S05: Manipulation avancée de données

Analyse de données quantitatives avec R

Samuel Coavoux

Manipulation de chaînes de caractères

Principe des expressions régulières

Principe

Une expression régulière (*regular expression*, désormais *regex*) consiste à chercher des motifs (*patterns*) dans des chaînes de caractères, à les capturer et à les remplacer. Il en existe dans tous les langages de programmation, mais certains langages de regex constituent des standards.

R emploie le langage Posix Extended Regular Expressions par défaut, ou le langage perl (avec l'argument `perl=TRUE`).

Quand l'utiliser

On utilise les expressions régulières comme des arguments de fonctions qui visent à manipuler, extraire, couper des variables caractères. Quelques exemples:

- pour chercher les valeurs d'une variable qui suivent un motif particulier : `grep()`
- pour substituer un motif dans une variable caractère : `gsub()`
- pour découper une variable : `tidyr::separate()`

Chaîne de caractère

Pour rechercher une chaîne de caractère, on entre simplement la chaîne en question.

```
## chercher les chaînes toto dans le vecteur x
grep("toto", x)
```

Certains caractères ne peuvent pas être cherchés de cette manière, parce qu'ils ont un sens particulier. Il s'agit de:

`.[]{}()\*+?|^$`

Pour rechercher un de ces caractère, il faut l'"échapper", c'est-à-dire ajouter un \ devant. Ainsi, pour le point, chercher `\.`

Cependant, dans R, il convient de redoubler tous les caractères backslash (\) : on écrira donc `\\.`

Quantifieurs

Il existe plusieurs quantifieurs : `*`, `+`, `?`, `{}`.

- `*` = répétition du caractère précédent 0 fois ou plus
- `+` = répétition du caractère précédent 1 fois ou plus
- `?` = répétition du caractère précédent 0 ou 1 fois
- `{}` = sélectionner le nombre de répétition

Quantifieurs : {}

On peut utiliser {} pour sélectionner un nombre exact, un minimum, ou un intervalle de répétitions :

- a{3} : répétition de a exactement trois fois
- a{3, } : répétition de a trois fois ou plus
- a{3, 5} : répétition de a trois à cinq fois

Ensemble de caractères

On peut sélectionner des ensembles de caractères avec `[]`. Dans ce cas, toutes les possibilités inclues dans les crochets sont considérées. Ainsi:

`[ab]c`

Selectionnera :

`ac`
`bc`

Mais pas:

`ab`

Ensemble par rang

On peut également employer des séries de caractères avec - ; ainsi:

`[a-d]`

selectionne a, b, c ou d. De sorte que:

`[a-zA-Z0-9]`

Sera souvent employé pour dire: n'importe quelle lettre minuscule ou majuscule, ou n'importe quel chiffre.

Alternatives: |

Les [] permettent de sélectionner des caractères appartenant à un ensemble.

| permet de sélectionner des groupes alternatifs. Par exemple, pour sélectionner soit “ab”, soit “cd”.

```
ab|cd
```

Si les alternatives sont une partie d’une chaîne plus longue, on peut les isoler avec des parenthèses ; pour sélectionner “abab” ou “abcd”:

```
ab(ab|cd)
```

Négation

Dans un ensemble, on peut inverser la sélection en employant `^`. Ainsi:

```
[^a-c]
```

sélectionner tous les caractères, sauf a, b et c.

Ancres

On peut *ancrer* un motif au début ou à la fin d'une chaîne en employant ^ (début) ou $ (fin). Par exemple:

`^ab`

Sélectionnera "abc" ou "abab" mais pas "bab".

À l'inverse:

`ab$`

Sélectionnera "abab" ou "bab" mais pas "abc".

Classes de caractères

Il est possible d'employer des ensembles de caractères pré-définis pour éviter d'avoir à mettre dans des crochets tous les caractères cherchés. Il y a deux manière de le faire, qui sont équivalentes:

- `\d` ou `[[:digit:]]` : n'importe quel nombre (équivalent à `[0-9]`)
- `\s` ou `[[:space:]]` : un espace (ou un saut de ligne). À préférer à l'espace simple ' ', car inclut également les espaces insécables, etc.
- `\w` ou `[[:word:]]` : n'importe que caractère alaphanumérique (équivalent à `[a-zA-Z0-9]`)

Sous R, on oublie pas de doubler `\` : `\\d`, `\\s`, `\\w`

Classes de caractères : négation

Il est également possible de rechercher tous les caractères qui ne *sont pas* d'un de ces types, en passant en majuscules

- `\D` ou `[^[:digit:]]` : n'importe quel caractère autre qu'un chiffre (équivalent à `[^0-9]`)
- `\S` ou `[^[:space:]]` : n'importe quel caractère qui ne soit pas un espace.
- `\W` ou `[^[:word:]]` : n'importe que caractère qui ne soit pas alaphanumérique (équivalent à `[^a-zA-Z0-9]`)

Sous R, on oublie pas de doubler `\` : `\\D`, `\\S`, `\\W`

.

Enfin, le point `.` sélectionne tous les caractères, sauf les sauts de ligne. De ce fait, on emploie souvent `.*`, qui sélectionne n'importe quelle suite de caractères.

Associations

Ces différents éléments peuvent être combinés:

```
[a-zA-Z]*\d{3}\W
```

Sélectionne une chaîne de 0 ou plusieurs lettres suivie de trois chiffre suivi d'un caractère qui n'est pas alphanumérique.

Chercher, remplacer, extraire

grep et ses variantes

On peut désormais repérer dans un vecteur caractère quelles sont les valeurs qui suivent un motif particulier. la fonction `grep()` prend deux arguments principaux, `pattern` (le motif exprimé en regex) et `x` (le vecteur caractère dans lequel on cherche).

```
grep(pattern = "@\\w+", x = d$tweets)
```

Par défaut, `grep()` renvoie un vecteur numérique : l'index des valeurs qui correspondent à l'expression régulière. Cela permet de l'utiliser dans l'indexation:

```
d[grep(pattern = "@\\w+", x = d$tweets), ]
```

grep et ses variantes

Pour tester des regex, ou pour extraire des chaînes particulières, on peut vouloir renvoyer d'autres éléments. L'argument `value = TRUE` permet de renvoyer le contenu des valeurs sélectionnées (ici le texte des tweets).

```
grep(pattern = "@\\w+", x = d$tweets, value = TRUE)
```

Dans certains cas, on préférera renvoyer un vecteur logique qui vaut `TRUE` si la valeur correspond à la regex et `FALSE` sinon. On emploie alors `grepl()` (grep logical).

```
grepl(pattern = "@\\w+", x = d$tweets)
```

Capture

L'usage de recherche est limité. Souvent, on va avoir besoin non seulement de rechercher, mais encore d'extraire ou de remplacer des motifs.

Pour cela, on a besoin de *capturer* des parties de la regex. Cela se fait avec les parenthèses : `()` ; un groupe entre parenthèse sera capturé et pourra être réemployé par la suite en invoquant `\n` où n est le numéro (le rang) du groupe. Ainsi, on capture dans la regex suivante les trois chiffres qui suivent le groupe alphanumérique:

```
\w+(\d{3})\W
```

Et on pourra les appeler avec `\1` ; si l'on a plusieurs groupes, on peut continuer avec `\2`, `\3` ... `\9`.

Extraire ou remplacer: `sub()` et `gsub()`

On peut alors employer les fonctions de *substitution* `sub()` et `gsub()`.

- `sub()` recherche le motif une seule fois dans chaque valeur, et le remplace quand elle le trouve. => utilisé pour extraire.
- `gsub()` recherche le motif plusieurs fois dans chaque valeur, et le remplace à chaque fois. => utilisé pour remplacer.

`sub()` et `gsub()` ont en commun de renvoyer un vecteur caractère avec les motifs substitués par leur remplacement. Ils prennent trois arguments principaux: `pattern`, `replacement`, et `x`.

Extraire : sub()

```
x <- c("2016-01", "2015-12-31", "1999", "1875/12")
x
```

```
## [1] "2016-01"    "2015-12-31" "1999"
## [4] "1875/12"
```

```
## Extraire les dates (les séries de chiffres)
sub(".*(\\d{4}).*", "\\1", x)
```

```
## [1] "2016" "2015" "1999" "1875"
```

Substituer: gsub()

```
x <- c("bla ", "   bla bla ", "bla bla", "bla\n")
x
```

```
## [1] "bla "         "   bla bla " "bla bla"
## [4] "bla\n"
```

```
## Enlève tous les espaces
gsub("\\s", "", x)
```

```
## [1] "bla"    "blabla" "blabla" "bla"
```

Quantifieur : ?

Certains motifs conduisent à sélectionner une chaîne *trop longue*. Par exemple, imaginons que l'on veut sélectionner un bout de chaîne, tout ce qui précède le caractère _. On pourrait écrire :

```
gsub("(.*)_.*", "\\1", c("bla_bla", "bla_bla_bla"))
```

```
## [1] "bla"     "bla_bla"
```

Cela arrive parce que _ fait partie de .. Par défaut, . est *greedy* : il avancera tant qu'il peut (jusqu'à s'arrêter ici au dernier _) ; on peut le rendre *lazy* en ajoutant ?. Dans ce cas, il s'arrêtera dès qu'il peut (dès qu'il rencontre le caractère suivant dans la regex).

```
gsub("(.*?)_.*", "\\1", c("bla_bla", "bla_bla_bla"))
```

```
## [1] "bla" "bla"
```

Autres usages

tidyr::separate()

Le package `tidyr` contient un ensemble de fonctions de manipulation de données. `separate()` permet de séparer un vecteur comprenant plusieurs variables en autant de vecteurs.

```
library(tidyr)
x <- data.frame(tags=c("#hash1;#hash2", "#hash1",
                       "#hash2;#hash3"),
                stringsAsFactors = FALSE)
separate(x, tags, into = c("tag1", "tag2"), sep=";")
```

```
## Warning: Too few values at 1 locations: 2
```

```
##     tag1   tag2
## 1 #hash1 #hash2
## 2 #hash1   <NA>
## 3 #hash2 #hash3
```

tidyr::separate_rows()

`separate_rows()` fait la même chose, mais mets chaque élément dans sa propre ligne.

```
separate_rows(x, tags, sep=";")
```

```
## # A tibble: 5 × 1
##     tags
##    <chr>
## 1 #hash1
## 2 #hash2
## 3 #hash1
## 4 #hash2
## 5 #hash3
```

Autres ressources

`stringr` est un package qui contient des fonctions avancées de manipulation de vecteurs caractères (en particulier pour réaliser un grand nombre de `gsub()` les uns à la suite des autres : `str_replace_all`).

Tidyverse

Principe

Le “tidyverse” désigne un ensemble de packages développés principalement par Hadley Wickham, dont l'objectif est de produire une manière simple et systématique de manipuler les données de façon propre (“tidy”).

On peut installer l'ensemble des packages concernés avec `install.packages("tidyverse")`. `library(tidyverse)` charge les principaux packages.

Contenu

- `haven`: importer des données depuis SAS, Stata, SPSS
- `readr`: redéfinit les fonctions `read.*()` en `read_*()`
- `tibble`: redéfinit le data.frame en tibble
- `dplyr`: manipuler des tibble
- `tidyr`: transposer des bases de données
- `stringr`: manipuler des vecteurs character
- `forcats`: manipuler des facteurs
- `ggplot2`: graphiques

Alternative: data.table

Il existe un autre package qui redéfinit complétement la façon d'interagir avec les données dans R, `data.table`. Il est particulièrement employé lorsque l'on souhaite interagir avec des data.frame de grande taille (au-delà du million de lignes).

Magrittr

Dans les langages de shell (`sh`, `bash`, `zsh`, etc.), le signe | est appelé “pipe” (tuyau). Il permet d’enchaîner plusieurs fonctions en passant le résultat de la fonction de gauche comme premier argument de la fonction de droite.

Le package, `magrittr` (“Ceci n’est pas un pipe”) contient principalement la fonction %>%, adaptant le “pipe” dans R. Il est désormais inclus automatiquement dans de nombreux autres packages, dont ceux développés par Hadley Whickam (`rvest`, `dplyr`, `tidyr`, etc.)

Sous R-Studio, le raccourci clavier ctrl+shift+M insert un pipe.

Usages

En pratique, un %>% permet d'enchaîner des fonctions sans avoir besoin de stocker le résultat dans des objets intermédiaires

```
x <- 1:10
mean(x)
```

Peut être écrit:

```
1:10 %>% mean()
```

Usages du pipe: enchaîner des opérations

On peut ainsi réécrire des opérations complexes en les enchaînant chacune sur une ligne plutôt qu'en les imbriquant les unes dans les autres.

```
x <- factor(c("43", "56", "78"))
# Transformer en caractere, puis en numérique,
# puis faire la moyenne
mean(as.numeric(as.character(x)))
```

```
## [1] 59
```

```
# Alternativement:
x %>% as.character() %>%
  as.numeric() %>%
  mean()
```

```
## [1] 59
```

dplyr

Principe

Histoire

`dplyr` est l'héritier de `plyr` (la pince), un package qui visait, pour aller vite, à simplifier et accélerer la famille de fonction `apply`. Le d de `dplyr` vient de data.frame.

Il s'agit en fait d'une façon de manipuler les data.frame qui redéfinit l'indexation au moment du recodage. On l'emploie en particulier pour recoder et préparer les données.

Le principe est d'enchaîner des actions via des %>%. Chacune des actions est définie par un **verbe**.

Verbes d'action

- Sélectionner des lignes : filter, slice (remplace `[`)
- Sélectionner des colonnes : rename, select (remplace `[`)
- Transformer des variables : mutate, mutate_all, mutate_each, mutate_at, transmute (remplace `lapply`)
- Diviser en groupes : group_by (remplace `tapply`)
- Calculer des paramètres : summarize
- Appareiller des data.frame : left_join, right_join, full_join (remplace `merge`), bind_rows, bind_cols (remplace rbind, cbind)

Du data.frame au `tibble`

`dplyr` produit des data.frame qui ont également la classe de `tbl_df` (package `tibble`). Cf. `?tibble` pour les différences entre les deux classes.

Non-standard evaluation

Les verbes d'action de dplyr ont pour particularité de prendre comme argument des noms d'objet, et non des vecteurs character. Le premier argument est **toujours** le data.frame que l'on souhaite manipuler. Ensuite, comme dans une formule `lm()`, on peut appeler les variables inclus dans ce data frame sans avoir à répéter `data.frame$variable` et sans avoir à mettre `"variable"` entre guillemets.

Cela pose parfois problème, en particulier lorsque l'on souhaite écrire des fonctions. Dans ce cas, on peut repasser en évaluation standard (demande un vecteur character). Tous les verbes d'action ont une version standard evaluation: il s'agit du verbe suivi d'un underscore `_`. Ainsi, `select_()` est la version standard de `select()`

Sélection et recodage

Sélectionner des lignes: filter

`filter()` sélectionne des lignes à partir d'un vecteur logique (condition)

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
## The following objects are masked from 'package:stats':
##
##     filter, lag
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
load("data/ACS_artists.Rdata")
```

Sélectionner des lignes: slice

`slice()` sélectionne des lignes à partir d'un index (rang).

```
# les 10 premiers individus
slice(dt, 1:10)
```

```
## # A tibble: 10 × 8
##      sexe   age                    state income
##     <chr> <dbl>                   <fctr>  <dbl>
## 1  Female    24 (37) North Carolina/NC   7600
## 2  Female    42       (36) New York/NY  18000
## 3  Female    64         (41) Oregon/OR  72100
## 4  Female    66      (55) Wisconsin/WI  19400
## 5    Male    50        (18) Indiana/IN  60000
## 6  Female    40 (37) North Carolina/NC  24500
## 7  Female    45     (34) New Jersey/NJ  -9999
## 8    Male    52       (08) Colorado/CO  75000
```

Sélectionner des colonnes: select

`select` permet de restreindre les colonnes d'un data.frame à un sous-ensemble. On peut également les renommer directement dans select. Toutes les colonnes qui ne sont pas nommées ne sont pas sélectionnées.

```
select(dt, sexe, age, d = dipl_c)
```

```
## # A tibble: 124,023 × 3
##      sexe   age           d
##     <chr> <dbl>      <fctr>
## 1  Female    24   HS degree
## 2  Female    42 Graduate ed.
## 3  Female    64 Graduate ed.
## 4  Female    66   HS degree
## 5    Male    50     College
## 6  Female    40   HS degree
```

Sélectionner des colonnes: select

`select()` permet l'indexation négative. un signe - devant un nom de colonne la supprime du data.frame. On ne peut pas mixer indexation normale et négative : soit on choisit des colonnes, soit on en élimine.

```
select(dt, -dipl_c, -dipl, -cit)
```

```
## # A tibble: 124,023 × 5
##      sexe   age                    state income
##     <chr> <dbl>                   <fctr>  <dbl>
## 1  Female    24 (37) North Carolina/NC   7600
## 2  Female    42       (36) New York/NY  18000
## 3  Female    64         (41) Oregon/OR  72100
## 4  Female    66      (55) Wisconsin/WI  19400
## 5    Male    50        (18) Indiana/IN  60000
## 6  Female    40 (37) North Carolina/NC  24500
```

Sélectionner des colonnes: selecteurs

On atteint là la grande force de dplyr. On peut sélectionner des colonnes par leur nom avec trois fonctions qui prennent toutes trois comme premier argument une chaîne de characteres:

- `starts_with()`: les colonnes dont le nom commence par cette chaîne;
- `ends_with()`: les colonnes dont le nom se termine par cette chaîne;
- `contains()`: les colonnes dont le nom contient cette chaîne.

Cf. `?select_helpers`

Sélectionner des colonnes: selecteurs

```
# équivalent en base-r:
# dt[, grep("^dipl", names(dt))]
select(dt, starts_with("dipl"))
```

```
## # A tibble: 124,023 × 2
##                                                  dipl
##                                                <fctr>
## 1  (19) 1 or more years of college credit, no degree
## 2                                  (22) Master's degree
## 3                                  (22) Master's degree
## 4            (18) Some college, but less than 1 year
## 5                                 (21) Bachelor's degree
## 6  (19) 1 or more years of college credit, no degree
## 7                                 (21) Bachelor's degree
## 8  (19) 1 or more years of college credit, no degree
```

Sélectionner des colonnes: rename

`rename()` renvoie toutes les colonnes du data.frame, nommées ou non, mais change le nom de certaines.

```
rename(dt, genre = sexe)
```

```
## # A tibble: 124,023 × 8
##     genre   age                    state income
##     <chr> <dbl>                   <fctr>  <dbl>
## 1  Female    24 (37) North Carolina/NC   7600
## 2  Female    42       (36) New York/NY  18000
## 3  Female    64         (41) Oregon/OR  72100
## 4  Female    66      (55) Wisconsin/WI  19400
## 5    Male    50        (18) Indiana/IN  60000
## 6  Female    40 (37) North Carolina/NC  24500
## 7  Female    45     (34) New Jersey/NJ  -9999
## 8    Male    52       (08) Colorado/CO  75000
```

Transformer des colonnes: mutate

`mutate()` permet de recoder ou de créer une variable à partir des variables existantes.

```
mutate(dt, sexe = factor(sexe, levels = c("Female", "Male"),
                         labels = c("Femme", "Homme")))
```

```
## # A tibble: 124,023 × 8
##      sexe   age                  state income
##    <fctr> <dbl>                 <fctr>  <dbl>
## 1  Femme    24 (37) North Carolina/NC   7600
## 2  Femme    42       (36) New York/NY  18000
## 3  Femme    64         (41) Oregon/OR  72100
## 4  Femme    66      (55) Wisconsin/WI  19400
## 5  Homme    50        (18) Indiana/IN  60000
## 6  Femme    40 (37) North Carolina/NC  24500
## 7  Femme    45     (34) New Jersey/NJ  -9999
```

Transformer des colonnes: mutate

```
mutate(dt, age_classe = cut(age, 5))
```

```
## # A tibble: 124,023 × 9
##      sexe   age                    state income
##     <chr> <dbl>                   <fctr>  <dbl>
## 1  Female    24 (37) North Carolina/NC   7600
## 2  Female    42       (36) New York/NY  18000
## 3  Female    64         (41) Oregon/OR  72100
## 4  Female    66       (55) Wisconsin/WI  19400
## 5    Male    50         (18) Indiana/IN  60000
## 6  Female    40 (37) North Carolina/NC  24500
## 7  Female    45     (34) New Jersey/NJ  -9999
## 8    Male    52         (08) Colorado/CO  75000
## 9  Female    42           (48) Texas/TX   4000
## 10   Male    52            (39) Ohio/OH 125000
```

Transformer des colonnes: mutate_all,

C'est là que dplyr commence à devenir une alternative intéressante ; `mutate_all()`, `mutate_at()` et `mutate_if()` permettent de recoder plusieurs variables d'un coup, en employant les sélecteurs. Contrairement à lapply, la fonction à appliquer doit être incluse dans `funs()` ; dans cette fonction, le point (`.`) est employé pour appeler le vecteur originel.

```
# Tout transformer en character
mutate_all(dt, funs(as.character(.)))
```

```
## # A tibble: 124,023 × 8
##      sexe   age                    state income
##     <chr> <chr>                    <chr>  <chr>
## 1  Female    24 (37) North Carolina/NC   7600
## 2  Female    42       (36) New York/NY  18000
## 3  Female    64         (41) Oregon/OR  72100
```

Transformer des colonnes: mutate_at,

`mutate_at()` permet de sélectionner les variables à recoder par des selecteurs ; il s'agit du second argument, `.cols` qui doit être inclu dans `vars()`.

```
# Transformer dipl et dipl_c en character
mutate_at(dt, vars(starts_with("dipl")), funs(as.character(
```

```
## # A tibble: 124,023 × 8
##      sexe   age                  state income
##     <chr> <dbl>                 <fctr>  <dbl>
## 1  Female    24 (37) North Carolina/NC   7600
## 2  Female    42       (36) New York/NY  18000
## 3  Female    64         (41) Oregon/OR  72100
## 4  Female    66      (55) Wisconsin/WI  19400
## 5    Male    50        (18) Indiana/IN  60000
## 6  Female    40 (37) North Carolina/NC  24500
```

Transformer des colonnes: mutate_if,

`mutate_if()` permet de sélectionner les variables à recoder ; il s'agit du second argument, `.predicate`, qui doit être un test de condition.

```
# Transformer les factor en character
mutate_if(dt, is.factor, funs(as.character(.)))
```

```
## # A tibble: 124,023 × 8
##      sexe   age                    state income
##     <chr> <dbl>                    <chr>  <dbl>
## 1  Female    24 (37) North Carolina/NC   7600
## 2  Female    42       (36) New York/NY  18000
## 3  Female    64         (41) Oregon/OR  72100
## 4  Female    66      (55) Wisconsin/WI  19400
## 5    Male    50        (18) Indiana/IN  60000
## 6  Female    40 (37) North Carolina/NC  24500
```

Groupes

group_by

`group_by()` est l'équivalent de `tapply`. Le premier argument est toujours le data.frame que l'on exploite. Ensuite, on liste les variables de ce data.frame par lesquelles on souhaite regrouper les valeurs.

Regroupe signifie découper les individus en autant de groupe que le produit des modalités des variables indiqués. Avec une variable à k_1 modalités, k_1 groupes, avec deux variables à k_1, k_2 modalités, $k_1 * k_2$ groupes, etc.

group_by renvoie un `tbl_df` groupé, mais ne fait pas d'autres changements ; il modifie par contre le résultat des verbes suivants.

group_by et filter/slice

```
# Sélectionner l'homme et la femme
# les mieux payés
group_by(dt, sexe) %>%
  filter(income == max(income))
```

```
## Source: local data frame [2 x 8]
## Groups: sexe [2]
##
##     sexe   age                              state
##    <chr> <dbl>                             <fctr>
## 1 Female    62 (11) District of Columbia/DC
## 2   Male    80                   (36) New York/NY
## # ... with 5 more variables: income <dbl>,
## #   dipl <fctr>, cit <fctr>, eng <fctr>,
## #   dipl_c <fctr>
```

group_by et filter/slice

```
# Sélectionner l'homme et la femme
# les mieux payés dans chaque région
group_by(dt, sexe, state) %>%
  filter(income == max(income))
```

```
## Source: local data frame [106 x 8]
## Groups: sexe, state [102]
##
##      sexe   age                   state  income
##     <chr> <dbl>                  <fctr>   <dbl>
## 1    Male    52     (34) New Jersey/NJ  725000
## 2    Male    42          (23) Maine/ME  310000
## 3  Female    56       (10) Delaware/DE  110000
## 4  Female    57    (09) Connecticut/CT  635000
## 5    Male    48 (37) North Carolina/NC  506000
```

group_by et mutate

```
group_by(dt, sexe) %>%
  mutate(mean_inc = sum(income) / n(),
         ecart_inc = income - mean_inc) %>%
  select(sexe, income, ecart_inc,
         mean_inc)
```

```
## Source: local data frame [124,023 x 4]
## Groups: sexe [2]
##
##      sexe income  ecart_inc mean_inc
##     <chr>  <dbl>      <dbl>    <dbl>
## 1  Female   7600 -26178.151 33778.15
## 2  Female  18000 -15778.151 33778.15
## 3  Female  72100  38321.849 33778.15
## 4  Female  19400 -14378.151 33778.15
```

Summarize

`summarize()` permet de réduire une base de données, groupée ou non, à des indicateurs agrégés. Particulièrement utile avec `group_by()`.

```
group_by(dt, sexe) %>%
  summarise(effectif = n(),
            inc_m = mean(income),
            inc_sd = sd(income))
```

```
## # A tibble: 2 × 4
##     sexe effectif    inc_m   inc_sd
##    <chr>    <int>    <dbl>    <dbl>
## 1 Female    61185 33778.15 44218.92
## 2   Male    62838 55750.69 66508.48
```