

TP 3 (et début de 4) : Manipulation de données (data mangling)

Merci de lire la fiche de TP en détail. Avant d'appeler le responsable du TP, vérifier **systématiquement** :

- que vous avez lu le message d'erreur (et traduit s'il est partiellement en anglais),
- que vous avez bien lu la fiche en détail,
- que votre code est bien formaté (espace, retour à la ligne, etc.) et qu'il n'y a pas de fautes de frappe,
- et que vous avez cherché dans les pages d'aide de R.

Les lignes qui commencent par > dans ce fichier Rmd sont des lignes qui correspondent à des questions qui vous sont posées.

Ce TP est largement inspiré du livre de Wickham et Grolemund, *R for Data Science*.

1 Importation de données

Il existe de nombreuses façons d'importer des données avec R.

Les fonctions historiques de R pour importer les données sont :

Fonction	Utilisation
<code>read.table</code>	Fichiers dont les champs sont séparés par des espaces
<code>read.csv</code>	Fichiers dont les champs sont séparés par des virgules
<code>read.csv2</code>	Fichiers dont les champs sont séparés par des points-virgules

Elles sont aujourd'hui dépassées par les fonctions des packages `readr`, `readxl`... Le package `readr` a des fonctions synonymes de celles-ci, mais environ 10 fois plus rapide, ce qui compte à l'ère du big data. Le package `readr` fait partie de la famille de `tidyverse`, que l'on avait essayé d'installer au premier TP.

Vérifier que le package `tidyverse` est bien installé sur votre compte. Si ce n'est pas le cas, refaire l'installation.

```
library(tidyverse)
```

Cette ligne de commande charge huit packages dont `tibble` et `readr`.

1.1 Commençons

Les fonctions `read_csv` et `read_csv2` permettent de lire des fichiers CSV (comma-separated values). Dans les fichiers CSV, les champs sont séparés par des virgules. Ce n'est pas très pratique dans les pays francophone, où la virgule sert de séparateur décimal. Dans ce cas, les champs sont séparés par des points-virgules, et il faut utiliser la seconde fonction.

Lorsque le séparateur est une tabulation, on utilise `read_tsv` est la fonction à utiliser. Enfin, lorsque le séparateur de champs est un caractère bien déterminé, mais différents de tous ceux-ci, il faut utiliser `read_delim` et spécifier le délimiteur.

Le premier argument de ces fonctions est un nom de fichier, avec son chemin partiel (depuis le working directory), ou complet.

Voir l'antisèche, page 1, colonne du milieu.

Un exemple :

Mettre le fichier `heights.csv` dans le répertoire de votre projet correspondant à ce cours.

```
heights <- read_csv("heights.csv")
class(heights)
nrow(heights)
ncol(heights)
heights
```

Lorsque l'on lance `read_csv`, R affiche les propriétés des colonnes de la table : leurs noms et leurs types. Contrairement à ce que nous avons vu dans les précédents TP, la table de données est stockée dans un objet de type `tbl_df`, qui est une amélioration des `data.frame`.

On peut aussi fournir le contenu du fichier CSV directement dans la ligne de commande, ce qui est pratique pour faire des tests.

```
toto <- read_csv2("a; b; c
                  1,1; 2,2; 3,1
                  -3,14; 4,25; -77")
toto
```

Remarquer ici que R a compris que la virgule servait de séparateur décimal. Dans tous les cas ci-dessus, les fonctions ont utilisé la première ligne pour attribuer des noms aux colonnes. Lorsque la première ligne contient directement les valeurs de la première observation, il faut utiliser l'option `col_names = FALSE`. On peut aussi utiliser ce paramètre optionnel pour donner des noms aux colonnes :

```
read_csv("1, 2, 3
          4, 5, 6")
read_csv("1, 2, 3
          4, 5, 6", col_names = FALSE)
read_csv("1, 2, 3
          4, 5, 6", col_names = c("riri", "fifi", "loulou"))
```

Enfin, on peut préciser comment on a encodé les valeurs manquantes. Il est assez standard d'utiliser un point tout seul. Dans ce cas, il faut utiliser le paramètre optionnel `na = .`. Par exemple :

```
read_csv("a, b, c
          1, ., 2
          4, 5, .",
          na = ".")
```

Enfin, les options `skip` et `comment` servent respectivement à sauter les premières lignes du fichier, ou à définir un caractère signalant les lignes de commentaires (par exemple `#`).

1.2 Analyser un vecteur

Chaque colonne du fichier que l'on veut importer est analysé à l'aide d'une fonction `parse_*`, où l'étoile est remplacée par le type de colonne que l'on veut obtenir.

```
str( parse_integer(c("1", "2", "3")) )
str( parse_date(c("2018-09-24", "2018-09-25", "2020-01-30")) )
```

L'option `na =` permet, comme plus haut, de fixer la convention pour les données non observées :

```
str( parse_integer(c("1", "2", "3", "."), na = ".") )
str( parse_date(c("2018-09-24", "2018-09-25", ".", "2020-01-30"), na = ".") )
```

En cas de problèmes avec ces fonctions comme ci-dessous, on peut utiliser la fonction `problems` sur l'object renvoyé. Par exemple :

```
toto <- parse_integer(c("123", "345", "abc", "123.45"))
problems(toto)
```

Comprendre les problèmes dans les résultats du chunk ci-dessus.

Voici les fonctions `parse_*`, qui transforment un vecteur de chaîne de caractères en vecteur de type de données :

Fonction	Type de sortie
<code>parse_logical</code>	crée un logical
<code>parse_double</code>	crée un double, encore appelé numeric
<code>parse_character</code>	crée un character, utile pour changer l'encodage de caractère
<code>parse_factor</code>	crée un factor

S'ajoutent `parse_datetime`, `parse_date` et `parse_time`, qui sont hors de programme pour ce cours.

1.3 Nombres

La façon dont sont représentés les nombres varie d'un pays à l'autre, et peut parfois inclure des symboles de contexte (%, \$, €, ¥, %...) dont on veut se débarrasser. En outre, deux milles vingt trois et cinq dixième s'écrit : "1.023,5" en France, et "1,023.5" aux US.

Il faut donc préciser à R quel est le séparateur décimal et le séparateur de regroupement. En France, le premier est une "," et le second un ".". Aux US, c'est la convention inverse ! Voici comment on fait :

```
parse_number("1.023,5", locale = locale(decimal_mark = ",",      # FR
                                         grouping_mark = "."))    # FR
parse_number("1,023.5", locale = locale(decimal_mark = ".",      # US
                                         grouping_mark = ","))    # US
```

1.4 Stratégie d'analyse d'un fichier

Revenons maintenant à l'importation d'un fichier. Quand on utilise une fonction `read_*`, celle-ci utilise les mille premières lignes pour deviner le type de la colonne.

Une fois qu'il a trouvé le type de colonne, on peut considérer qu'il met toutes les chaînes de caractères dans un vecteur de type `character`, et qu'il utilise ensuite une des fonctions `parse_*` pour transformer ces chaînes en nombres, `factor`, `logical`,...

Il peut se produire deux problèmes, en particulier sur les grandes tables de données :

- les 1000 premières lignes ne sont que des cas particuliers, et le choix de type de colonnes est trop restrictif (par exemple, `integer` au lieu de `double`)
- les 1000 premières lignes ne contiennent que des valeurs manquantes dans certaines colonnes...

C'est le cas du fichier `challenge.csv`, que vous devez recopier dans votre working directory. Il faut alors utiliser l'argument optionnel `col_types` de ces fonctions `read_*`. Voici un exemple :

```
challenge <- read_csv("challenge.csv",
                      col_types = cols(
                        x = col_double(),
                        y = col_date()
                      ))
```

Ouvrir le fichier csv et comprendre pourquoi on a proposé ces deux types. Quel type de problème se produit pour chacune des colonnes si on n'utilise pas cette option `col_types`.

On peut aussi lui spécifier le nombre de lignes qu'il doit utiliser pour deviner le type de chacune des colonnes, avec l'option `guess_max`. Dans notre cas, tous les problèmes disparaissent si on fixe ce paramètre à 1001 :

```
challenge <- read_csv("challenge.csv", guess_max = 1001)
```

1.5 Autres formats

Le package `haven` permet de lire des données au format SPSS, Stata et SAS, qui sont des logiciels de statistique / analyse de données.

Le package `readxl` permet d'importer depuis des fichiers Excel `.xls` et `.xlsx`. Nous vous laissons lire (en dehors du TP) la documentation de ce package, en particulier les deux vignettes si besoin.

D'autres packages permettent d'aller lire directement dans des bases de données ou de faire du "webscraping"...

Enfin, le package `rio` propose une fonction `import` qui est entièrement automatisée, et qui marche dans les cas les plus classiques.

2 Transformation de données

Nous allons illustrer cette partie sur le jeu de données `flights`, issu du package `nycflights13`.

Commencer par installer ce package

```
library(nycflights13)
?flights
```

Cette table de données contient des informations sur tous les 336 776 vols qui sont partis de New York en 2013 (depuis les aéroports JFK, LGA ou EWR), tels qu'enregistrés dans la base de données du *US Bureau of Transportation Statistics*. La page d'aide de `flights` vous donne des explications sur ce jeu de données. Voici les 6 premières lignes (attention, il y a 19 colonnes) :

```
head(flights)
```

Noter que cliquer sur le triangle à droite des noms de colonnes permet d'afficher les colonnes suivantes.

2.1 Filtrer

On peut filtrer les lignes d'une telle table à l'aide de la fonction `filter` du package `dplyr`. Par exemple, si on s'intéresse uniquement aux vols du 1er janvier 2013, ou au 25 décembre :

```
filter(flights, month == 1, day == 1)
filter(flights, month == 12, day == 25)
```

Combien y a-t-il de vols sur chacun de ces deux jours ?

On peut alors enregistrer le résultat dans un nouvel objet, par exemple `jan1` :

```
jan1 <- filter(flights, month == 1, day == 1)
```

Remarquer que tous ce qui suit la virgule dans cette ligne sont des conditions sur les colonnes : `month == 1` et `day == 1`, et que seules sont sélectionnées les lignes qui satisfont toutes les conditions.

ATTENTION, il ne faut pas confondre le signe `==` de test d'égalité, avec le signe `=`, qui sert à donner la valeur d'un paramètre optionnel d'une fonction. Par exemple :

```
filter(flights, month = 1, day = 1)
```

Cette expression propose d'évaluer la fonction `filter` sur la table `flights`, avec le paramètre option `month` de la fonction `filter` égal à 1, et le second paramètre optionnel `day` égal à 1 aussi. Comme ni `month`, ni `day` ne sont des paramètres optionnels de `filter`, cela produit un message d'erreur. Il est plus ou moins parlant suivant les versions de R et du package `dplyr` utilisé.

On peut faire la même chose que pour un `data.frame`, et utiliser `[,]` pour sélectionner, mais cette ligne de commande est plus compliquée à lire pour l'oeil humain :

```
flights[flights$month == 1 & flights$day == 1, ]
```

ATTENTION aussi, l'égalité entre deux nombres à virgules flottantes est une chose qui est un peu délicate, à cause des erreurs d'arrondi. Par exemple, $(\sqrt{2})^2$ ne produit pas exactement 2 numériquement. On peut dans ce cas utiliser la fonction `near` qui permet de savoir si deux nombres à virgules flottantes sont proches.

```
sqrt(2) ^ 2 == 2
near(sqrt(2) ^ 2, 2)
```

Si on veut sélectionner les vols des mois de novembre et décembre, on peut soit utiliser le ou logique (`|`), soit utiliser l'opérateur `%in%` :

```
filter(flights, month == 11 | month == 12)
filter(flights, month %in% c(11, 12))
```

Comment ferait-on pour sélectionner uniquement les jours pairs ? (on pourra utiliser la fonction `seq`, déjà vue dans les TP précédents)

2.2 Valeurs manquantes

Attention, les règles arithmétiques sont un peu particulières avec NA. On notera le résultat des opérations ci-dessous :

```
# ça, pourquoi pas (quand on ne sait pas, on ne sait pas non plus)
NA > 5

## [1] NA

10 == NA

## [1] NA
```

```
NA + 10

## [1] NA
# celle là est piègeuse :
NA == NA

## [1] NA
# Faut-il, ou ne faut-il pas connaitre un booléen pour un calcul?
NA | TRUE

## [1] TRUE
NA & FALSE

## [1] FALSE
NA & TRUE

## [1] NA
# Et alors, là, on devient fou :
NA ^ 0

## [1] 1
NA * 0

## [1] NA
# Noter, que, pour tester si quelque chose vaut NA, il faut utiliser is.na :
is.na(NA)

## [1] TRUE
```

Comprendre l'idée derrière ces résultats.

On peut aussi (sous-)échantillonner (avec ou sans remise) parmi les lignes du tableau, à l'aide des fonctions `sample_frac` et `sample_n`, suivant que l'on souhaite définir la taille du sous-échantillon comme une fraction de la taille totale, ou par le nombre de lignes dans le sous-échantillon.

Par exemple, en notant que `0.0001 * nrow(flights) = 33.66776...` :

```
sample_n(flights, 6)
sample_frac(flights, .0001)
```

Enfin, on peut choisir uniquement des lignes en fonction de leurs numéros avec la fonction `slice` :

```
slice(flights, c(123457 + 10:13, 4, 55578))
```

2.3 Organiser les lignes

La fonction `arrange` permet de trier les lignes en fonction des valeurs dans certaines colonnes. Par exemple, pour trier par date :

```
arrange(flights, year, month, day)
```

Noter qu'il n'était pas utile de mettre `year` ici, car tous les vols concernent 2013...

On peut aussi trier dans l'ordre décroissant. Dans tous les cas, les valeurs manquantes sont classées à la fin. Par exemple, pour obtenir tous les vols dans l'ordre décroissant de leurs retards à l'arrivée :

```
arrange(flights, desc(arr_delay))
```

Les retards étant en minutes, les 1272 minutes de retard correspondent à 21 h et 12 minutes. . .

Trier la table par retard au départ. Quel était le délais au départ le plus important ?

2.4 Organiser les colonnes

On peut se concentrer sur quelques colonnes à l'aide de la fonction `select`. Par exemple, on peut faire :

```
# Sélectionne les colonnes par leurs noms
select(flights, year, month, day)
# Sélectionne toutes colonnes entre year et day
select(flights, year:day)
# Supprimer ces mêmes colonnes
select(flights, -(year:day))
```

On peut utiliser des fonctions auxiliaires pour sélectionner des colonnes dont le nom commence, finit, ou contient certains motifs avec `starts_with`, `ends_with`, et `contains`. Voir l'aide de la fonction `select` :

?select

Pour renommer des colonnes, on peut utiliser la fonction `rename`. Contrairement à `select`, cette fonction conserve toutes les colonnes dont le nom n'apparaît pas explicitement dans la commande :

```
rename(flights, tail_num = tailnum)
```

ATTENTION, R fait la différence entre les majuscules et les minuscules dans les noms de colonnes (comme partout ailleurs).

On peut aussi utiliser la fonction `select` avec sa fonction auxiliaire `everything` pour réordonner les colonnes. Par exemple :

```
select(flights, time_hour, air_time, everything())
```

2.5 Ajout de nouvelles variables (colonnes)

La fonction `mutate` ajoute les nouvelles colonnes à la fin de la table. On commence donc par se concentrer sur certaines colonnes du jeu de données. Ici, on peut s'intéresser à la différence des retards (au départ et à l'arrivée), ainsi que la vitesse moyenne de vol.

```
flights_small <- select(flights,
                        year:day,
                        ends_with("delay"),
                        distance,
                        air_time)
mutate(flights_small,
       gain = arr_delay - dep_delay,
       speed = distance / air_time * 60)
```

En définissant de nouvelles colonnes, on peut utiliser les colonnes que l'on vient de créer. Par exemple, pour regarder la différence des retards par heures de vols :

```
mutate(flights_small,
       gain = arr_delay - dep_delay,
```

```
hours = air_time / 60,  
gain_per_hour = gain / hours)
```

Quel est le sens de la nouvelle colonne gain créée par la commande ci-dessus ?

Si on veut uniquement conserver les nouvelles colonnes (variables), il faut utiliser la fonction `transmute` :

```
transmute(flights,  
  gain = arr_delay - dep_delay,  
  hours = air_time / 60,  
  gain_per_hour = gain / hours)
```

À ce niveau, il faut comprendre que les opérations pour créer de nouvelles variables sont des opérations de type vectorielles. On peut donc utiliser toutes les fonctions vectorielles de R. Par exemple, dans le dernier chunk ci-dessus, on crée un vecteur `gain` en faisant la différence entre les deux vecteurs `arr_delay` et `dep_delay`, qui sont respectivement les retards au départ et à l'arrivée du vol. En particulier, on peut utiliser toutes les opérations arithmétiques (+, -, *, /, %/%, %%, ^, ...), les fonctions mathématiques (log, exp, sin, ...), les comparaisons logiques (<, <=, >, >=, !=, ==), etc.

3 Résumés par groupes

On peut utiliser la fonction `summarize` pour obtenir des résumés statistiques. Par exemple :

```
summarize(flights,  
  count = n(),  
  mean_dep_delay = mean(dep_delay, na.rm = TRUE),  
  sd_dep_delay = sd(dep_delay, na.rm = TRUE)  
)
```

Comme il n'y a qu'un seul groupe ici, le résultat de cette fonction n'est pas très épatant. On peut, en revanche, commencer par regrouper les vols en fonction de leurs dates avant d'appliquer cette fonction. Voilà un exemple :

```
flights_by_day <- group_by(flights, year, month, day)  
summarize(flights_by_day,  
  count = n(),  
  mean_dep_delay = mean(dep_delay, na.rm = TRUE),  
  sd_dep_delay = sd(dep_delay, na.rm = TRUE))
```

Calculer les retards moyens à l'arrivée en fonction de l'aéroport d'arrivée, et trier les résultats du délai le plus grand au plus petit dans le chunk ci-dessous.

3.1 Combiner des opérations avec le tuyau (pipe en anglais)

Supposons que l'on veuille étudier la relation entre la distance et le retard moyen à l'arrivée. On veut aussi supprimer les vols à destination de "HNL" (Honolulu airport), et les destinations ayant moins de 20 vols de retard. On va donc faire :

```
by_dest <- group_by(flights, dest)  
delays <- summarize(by_dest,  
  count = n(),  
  dist = mean(distance, na.rm = TRUE),  
  delay = mean(arr_delay, na.rm = TRUE))  
delays <- filter(delay, count > 20, dest != "HNL")
```


Il est un peu frustrant de devoir stocker les résultats intermédiaires dans des objets que l'on ne va pas garder (ici `by_dest`, et la première version de `delay`). Il y a une autre solution pour régler ce problème avec le tuyau `%>%`.

```
delays <- flights %>%
  group_by(dest) %>%
  summarize(count = n(),
            dist = mean(distance, na.rm = TRUE),
            delay = mean(arr_delay, na.rm = TRUE)) %>%
  filter(count > 20, dest != "HNL")
```

En fait, `x %>% f(y)` est un synonyme de `f(x, y)` et `x %>% f(y) %>% g(z)` un synonyme de `g(f(x, y), z)`. On peut, par exemple, écrire :

```
3.14159 %>% sin()
0.1234567890 %>% round(digits = 6)
```

3.2 Valeurs manquantes

Précédemment, lorsque l'on a calculé moyenne et variance du retard à l'arrivée par jour de l'année, on a ajouté l'option `na.rm = TRUE` à `mean` et `sd` pour que les calculs se passent bien. Si on ne fait pas ça, on obtient beaucoup de valeurs manquantes (qui ont tendance à être envahissantes au moindre calcul, voir les règles arithmétiques ci-dessus).

```
flights %>%
  group_by(year, month, day) %>%
  summarize(
    count = n(),
    dist = mean(distance),
    delay = mean(arr_delay)
  )
```

En fait, dans notre situation, les valeurs manquantes représentent les vols qui ont été annulés. On peut donc les supprimer, puis refaire le calcul de moyenne et écart-type, sans tenir compte des NA.

```
not_canceled <- flights %>%
  filter(!is.na(dep_delay), !is.na(arr_delay))

not_canceled %>%
  group_by(year, month, day) %>%
  summarize(
    count = n(),
    dist = mean(distance),
    delay = mean(arr_delay)
  )
```

Combien y a-t-il de valeurs manquantes dans la colonne `arr_delay` ? et dans la colonne `dep_delay` ? Faire le calcul dans le chunk ci-dessous

3.3 Exercice (TP 4)

4 Ranger des données

Les tables de données `table1`, `table2`, `table3`, `table4a`, `table4b` et `table5` représentent les mêmes informations, mais rangées différemment :

- `country` : le pays
- `year` : l'année
- `population` : la population du pays cette année là
- `cases` : le nombre de cas d'intérêt parmi la population du pays, cette année là.

Ce sont des exemples illustratifs très courts. Les voici :

```
table1
table2
table3
table4a
table4b
table5
```

4.1 Diffuser et rassembler

Un problème assez courant est celui de valeurs de variables qui sont planquées dans des noms de colonnes. C'est le cas de la `table4a`. On peut faire une opération de rassemblement (`gather`) pour retrouver la variable année dans une colonne. Voici comment on procède :

```
table4a %>%
  gather(`1999`, `2000`, key = "year", value = "cases")
```

On a multiplié le nombre de lignes par 2 (autant que d'années ou de colonnes à rassembler). Les valeurs dans ces colonnes sont rassemblées dans une unique colonne, dont on donne le nom en donnant une valeur au paramètre `value` de la fonction `gather`. Le nom de la colonne où a été extrait le chiffre qui se trouve maintenant dans la colonne `cases` est dans la variable (colonne) qui porte le nom affecté au paramètre `key` de la fonction `gather` (ici, `year`).

Attention, ici les noms de colonnes ne sont pas des noms d'objets R car ils ne commencent pas un caractère alphabétique. Dans ce cas, il faut entourer le nom de colonne de guillemets inversés (`).

Les tailles de population sont dans une seconde table, nommée `table4b`. On peut faire le même rassemblement, sauf que l'on construit maintenant la colonne `population` :

```
table4b %>%
  gather(`1999`, `2000`, key = "year", value = "population")
```

Pour rassembler ces deux tables, il faut faire une jointure (à gauche) comme suit :

```
tidy4a <- table4a %>%
  gather(`1999`, `2000`, key = "year", value = "cases")
tidy4b <- table4b %>%
  gather(`1999`, `2000`, key = "year", value = "population")
left_join(tidy4a, tidy4b)
```

Au contraire, on peut diffuser les valeurs d'une seule colonne dans plusieurs colonnes. C'est l'opération inverse de celle que l'on vient d'effectuer, et se réalise avec la fonction `spread`. Notre point de départ est la `table2`.

```
table2
table2 %>%
  spread(key = type, value = count)
```

Cette fois-ci, le nom de la future colonne est caché dans la colonne `type`, qui indique si le comptage est celui du nombre de cas, ou de la population totale. C'est ce que l'on passe dans l'argument `key` de la fonction `spread`. Et le nom de la colonne, que l'on veut diffuser sur plusieurs colonnes est passée dans l'argument `value`.

4.2 Séparer et réunifier

Dans la `table3`, les informations sur le nombre de cas et la taille de la population sont dans une seule colonne, qu'il convient de séparer en deux. On notera que ces deux nombres sont séparés par le caractère : `"/"`. Voici comment il faut faire :

```
table3
table3 %>% separate(rate, into = c("cases", "population"), sep = "/")
```

La fonction `unite` fait l'opération inverse...

5 Utiliser des données de la SNCF

La SNCF met en ligne un certain nombre de données sur le site web SNCF Open Data

On se propose d'étudier les gares de la SNCF.

5.1 Quelques jeux de données

Télécharger et mettre en forme le "référentiel des gares de voyageurs" sous R. Les différentes gares sont identifiées par un code unique, appelé code UIC.

Ce jeu de données contient de nombreuses informations par gare, en particulier leur position géographique (latitude, longitude), et dans différents systèmes de projection planaire.

Télécharger et mettre en forme le tableau "fréquentations des gares".

Faire la même chose avec les tableaux "Répartition des clients par type (Enquêtes en gare)", "Répartition des clients par âge (Enquêtes en gare)", "Répartition des clients par catégorie socio-professionnelle (Enquêtes en gare)" et "Répartition des clients par genre (Enquêtes en gare)".

5.2 Suite

À venir