr`.
- Modification de l'ordre des données : `arrange` du package `dplyr`,
  - encadrer les noms de variables (colonnes) d'un appel à la fonction `desc` du package `tidyr` pour obtenir un ordre décroissant.
- Changement de mise en forme de jeux de données :
  - transformation vers un format plus long : `pivot_longer` du package `tidyr` ;
  - transformation vers un format plus large : `pivot_wider` du package `tidyr`.

Package `data.table`

- Transformation de variables : opérateur `:=` (modification par référence).
- Variable conditionnelle à une autre : `fifelse`.
- Sélection de sous-ensembles de données :
  - sélection de lignes et de colonnes : opérateur `[`, méthode `subset` ;
  - sélection des lignes ne contenant aucune donnée manquante : méthode `na.omit`.
- Combinaison de données :
  - concaténation de lignes : `rbindlist` ;
  - fusion par association : méthode `merge`.
- Modification de l'ordre des données :
  - ordre des lignes : `setorder` ;
  - ordre des colonnes : `setcolorder`.
- Changement de mise en forme de jeux de données :
  - transformation vers un format plus long : `melt` ;
  - transformation vers un format plus large : `dcast`.

Tableau résumé des fonctions pour la transformation de mise en forme :

Transformation	Package `utils`	Package `stats`	Package `tidyr`	Package `data.table`
large vers long	`stack`	`reshape (direction = "long")`	`pivot_longer`	`melt`
long vers large	`unstack`	`reshape (direction = "wide")`	`pivot_wider`	`dcast`

Références

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

Pour aller plus loin en manipulation de chaînes de caractères : Package `stringr` :

- https://stringr.tidyverse.org/
- Vignette : Introduction to stringr
- Tutoriel d'étudiants : Faciliter la manipulation de chaînes de caractères avec le package stringr

Expressions régulières :

- documentation R : `help(regexp)`
- Page Wikipédia : Expression régulière
- Wickham, H. et Grolemund, G. (2016). R for Data Science. O'Reilly Media, Inc., Chapitre 14. URL https://r4ds.had.co.nz/strings.html
- Feuille de triche : Basic regular expressions in R

Autres références en manipulation de chaînes de caractères en R :

- Sanchez, G. (2013) Handling and Processing Strings in R. Trowchez Editions. Berkeley. URL http://gastonsanchez.com/Handling_and_Processing_Strings_in_R.pdf
- Article de blogue : How to work with strings in base R - An overview of 20+ methods for daily use

Pour aller plus loin en manipulation de dates :

- documentation R à propos de tous les format possibles : `help(strptime)`,

Package `lubridate` :

- https://lubridate.tidyverse.org/
- 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 manipulation de jeux de données :

- Article de blogue : A data.table and dplyr tour

Package `dplyr` :

- https://dplyr.tidyverse.org/
- Tutoriel d'étudiants : Manipulations de base sur un jeu de données

Package `tidyr` :

- https://tidyr.tidyverse.org/
- Tutoriel d'étudiants : Réorganisation des données par concaténation et séparation d'observations
- Article de blogue : Tidy Animated Verbs
- Présentation : A Graphical Introduction to tidyr's pivot_*()

Package `data.table` :

- https://rdatatable.gitlab.io/data.table/index.html
- Tutoriel d'étudiants : Fonctionnalités de l'opérateur `[` du package data.table

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

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

Annexe

Exemples supplémentaires de changement de mise en forme

Voici quelques exemples supplémentaires de changements de mise en forme effectués avec la fonction `reshape` ou avec le package `tidyr`.

Mise en forme large pure vers mise en forme 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
```

```
# Avec pivot_longer :
# Il faut réaliser la transformation en deux temps,
# parce que l'argument values_to accepte seulement un nom.
hauteur_long <- pivot_longer(
  data = plants_large,
  cols = c(Hauteur_Temps1, Hauteur_Temps2),
  names_to = "Temps",
  names_prefix = "Hauteur_Temps",
  values_to =  "Hauteur"
)
diametre_long <- pivot_longer(
  data = plants_large,
  cols = c(Diametre_Temps1, Diametre_Temps2),
  names_to = "Temps",
  names_prefix = "Diametre_Temps",
  values_to =  "Diametre",
)
# Il ne reste qu'à fusionner les deux `data frames tibbles
# et à éliminer des variables.
full_join(
  x = select(hauteur_long, Plant, Temps, Hauteur),
```

```
  y = select(diametre_long, Plant, Temps, Diametre)
)
```

```
## Joining, by = c("Plant", "Temps")
```

```
## # A tibble: 6 x 4
##   Plant Temps Hauteur Diametre
##   <chr> <chr>   <dbl>    <dbl>
## 1 a     1          67       15
## 2 a     2          69       15
## 3 b     1          59       16
## 4 b     2          65       17
## 5 c     1          62       13
## 6 c     2          66       14
```

Mise en forme large pure vers mise en forme longue pure

```
# 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
```

```
# Avec pivot_longer :
pivot_longer(
  data = plants_large,
  cols = c(Hauteur_Temps1, Hauteur_Temps2, Diametre_Temps1, Diametre_Temps2),
  names_to = "Variable",
  values_to =  "Valeur"
)
```

```
## # A tibble: 12 x 3
##    Plant Variable        Valeur
##    <chr> <chr>            <dbl>
##  1 a     Hauteur_Temps1      67
##  2 a     Hauteur_Temps2      69
##  3 a     Diametre_Temps1     15
```

```
##  4 a     Diametre_Temps2     15
##  5 b     Hauteur_Temps1      59
##  6 b     Hauteur_Temps2      65
##  7 b     Diametre_Temps1     16
##  8 b     Diametre_Temps2     17
##  9 c     Hauteur_Temps1      62
## 10 c     Hauteur_Temps2      66
## 11 c     Diametre_Temps1     13
## 12 c     Diametre_Temps2     14
```

Mise en forme intermédiaire vers mise en forme longue pure

```
# 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 pivot_longer :
pivot_longer(
  data = plants_inter,
  cols = c(Hauteur, Diametre),
  names_to = "Variable",
  values_to =  "Valeur"
)
```

```
## # A tibble: 12 x 4
##    Plant Temps Variable Valeur
##    <chr> <int> <chr>     <dbl>
##  1 a         1 Hauteur      67
##  2 a         1 Diametre     15
##  3 b         1 Hauteur      59
##  4 b         1 Diametre     16
##  5 c         1 Hauteur      62
##  6 c         1 Diametre     13
##  7 a         2 Hauteur      69
##  8 a         2 Diametre     15
```

```
##  9 b         2 Hauteur      65
## 10 b         2 Diametre     17
## 11 c         2 Hauteur      66
## 12 c         2 Diametre     14
```

Mise en forme longue pure vers mise en forme 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 pivot_wider :
pivot_wider(
  data = plants_long,
  names_from = Variable,
  values_from = Valeur
)
```

```
## # A tibble: 6 x 4
##   Plant Temps Hauteur Diametre
##   <chr> <int>   <dbl>    <dbl>
## 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
```

Mise en forme longue pure vers mise en forme large pure

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
```

```r
# Avec pivot_wider :
pivot_wider(
  data = plants_long2,
  names_from = Variable,
  values_from = Valeur
)
```

```
## # A tibble: 3 x 5
##   Plant Hauteur_Temps1 Hauteur_Temps2 Diametre_Temps1 Diametre_Temps2
##   <chr>          <dbl>          <dbl>           <dbl>           <dbl>
## 1 a                 67             69              15              15
## 2 b                 59             65              16              17
## 3 c                 62             66              13              14
```

Mise en forme intermédiaire vers mise en forme large pure

```r
# 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
```

```r
# Avec pivot_wider :
pivot_wider(
  data = plants_inter,
  names_from = Temps,
  values_from = c(Hauteur, Diametre)
)
```

```
## # A tibble: 3 x 5
##   Plant Hauteur_1 Hauteur_2 Diametre_1 Diametre_2
##   <chr>     <dbl>     <dbl>      <dbl>      <dbl>
## 1 a            67        69         15         15
## 2 b            59        65         16         17
## 3 c            62        66         13         14
```