06 - Fonction, itération et conditions

PRO1036 – Analyse des données scientifiques en R

Théo Devèze et Tim Bollé

4 Novembre 2024

Fonctions

Fonctions

Les `fonctions` sont souvent des verbes, suivi de parenthèses, contenant des arguments:

```
1 fait_ca(avec_ca)
2 fait_ca(avec_ca, et_ca, et_encore_ca)
```

Vous avez déjà utilisé des fonctions comme mutate() ou filter() qui font partie du tidyverse.

Fonctions

Il est également possible de créer vos propres fonctions lorsque vous copiez-collez le même code de manière répétée.

Elles permettent de :

- Donner un nom clair qui rend le code facile à comprendre
- Apporter les modifications à un seul endroit au lieu de changer tout le code
- Eliminer les possibles erreurs de copier-coller
- Elles sont réutilisables entre différents projets

Fonctions : 3 types

- Fonctions **vecteurs**
- Fonctions **dataframe**
- Fonctions **plot**

Fonctions vecteurs

- Une fonction vecteur prend comme argument un vecteur et donne comme résultat (*output*) un vecteur.
- Un vecteur est une liste d'objets du même type

Ex :

- Vecteur de chaines de caractères (*string vectors*)

fruits <- c("banane", “pomme", "orange")

- Vecteur de valeurs numériques

ages <- c(12, 15,77)

Quand créer une fonction vecteur

```
df %>% mutate(
  a = (a - min(a, na.rm = TRUE)) /
    (max(a, na.rm = TRUE) - min(a, na.rm = TRUE)),
  b = (b - min(a, na.rm = TRUE)) /
    (max(b, na.rm = TRUE) - min(b, na.rm = TRUE)),
  c = (c - min(c, na.rm = TRUE)) /
    (max(c, na.rm = TRUE) - min(c, na.rm = TRUE)),
  d = (d - min(d, na.rm = TRUE)) /
    (max(d, na.rm = TRUE) - min(d, na.rm = TRUE)),
```

Code très répétitif

Quand créer une fonction vecteur

```
df %>% mutate(
  a = (a - min(a, na.rm = TRUE)) /
    (max(a, na.rm = TRUE) - min(a, na.rm = TRUE)),
  b = (b - min(a, na.rm = TRUE)) /
    (max(b, na.rm = TRUE) - min(b, na.rm = TRUE)),
  c = (c - min(c, na.rm = TRUE)) /
    (max(c, na.rm = TRUE) - min(c, na.rm = TRUE)),
  d = (d - min(d, na.rm = TRUE)) /
    (max(d, na.rm = TRUE) - min(d, na.rm = TRUE)),
```

Simplifié en :

```
(  - min(  , na.rm = TRUE)) / (max(  , na.rm = TRUE) - min(  , na.rm = TRUE))
```

(x - min(x, na.rm = TRUE)) / (max(x, na.rm = TRUE) - min(x, na.rm = TRUE))

Pour transformer cette ligne en une fonction il faut :

- Un nom précis : ici on utilise rescale01
- Des arguments : les arguments sont les éléments variables entre les appels de la fonction.

Ci-dessus, il n'y a qu'un argument que nous nommons x (convention pour les vecteurs numériques).

- Le corps qui est la partie de code répétée entre chaque appels de la fonction.

On peut ensuite créer la fonction en suivant la structure ci-dessous :

```
nom <- function(arguments) {
corps
}
```

```
( - min(, na.rm = TRUE)) / (max(, na.rm = TRUE) - min(, na.rm = TRUE))
```

Dans ce cas, cela nous donne :

```
rescale01 <- function(x) {
  (x - min(x, na.rm = TRUE)) / (max(x, na.rm = TRUE) - min(x, na.rm = TRUE))
}
```

Nous avons donc créé la function rescale01 que nous pourrons appeler à chaque fois que l'on en aura besoin.

Nous pouvons tester notre fonction avec un vecteur numérique comme argument :

```
rescale01(c(-10, 0, 10))
#> [1] 0.0 0.5 1.0

rescale01(c(1, 2, 3, NA, 5))
#> [1] 0.00 0.25 0.50   NA 1.00
```

Le code précédent devient :

```
df %>% mutate(
  a = (a - min(a, na.rm = TRUE)) /
 (max(a, na.rm = TRUE) - min(a, na.rm = TRUE)),
  b = (b - min(a, na.rm = TRUE)) /
  (max(b, na.rm = TRUE) - min(b, na.rm = TRUE)),
  c = (c - min(c, na.rm = TRUE)) /
  (max(c, na.rm = TRUE) - min(c, na.rm = TRUE)),
  d = (d - min(d, na.rm = TRUE)) /
  (max(d, na.rm = TRUE) - min(d, na.rm = TRUE)),
```

```
df %>% mutate(
  a = rescale01(a),
  b = rescale01(b),
  c = rescale01(c),
  d = rescale01(d),
)
```

Cependant il semble encore répétitif…

Intérêt pour modifier une fonction

```
rescale01 <- function(x) {
  (x - min(x, na.rm = TRUE)) / (max(x, na.rm = TRUE) - min(x, na.rm = TRUE))
}
```

On peut vouloir optimiser notre fonction rescale01 en lui faisant utiliser range() pour calculer maximum et minimum en une fois au lieu de max() et min() :

```
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}
```

On peut également tester comment la fonction gère des valeurs infinies :

```
rescale01(c(1:10, Inf))
#>  [1]   0   0   0   0   0   0   0   0   0   0 NaN
```

Le problème provient de la division par l'infini : on peut spécifier à range() d'ignorer les valeurs infinies :

```
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE, finite = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}

rescale01(c(1:10, Inf))
#>  [1] 0.0000000 0.1111111 0.2222222 0.3333333 0.4444444 0.5555556 0.6666667
#>  [8] 0.7777778 0.8888889 1.0000000       Inf
```

Ces changements illustrent la simplicité de modifier une fonction à un seul endroit plutôt que dans tout le code.

Fonction visant à modifier un vecteur mutate()

Ici, case_when() est utilisée pour assurer que toutes les valeurs d'un vecteur sont comprises entre un minimum et un maximum.

```
clamp <- function(x, min, max) {
  case_when(
    x < min ~ NA,
    x > max ~ NA,
    .default = x
  )
}
clamp(1:10, min = 3, max = 7)
#>  [1] NA NA 3 4 5 6 7 NA NA NA
```

Fonction visant à modifier un vecteur mutate()

Evidemment, les fonctions peuvent également fonctionner sur des vecteurs de chaines de caractères (*string vector*).

Ici, la fonction s'assure que les caractères d'un vecteur commencent par une majuscule : str_sub prend le caractère en position 1 et str_to_upper transforme ce caractère en majuscule (*upper case*)

```
first_upper <- function(x) {
  str_sub(x, 1, 1) <- str_to_upper(str_sub(x, 1, 1))
  x
}
first_upper("hello")
#> [1] "Hello"
```

Fonction visant à effectuer des sommaires summary()

Cette fonction calcule directement le coefficient de variation, qui divise l'écart-type par la moyenne :

```
cv <- function(x, na.rm = FALSE) {
  sd(x, na.rm = na.rm) / mean(x, na.rm = na.rm)
}

cv(runif(100, min = 0, max = 50))
#> [1] 0.5196276
cv(runif(100, min = 0, max = 500))
#> [1] 0.5652554
```

Fonctions dataframe

- Une fonction dataframe prend comme argument un tableau (*dataframe*) et donne comme résultat (*output*) un dataframe ou un vecteur. Elle fonctionne ainsi comme un verbe dplyr.

- Elle peut être pertinente dans le cas où l'on copie trop régulièrement des fonctions vecteur pour travailler sur des dataframes.

Particularité de fonctionnement des fonctions dataframe

- Lorsque l'on commence à écrire des fonctions avec des verbes dplyr, on rencontre vite le problème de l'indirection.
- Pour illustrer ce problème, on utilise une fonction simple : grouped_mean(). L'objectif de cette fonction est de calculer la moyenne de mean_var groupée par group_var :

```
grouped_mean <- function(df, group_var, mean_var) {
 df %>%
  group_by(group_var) %>%
  summarize(mean(mean_var))
}
```

On obtient cette erreur :

```
diamonds %>% grouped_mean(cut, carat)
#> Error in `group_by()`:
#> ! Must group by variables found in `.data`.
#> ✖ Column `group_var` is not found.
```

Pour y voir plus clair on regarde avec des données

```
grouped_mean <- function(df, group_var,
mean_var) {
        df %>%
         group_by(group_var) %>%
         summarize(mean(mean_var))
       }
```

	mean_var	group_var	group	x	y
1	1	g	1	10	100

```
df %>% grouped_mean(group, x)
#> # A tibble: 1 × 2
#>   group_var `mean(mean_var)`
#>   <chr>              <dbl>
#>    g                  1
```

```
df %>% grouped_mean(group, y)
#> # A tibble: 1 × 2
#>   group_var `mean(mean_var)`
#>   <chr>              <dbl>
#>    g                  1
```

Il cherche group_var et mean_var comme des noms de variables (inexistants dans un vrai jeu de données) et pas comme des colonnes dans lesquelles aller chercher les données.

Solution { }

- Pour faire fonctionner grouped_mean(), il faut encadrer group_var et mean_var de {{braquettes}} :

```
grouped_mean <- function(df, group_var,
mean_var) {
        df %>%
         group_by(group_var) %>%
         summarize(mean(mean_var))
        }
```

```
grouped_mean <- function(df, group_var, mean_var) {
  df %>%
    group_by({{ group_var }}) |>
    summarize(mean({{ mean_var }}))
}
```

```
df |> grouped_mean(group, x)
#> # A tibble: 1 × 2
#>   group `mean(x)`
#>   <dbl>     <dbl>
#>     1        10
```

- Difficulté a savoir quel élément mettre entre braquettes : lire la documentation

Fonctions plot

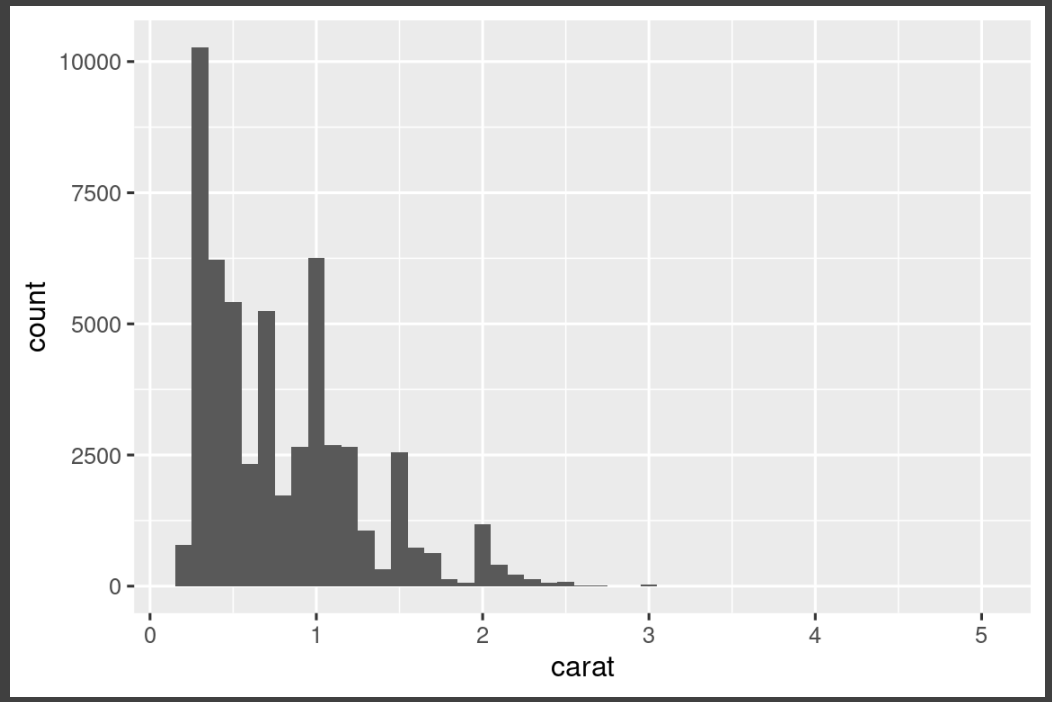
- Une fonction plot prend comme argument un tableau (*dataframe*) ou un vecteur et donne comme résultat (*output*) un plot.

- Elle peut être pertinente dans le cas où l'on réalise régulièrement le même type de graphiques.

Fonctions plot

Exemple pour automatiser des histogrames :

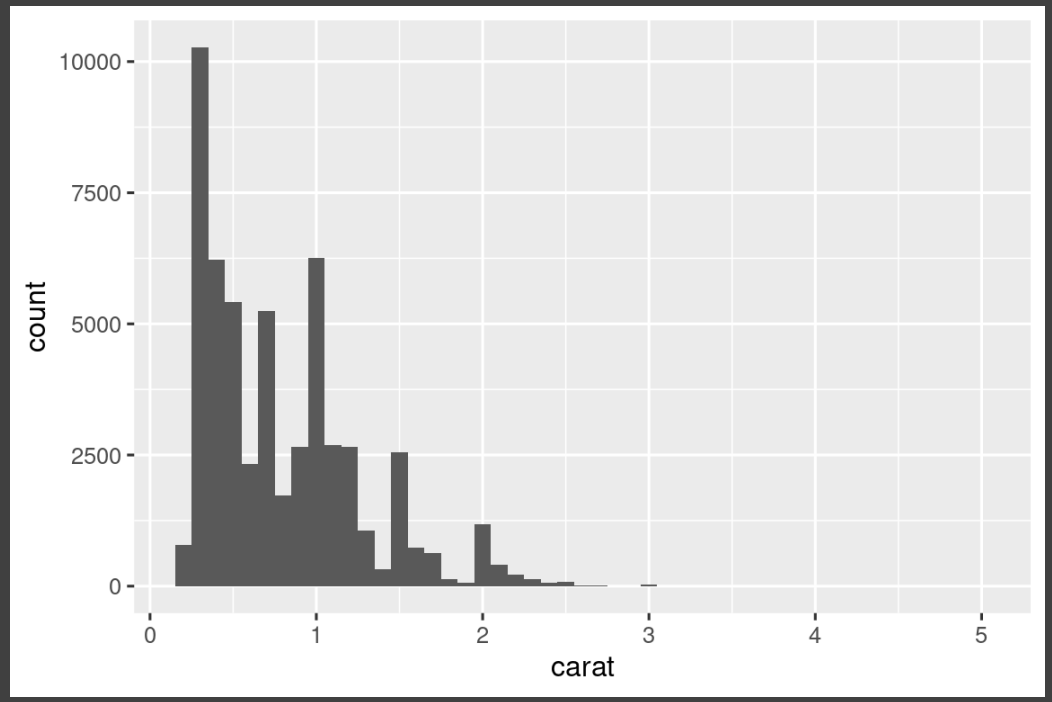
```
diamonds %>%
  ggplot(aes(x = carat)) +
  geom_histogram(binwidth = 0.1)

diamonds %>%
  ggplot(aes(x = carat)) +
  geom_histogram(binwidth = 0.05)
```



Fonctions plot

Penser à mettre aes {{entre braquettes}} comme pour les fonctions dataframes.

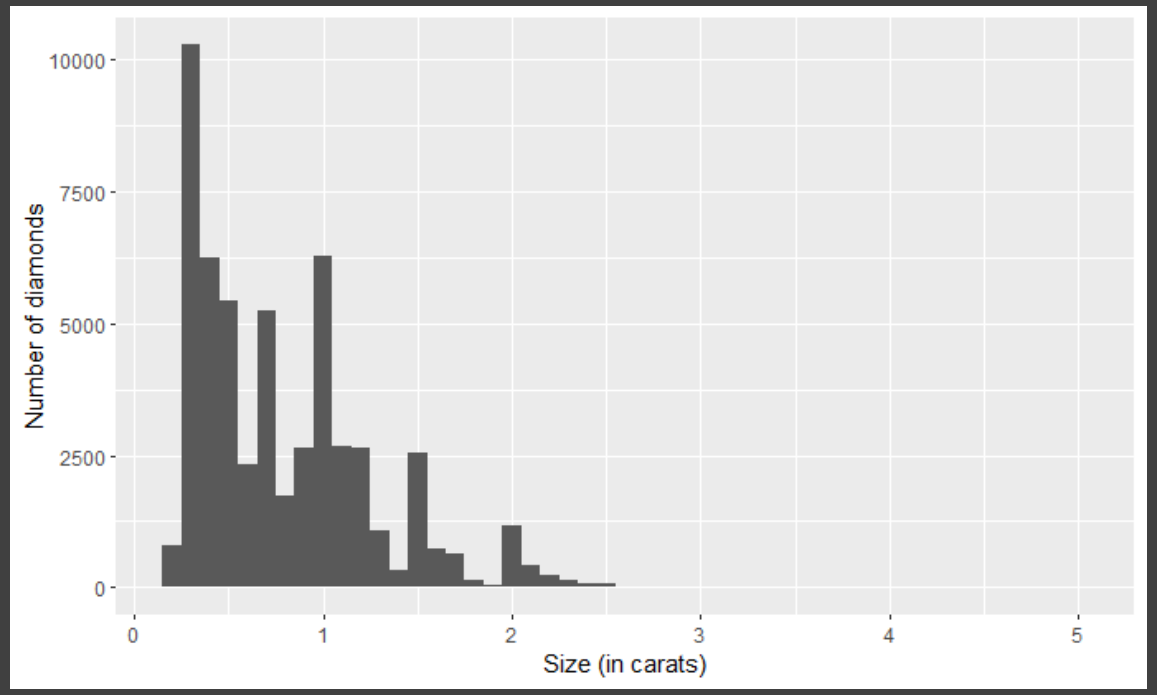
```
histogram <- function(df, var, binwidth = NULL) {
  df %>%
    ggplot(aes(x = {{ var }})) +
    geom_histogram(binwidth = binwidth)
}

diamonds %>%
 histogram(carat, 0.1)
```



Fonctions plot

- Comme il s'agit d'un ggplot : possibilité de rajouter des couches avec un +

```
diamonds %>%
  histogram(carat, 0.1) +
  labs(x = "Size (in carats)", y = "Number of diamonds")
```



Ajouter des variables

Si on reprend notre function :

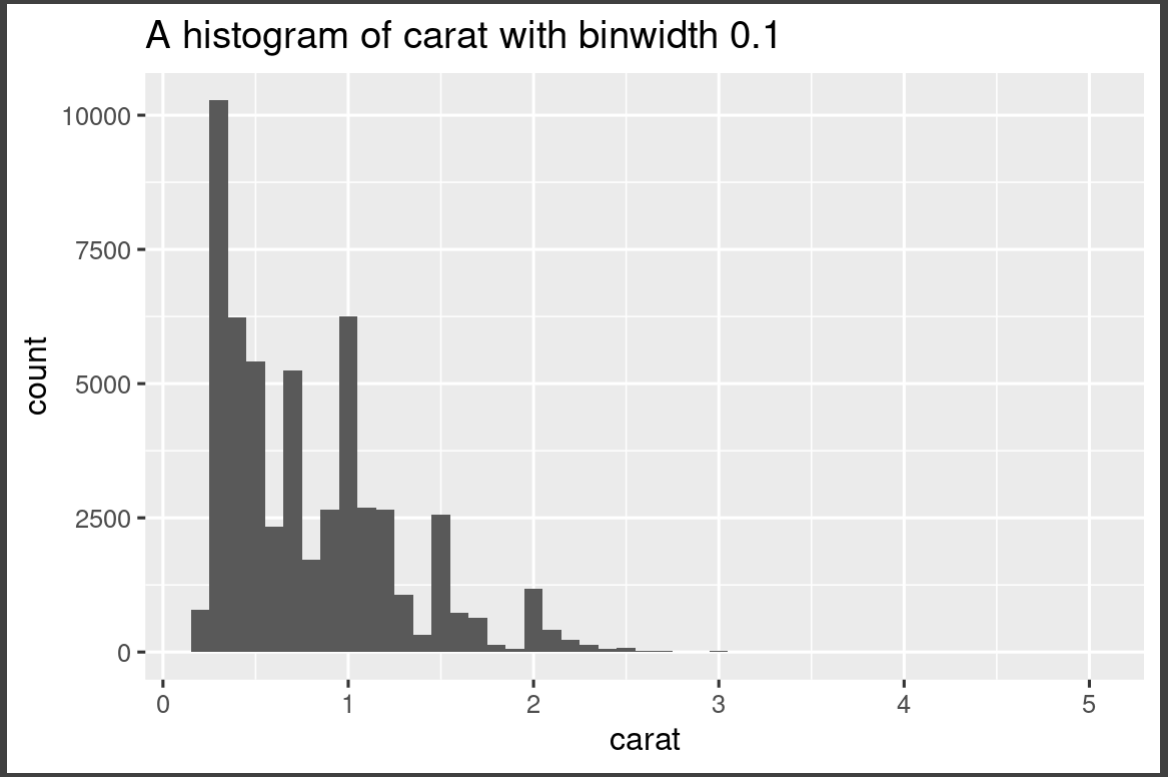
```
histogram <- function(df, var, binwidth = NULL) {
  df |>
    ggplot(aes(x = {{ var }})) +
    geom_histogram(binwidth = binwidth)
}
```

- Si on veut ajouter la largeur des colonnes utilisée dans le titre de l'output nous allons devoir utiliser une function du package rlang : rlang::englue().
- Fonctionne de manière similaire à str_glue(), les variables {entre braquettes} seront lues et insérées dans la chaine de caractères :

- Si on veut ajouter la largeur des colonnes utilisée dans le titre de l'output nous allons devoir utiliser une function du package rlang : rlang::englue().
- Fonctionne de manière similaire à str_glue(), les variables {entre braquettes} seront lues et insérées dans la chaine de caractères :

```
histogram <- function(df, var, binwidth) {
  label <- rlang::englue("A histogram of {{var}} with binwidth
{binwidth}")
  df %>%
    ggplot(aes(x = {{ var }})) +
    geom_histogram(binwidth = binwidth) +
    labs(title = label)
}

diamonds %>% histogram(carat, 0.1)
```

On peut utiliser la même approche pour ajouter une chaine de caractère issue d'une variable à n'importe quel endroit dans un ggplot.



Conclusion et bonnes pratiques

Pour terminer cette partie, retenez ces bonnes pratiques pour nomme vos fonctions :

- Les fonctions doivent être nommés avec des verbes compréhensibles qui décrivent leur action

f() = trop court et non-descriptif

my_awesome_function() = pas un verbe et non-descriptif

impute_missing()
collapse_years() = clair et évocateur

Itérations

Besoin d'itérations

Si on voulait simplement compter le nombre de valeurs et calculer la médiane pour chaque colonne de ce tableau :

```
df <- tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
```

	a	b	c	d
1	1.2965865	-1.49476484	0.76378928	0.1679497
2	1.4913157	0.45889206	-1.23215975	2.4243161
3	-1.2218145	-0.30583207	1.51224205	-1.1256529
4	-1.6894459	0.87123772	-0.08638624	0.5235893
5	-0.3298861	-0.56977245	1.52227574	-0.6636826
6	-1.2779200	-1.13954023	0.81013779	1.5878655
7	0.9641614	-1.15009024	1.49020054	-0.4275872
8	-0.5456151	-0.05049461	-0.15076641	1.2236100
9	2.2322583	0.36353693	2.14747288	-0.7354068
10	0.5557191	-1.41745140	-1.07698259	-1.2910551

Quel serait le problème ?

On pourrait faire comme cela :

```
df %>% summarize(
  n = n(),
  a = median(a),
  b = median(b),
  c = median(c),
  d = median(d),
)
```

```
#> # A tibble: 1 × 5
#>       n     a     b      c     d
#>   <int> <dbl> <dbl>  <dbl> <dbl>
#> 1    10 -0.246 -0.287 -0.0567 0.144
```

Besoin d'itérations

- On ne veut pas avoir à copier-coller le même code ET ce serait infaisable si nous avions des centaines de colonnes.
- Il est pertinent dans ce cas d'utiliser across():

```
df %>% summarize(
  n = n(),
  across(a:d, median),
)
```

```
#> # A tibble: 1 × 5
#>       n     a      b      c     d
#>   <int> <dbl>  <dbl>  <dbl> <dbl>
#> 1    10 -0.246 -0.287 -0.0567 0.144
```

Arguments de across() :

across(colonnes_a_traiter, function_a_effectuer, noms_des_colonnes output)

Sélectionner des colonnes multiples avec across()

Arguments de across() :

across(colonnes_a_traiter, function_a_effectuer, noms_des_colonnes output)

- Il est possible d'utiliser les fonctions starts_with() and ends_with() pour sélectionner des colonnes sur leurs noms.

- Il y a deux techniques additionnelles particulièrement utiles avec across(): everything() et where().

	grp	a	b	c	d
1	2	-0.4059170	-0.7714471	-0.93698367	-0.72080785
2	1	0.5298969	1.8281175	0.12169661	-0.20639934
3	2	-0.3402562	-0.5123042	2.04960298	-0.21257297
4	2	0.6540518	-0.7064512	0.25096472	0.45084845
5	1	0.2616810	-0.3017467	1.10576969	-0.14867660
6	2	0.2761993	-0.1910638	-0.94477472	-2.05080784
7	1	0.1997515	-1.1948940	0.58169231	-0.03201269
8	1	-0.1128377	-1.3590989	0.51194022	-1.55288109
9	1	-0.7932780	0.2856397	-0.09897756	-0.37340517
10	1	-1.3929185	1.2028360	-0.52177788	0.77241182

Sélectionner des colonnes multiples avec across()

everything() est assez equivoque : elle sélectionne les colonnes qui ne sont pas utilisées dans group_by() :

```
df %>%
  group_by(grp) %>%
  summarize(across(everything(), median))
```

```
#> # A tibble: 2 × 5
#>     grp      a      b     c     d
#>   <int>  <dbl>  <dbl> <dbl> <dbl>
#> 1     1 -0.0935 -0.0163 0.363 0.364
#> 2     2  0.312  -0.0576 0.208 0.565
```

	grp	a	b	c	d
1	2	-0.4059170	-0.7714471	-0.93698367	-0.72080785
2	1	0.5298969	1.8281175	0.12169661	-0.20639934
3	2	-0.3402562	-0.5123042	2.04960298	-0.21257297
4	2	0.6540518	-0.7064512	0.25096472	0.45084845
5	1	0.2616810	-0.3017467	1.10576969	-0.14867660
6	2	0.2761993	-0.1910638	-0.94477472	-2.05080784
7	1	0.1997515	-1.1948940	0.58169231	-0.03201269
8	1	-0.1128377	-1.3590989	0.51194022	-1.55288109
9	1	-0.7932780	0.2856397	-0.09897756	-0.37340517
10	1	-1.3929185	1.2028360	-0.52177788	0.77241182

Sélectionner des colonnes multiples avec across()

where() permet de sélectionner des colonnes en fonction de leur type :

- where(is.numeric) sélectionne les colonnes numériques.
- where(is.character) sélectionne les colonnes chaines de caractères.
- where(is.Date) sélectionne les colonnes dates.
- where(is.POSIXct) sélectionne les colonnes *date-time*.
- where(is.logical) sélectionne les colonnes logiques.

Comme tous les sélecteurs, on peut combiner where() avec l'algèbre booléenne :
!where(is.numeric) sélectionne les colonnes non-numériques,
starts_with("a") & where(is.logical) sélectionne les colonnes logiques don le nom commence par "a".

Appeler une seule fonction

Arguments de across() :

across(colonnes_a_traiter, function_a_effectuer, noms_des_colonnes output)

- Le second argument de across() définies comment les colonnes sélectionnées seront transformées. Dans la majortié des cas, il s'agira d'une fonction éxistante (median, mean, str_flatten, …) que l'on passe à la fonction across().

- Une particularité ici est que l'on passe la fonction choisie à across() il ne faut donc pas mettre de parenthèse à celle-ci :

across(a:d, median) et pas across(a:d, median())

Illustration de l’erreur

```
df %>%
  group_by(grp) %>%
  summarize(across(everything(), median()))
#> Error in `summarize()`:
#> ℹ In argument: `across(everything(), median())`.
#> Caused by error in `median.default()`:
#> ! argument "x" is missing, with no default
```

C’est la même erreur que si on appelle une fonction sans argument :

```
median()
#> Error in median.default(): argument "x" is missing, with no default
```

Utiliser plusieurs arguments

On peut utiliser function() dans across() !

Ici on rajoute l'agument pour retirer les NA :

```
df_miss %>%
  summarize(
    across(a:d, function(x) median(x, na.rm = TRUE)),
    n = n()
  )
```

```
df_miss %>%
  summarize(
    across(a:d, \(x) median(x, na.rm = TRUE)),
    n = n()
  )
```

```
#> # A tibble: 1 × 5
#>       a     b      c     d     n
#>   <dbl> <dbl>  <dbl> <dbl> <int>
#> 1 0.139 -1.11 -0.387  1.15     5
```

NB : On préfèrera l'écrire comme ci-dessus pour éviter de surcharger le code

Appeler plusieurs fonctions

Ici on aimerait savoir combien de NAs ont été retirés : pour cela on va utiliser plusieurs fonctions dans across() :

Il est possible de passer plusieurs fonctions à across via une liste list(fonctionA, fonctionB).

```
df_miss %>%
  summarize(
    across(a:d, list(
      median = \(x) median(x, na.rm = TRUE),
      n_miss = \(x) sum(is.na(x))
    )),
    n = n()
  )
```

```
#> # A tibble: 1 × 9
#>   a_median a_n_miss b_median b_n_miss c_median
c_n_miss d_median d_n_miss
#>      <dbl>    <int>    <dbl>    <int>    <dbl>
<int>
#> 1    0.139        1    -1.11        1   -0.387
#> # ℹ 1 more variable: n <int>
```

Spécifier les noms de colonnes

- Le résultat de across() est nommé selon les spécifications en arguments :

```
df_miss %>%
  summarize(
    across(
      a:d,
      list(
        median = \(x) median(x, na.rm = TRUE),
        n_miss = \(x) sum(is.na(x))
      ),
      .names = "{.fn}_{.col}"
    ),
    n = n(),
  )
```

```
#> # A tibble: 1 × 9
#>   median_a n_miss_a median_b n_miss_b median_c
n_miss_c median_d n_miss_d
#>      <dbl>    <int>    <dbl>    <int>    <dbl>    <int>    <dbl>
<int>
#> 1    0.139        1    -1.11        1   -0.387        2    1.15        0
#> # ℹ 1 more variable: n <int>
```

Spécifier les noms des colonnes quand on utilise across() avec mutate()

- Par défaut, across() nomme le résultat comme les colonnes utilisée, ce qui remplacera les colonnes existantes par le résultat. Pour éviter cela, il faut specifier le nom des colonnes du résultat :

```
df_miss %>%
  mutate(
    across(a:d, \(x) coalesce(x, 0))
  )
```

```
#> # A tibble: 5 × 4
#>       a      b      c     d
#>   <dbl>  <dbl>  <dbl> <dbl>
#> 1  0.434 -1.25   0     1.60
#> 2  0     -1.43  -0.297 0.776
#> 3 -0.156 -0.980  0     1.15
#> 4 -2.61  -0.683 -0.785 2.13
#> 5  1.11   0     -0.387 0.704
```

- coalesce() retourne les valeurs non-manquantes et transforme les NA en 0

Spécifier les noms des colonnes quand on utilise across() avec mutate()

- Si au lieu de ça on voulait créer de nouvelles colonnes :

```
df_miss %>%
  mutate(
    across(a:d, \(x) coalesce(x, 0), .names = "{.col}_na_zero")
  )
```

```
#> # A tibble: 5 × 8
#>       a      b      c     d a_na_zero b_na_zero c_na_zero
d_na_zero
#>   <dbl>  <dbl>  <dbl> <dbl>    <dbl>    <dbl>    <dbl>
<dbl>
#> 1  0.434 -1.25  NA     1.60     0.434    -1.25     0        1.60
#> 2 NA     -1.43  -0.297 0.776     0       -1.43     -0.297     0.776
#> 3 -0.156 -0.980 NA     1.15     -0.156    -0.980     0        1.15
#> 4 -2.61  -0.683 -0.785 2.13     -2.61     -0.683    -0.785     2.13
#> 5  1.11  NA     -0.387 0.704     1.11      0       -0.387     0.704
```

Particularité de across() avec filter()

across() fonctionne bien avec summarize() et mutate() mais il est moins efficace avec filter() car il faut multiplier les conditions avec | ou &.

Une manière plus efficace est d'utiliser les alternatives à across() : **if_any** et **if_all**.

```
df_miss %>% filter(is.na(a) | is.na(b) | is.na(c) | is.na(d))
```

devient :

```
df_miss %>% filter(if_any(a:d, is.na))
```

```
df_miss %>% filter(is.na(a) & is.na(b) & is.na(c) & is.na(d))
```

devient :

```
df_miss %>% filter(if_all(a:d, is.na))
```

Les opérations logiques

Opérateur	Définition	Opérateur	Définition
`<`	plus petit que	`x | y`	`x` OU `y`
`<=`	plus petit ou égal	`is.na(x)`	test si `x` est `NA`
`>`	plus grand que	`!is.na(x)`	test si `x` est différent de `NA`
`>=`	plus grand ou égal	`x %in% y`	test si `x` est dans `y`
`==`	égal à	`!(x %in% y)`	test si `x` n'est pas dans `y`
`!=`	différent de	`!x`	non `x`
`x & y`	`x` ET `y`		

Conditions

Rappel Cours 3

- starts_with(): Commence par un préfixe
- ends_with(): Termine par un suffixe
- contains(): Contient une certaine chaine de caractères
- num_range(): Match un certain range de nombres
- one_of(): Match les variables font parties d'une liste
- everything(): Match les variable qui contiennent tous les éléments de la liste
- last_col(): Sélectionne la dernière variable (possibilité d'indiquer un offset)
- matches(): Match une expression régulière

La fonction if_else()

- Dplyr nous offre la fonction if_else() qui permet de placer une condition qui peut être **VRAIE ou FAUSSE (TRUE ou FALSE)**. The first argument, condition, is a logical vector, the second, true, gives the output when the condition is true, and the third, false, gives the output if the condition is false.

- Commençons avec un exemple simple qui indique si un vecteur est “+ve” (positif) or “-ve” (negatif) :

```
x <- c(-3:3, NA)
if_else(x > 0, "+ve", "-ve")
#> [1] "-ve" "-ve" "-ve" "-ve" "+ve" "+ve" "+ve" NA
```

Il y a un quatrième argument optionnel qui indique que faire des NA :

```
if_else(x > 0, "+ve", "-ve", "???")
#> [1] "-ve" "-ve" "-ve" "-ve" "+ve" "+ve" "+ve" "???"
```

La fonction if_else()

- On peut également utiliser des vecteurs pour la condition VRAIE ou FAUSSE.

Ex : ici on recrée abs() qui retourne la valeur absolue d'un vecteur numérique :

```
if_else(x < 0, -x, x)
#> [1]  3  2  1  0  1  2  3 NA
```

- On peut ajouter une condition à notre fonction if_else(x > 0, "+ve", "-ve") qui prend en compte si x = 0.

```
if_else(x == 0, "0", if_else(x < 0, "-ve", "+ve"), "???")
#> [1] "-ve" "-ve" "-ve" "0"   "+ve" "+ve" "+ve" "???"
```

- Comme on ajoute des conditions, on doit multiplier les if_else et il est clair que plus on en ajoute, moins c'est lisible : il est mieux d'utiliser case_when().

La fonction case_when()

```r
cat_lovers <- cat_lovers %>%
  mutate(
    number_of_cats = case_when(
      name == "Ginger Clark" ~ "2",
      name == "Doug Bass"    ~ "3",
      TRUE                   ~ number_of_cats
      ),
    number_of_cats = as.numeric(number_of_cats)
    )
```

Utilisée lorsqu'on applique une liste de conditions et des transformations différentes selon celles-ci sur un vecteur.

La fonction case_when()

Sa synthaxe est spéciale et nous allons la décortiquer maintenant :

- Elle fonctionne avec des paires condition ~ output :
- condition doit être un vecteur logique ; quand elle est VRAIE, on applique l'output.

- Voici ce que serait le remplacement de notre précédent if_else() :

```
x <- c(-3:3, NA)
case_when(
  x == 0   ~ "0",
  x < 0    ~ "-ve",
  x > 0    ~ "+ve",
  is.na(x) ~ "???"
)
```

```
#> [1] "-ve" "-ve" "-ve" "0"   "+ve" "+ve" "+ve" "???"
```

- Etudions case_when() dans un cas plus simple : si aucune des conditions n'est rencontrée, case_when() nous retourne NA.

```
case_when(
  x < 0 ~ "-ve",
  x > 0 ~ "+ve"
)
#> [1] "-ve" "-ve" "-ve" NA    "+ve" "+ve" "+ve" NA
```

- On utilise l'argument .default pour créer une option différente de NA si aucune condition n'est rencontrée :

```
case_when(
  x < 0 ~ "-ve",
  x > 0 ~ "+ve",
  .default = "???"
)
#> [1] "-ve" "-ve" "-ve" "???" "+ve" "+ve" "+ve" "???"
```

La fonction case_when()

- Comportement de case_when() lorsque plusieurs conditions sont rencontrées :

```
case_when(
  x > 0 ~ "+ve",
  x > 2 ~ "big"
)
#> [1] NA    NA    NA    NA    "+ve" "+ve" "+ve" NA
```

- La condition qui est rencontrée en première dans case_when() **prévaut sur les suivantes**.

Sources

- Wickham, H., Çetinkaya-Rundel, M. and Grolemund, G. (2023). *R for Data Science* (2nd ed.). O'Reilly Media, Inc.