Notes de leçon | Transformation conditionnelle

February 2024

Introduction

Dans la dernière leçon, vous avez appris les bases de la transformation des données en utilisant la fonction `mutate()` de {dplyr}.

Dans cette leçon, nous avons principalement examiné les transformations *globales* ; c'est-à-dire, des transformations qui font la même chose à une variable entière. Dans cette leçon, nous allons voir comment manipuler *conditionnellement* certaines lignes en fonction de si elles répondent ou non à des critères définis.

Pour cela, nous utiliserons principalement la fonction `case_when()`, que vous considérerez probablement comme l'une des fonctions les plus importantes de {dplyr} pour les tâches de préparation des données.

Commençons.

Fig: les conditions case_when().

Objectifs d'apprentissage

1. Vous pouvez transformer ou créer de nouvelles variables en fonction des conditions en utilisant `dplyr::case_when()`
2. Vous savez comment utiliser la condition `TRUE` dans `case_when()` pour faire correspondre les cas non appariés.
3. Vous pouvez gérer les valeurs `NA` dans les transformations `case_when()`.
4. Vous comprenez comment conserver les valeurs par défaut d'une variable dans une formule `case_when()`.
5. Vous pouvez écrire des conditions `case_when()` impliquant plusieurs comparateurs et plusieurs variables.
6. Vous comprenez l'ordre de priorité des conditions `case_when()`.
7. Vous pouvez utiliser `dplyr::if_else()` pour l'assignation conditionnelle binaire.

Packages

Cette leçon nécessitera la suite de package tidyverse :

```
ire(pacman)) install.packages("pacman")
p_load(tidyverse)
```

Jeux de données

Dans cette leçon, nous utiliserons à nouveau les données de l'enquête sérologique COVID-19 menée à Yaoundé, au Cameroun.

```
ter et visualiser le jeu de données
 <- read_csv(here::here('data/fr_yaounde_data.csv')) %>%
ndre manquant chaque 5ème âge
e(age = case_when(row_number() %in% seq(5, 900, by = 5) ~ NA_real_,
                  TRUE ~ age)) %>%
ommer la variable âge
e(age_annees = age) %>%
primer la colonne de catégorie d'âge
t(-cat_age)
```

Notez que dans le bloc de code ci-dessus, nous avons légèrement modifié la colonne d'âge, en introduisant artificiellement quelques valeurs manquantes, et nous avons également supprimé la colonne `cat_age`. Ceci pour aider à illustrer certains points clés du tutoriel.

Pour les questions de pratique, nous utiliserons également une liste d'une épidémie de 136 cas d'influenza A H7N9 lors d'une épidémie en 2013 en Chine. Il s'agit d'une version modifiée d'un ensemble de données compilé par Kucharski et al. (2014).

```
ter et afficher l'ensemble de données
nflu <- read_csv(here::here('data/fr_flu_h7n9_china_2013.csv'))
nflu
```

Rappel : opérateurs relationnels (comparateurs) sur R

Tout au long de cette leçon, vous utiliserez beaucoup d'opérateurs relationnels en R. Rappelons que les opérateurs relationnels, parfois appelés "comparateurs", testent la relation entre deux valeurs, et renvoient `TRUE`, `FALSE` ou `NA`.

Une liste des opérateurs les plus couramment utilisés est donnée ci-dessous :

Opérateur	est VRAI si
A < B	A est **inférieur à** B
A <= B	A est **inférieur ou égal à** B
A > B	A est **supérieur à** B
A >= B	A est **supérieur ou égal à** B
A == B	A est **égal à** B
A != B	A est **différent de** B
A %in% B	A **est un élément de** B

Introduction à `case_when()`

Pour se familiariser avec `case_when()`, commençons par une simple transformation conditionnelle sur la colonne `age_annees` de l'ensemble de données `yaounde`. Nous commençons par extraire uniquement la colonne `age_annees` du jeu de données pour illustrer facilement :

```
_age <-
le %>%
(age_annees)

_age
```

Maintenant, en utilisant `case_when()`, nous pouvons créer une nouvelle colonne, appelée "age_groupe", qui a la valeur "Enfant" si la personne a moins de 18 ans, et "Adulte" si la personne a 18 ans et plus :

```
_age %>%
(age_groupe = case_when(age_annees < 18 ~ "Enfant",
                        age_annees >= 18 ~ "Adulte"))
```

La syntaxe de `case_when()` peut sembler un peu étrangère, mais elle est assez simple : du côté gauche (LHS) du signe `~` (appelé "tilde"), vous fournissez la ou les conditions que vous voulez évaluer, et du côté droit (RHS), vous fournissez une valeur à insérer si la condition est vraie.

Donc, la déclaration `case_when(age_annees < 18 ~ "Enfant", age_annees >= 18 ~ "Adulte")` peut se lire comme suit : "si `age_annees` est inférieur à 18, insérez 'Enfant', sinon si `age_annees` est supérieur ou égal à 18, insérez 'Adulte'".

Formules, LHS et RHS

Chaque ligne d'un appel à `case_when()` est appelée une "formule" ou, parfois, une "formule à deux côtés". Et chaque formule a un côté gauche (LHS) et un côté droit (RHS).

Par exemple, le code `age_annees < 18 ~ "Enfant"` est une "formule", son LHS est `age_annees < 18` tandis que son RHS est `"Enfant"`.

Vous allez probablement rencontrer ces termes en lisant la documentation pour la fonction `case_when()`, et nous les utiliserons également dans cette leçon.

Après avoir créé une nouvelle variable avec `case_when()`, il est recommandé de l'inspecter minutieusement pour s'assurer qu'elle a fonctionné comme prévu.

Pour inspecter la variable, vous pouvez passer votre jeu de données dans la fonction `View()` pour la visualiser sous forme de tableau :

```
_age %>%
e(age_groupe = case_when(age_annees < 18 ~ "Enfant",
                         age_annees >= 18 ~ "Adulte")) %>%
```

Cela ouvrirait un nouvel onglet dans RStudio où vous devriez manuellement scanner la nouvelle colonne, `age_groupe` et la colonne référencée `age_annees` pour vous assurer que votre déclaration `case_when()` a fait ce que vous vouliez qu'elle fasse.

Vous pourriez aussi passer la nouvelle colonne dans la fonction `tabyl()` pour vous assurer que les proportions "ont du sens" :

```
_age %>%
e(age_groupe = case_when(age_annees < 18 ~ "Enfant",
                         age_annees >= 18 ~ "Adulte")) %>%
(age_groupe)
```

Avec les données `liste_influ`, créez une nouvelle colonne, appelée `age_groupe`, qui a la valeur "Moins de 50" pour les personnes de moins de 50 ans et "50 et plus" pour les personnes âgées de 50 ans et plus. Utilisez la fonction `case_when()`.

```
étez le code avec votre réponse :

roupe <- liste_influ %>%
e(age_groupe = _____________________)
```

Parmi l'ensemble des individus dans le jeu de données `liste_influ`, quel pourcentage est confirmé comme étant inférieur à 60 ans ? (Répétez la procédure ci-dessus mais avec le seuil de `60`, puis appelez `tabyl()` sur la variable de groupe d'âge. Utilisez la colonne `percent`, pas la colonne `valid_percent`).

```
 votre réponse sous forme d'un nombre ENTIER sans guillemets :

roupe_pourcentage(_____________________________________)
```

L'argument par défaut `TRUE`

Dans une déclaration `case_when()`, vous pouvez utiliser une condition littérale `TRUE` pour faire correspondre toutes les lignes non encore appariées avec les conditions fournies.

Par exemple, si nous ne gardons que la première condition de l'exemple précédent, `age_annees < 18`, et définissons la valeur par défaut à `TRUE ~ "Pas enfant"`, alors tous les adultes et les valeurs `NA` dans l'ensemble de données seront étiquetés `"Pas enfant"` par défaut.

```
_age %>%
e(age_groupe = case_when(age_annees < 18 ~ "Enfant",
                         TRUE ~ "Pas enfant"))
```

Cette condition `TRUE` peut être lue comme "pour tout le reste…".

Ainsi, la déclaration `case_when()` utilisée précédemment, `age_annees < 18 ~ "Enfant", TRUE ~ "Pas enfant"`, se lirait alors comme suit : "si l'âge est inférieur à 18, entrez 'Enfant' et *pour tout le monde qui n'a pas encore été apparié*, entrez 'Pas enfant'".

Il est important d'utiliser `TRUE` comme condition *finale* dans `case_when()`. Si vous l'utilisez comme première condition, elle aura la priorité sur toutes les autres, comme on peut le voir ici :

```
_age %>%
e(age_groupe = case_when(TRUE ~ "Pas enfant",
                         age_annees < 18 ~ "Enfant"))
```

Comme vous pouvez le constater, tous les individus sont maintenant codés comme "Pas enfant", parce que la condition `TRUE` a été placée en premier, et a donc pris le pas. Nous examinerons plus loin la question de la priorité.

Appariement des NA avec `is.na()`

Nous pouvons appairer les valeurs manquantes manuellement avec `is.na()`. Ci-dessous, nous appairons les âges `NA` avec `is.na()` et définissons leur groupe d'âge à "Age manquant" :

```
_age %>%
e(age_groupe = case_when(age_annees < 18 ~ "Enfant",
                         age_annees >= 18 ~ "Adulte",
                         is.na(age_annees) ~ "Age manquant"))
```

Comme précédemment, en utilisant les données `liste_influ`, créez une nouvelle colonne, appelée `age_groupe`, qui a la valeur “Moins de 60” pour les personnes de moins de 60 ans et “60 et plus” pour les personnes âgées de 60 ans et plus. Mais cette fois, affectez également à ceux dont l’âge est manquant la valeur “Age manquant”.

```
étez le code avec votre réponse :

roupe_nas <-
_influ %>% ____________________
```

La colonne `sexe` du jeu de données `liste_influ` contient les valeurs “f”, “m” et `NA` :

```
ıflu %>%
(sexe)
```

Recodez “f”, “m” et `NA` respectivement en “Femme”, “Homme” et “Sexe manquant”. Vous devez modifier la colonne `sexe` existante, pas créer une nouvelle colonne.

```
étez le code avec votre réponse :
recoder <-
_influ %>%
e(sexe = ____________________________________)
```

Conserver les valeurs par défaut d’une variable

Le côté droit (RHS) d’une formule `case_when()` peut également prendre une variable de votre jeu de données. C’est souvent utile lorsque vous voulez changer seulement quelques valeurs dans une colonne.

Voyons un exemple avec la colonne `edu_haute`, qui contient le plus haut niveau d’éducation atteint par un répondant :

```
_educ <-
de %>%
(edu_haute)

_educ
```

Ci-dessous, nous créons une nouvelle colonne, `edu_haute_recode`, où nous recodons à la fois “University” et “Doctorate” par la valeur “Post-secondary” :

```
_educ %>%
e(edu_haute_recode =
    case_when(
      edu_haute %in% c("University", "Doctorate") ~ "Post-secondary"
    ))
```

Ça a fonctionné, mais maintenant nous avons des `NA` pour toutes les autres lignes. Pour conserver ces autres lignes à leurs valeurs par défaut, nous pouvons ajouter la ligne `TRUE ~ edu_haute` (avec une variable, `edu_haute`, sur le côté droit d’une formule) :

```
_educ %>%
e(edu_haute_recode =
    case_when(
      edu_haute %in% c("University", "Doctorate") ~ "Post-secondary",
      TRUE ~ edu_haute
      ))
```

Maintenant, la déclaration `case_when()` se lit : ‘Si l’éducation la plus élevée est “University” ou “Doctorate”, inscrivez “Post-secondary”. Pour tout le monde, inscrivez la valeur de `edu_haute`’.

Ci-dessus, nous avons placé les valeurs recodées dans une colonne séparée, `edu_haute_recode`, mais pour ce type de remplacement, il est plus courant de simplement écraser la colonne existante :

```
_educ %>%
e(edu_haute =
    case_when(
      edu_haute %in% c("University", "Doctorate") ~ "Post-secondary",
      TRUE ~ edu_haute
      ))
```

Nous pouvons lire cette dernière déclaration `case_when()` comme suit : ‘Si l’éducation la plus élevée est “University” ou “Doctorate”, *changez la valeur en* “Post-secondary”. Pour tout le monde, *laissez la valeur de* `edu_haute`’.

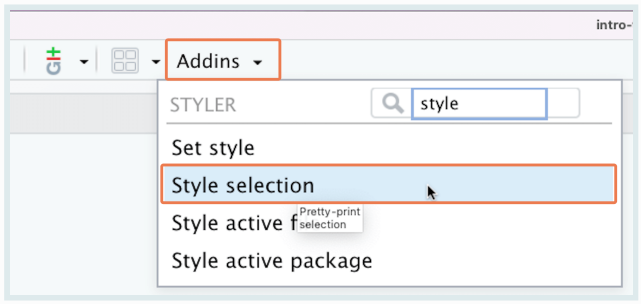

```
ètez le code avec votre réponse :

er_recovery <-
 influ %>%
e(devenir = _____________________________)
```

(Nous savons que c'est beaucoup de code pour un si petit changement. Plus tard, vous verrez des façons plus simples de faire cela.)

Éviter les lignes de code trop longues Au fur et à mesure que vous commencez à écrire des instructions `case_when()` de plus en plus complexes, il devient utile d'utiliser des sauts de ligne pour éviter des lignes de code trop longues.

Pour vous aider à créer des sauts de ligne, vous pouvez utiliser le package {styler}. Installez-le avec `pacman::p_load(styler)`. Ensuite, pour reformater n'importe quelle partie du code, mettez en surbrillance le code, cliquez sur le bouton "Addins" dans RStudio, puis cliquez sur "Style selection" :



Alternativement, vous pourriez mettre en surbrillance le code et utiliser le raccourci `Shift` + `Command/Control` + `A` pour utiliser le reformateur de code intégré de RStudio. Parfois, {styler} fait un meilleur travail de reformatage. Parfois, le reformateur intégré fait un meilleur travail.

Conditions multiples sur une seule variable

Les conditions LHS dans les formules `case_when()` peuvent avoir plusieurs parties. Voyons un exemple de cela.

Mais tout d'abord, nous allons nous inspirer de ce que nous avons appris dans la leçon `mutate()` et recréer la variable IMC. Cela implique d'abord de convertir la variable `taille_cm` en mètres, puis de calculer l'IMC.

```
_IMC <-
de %>%
vertissons notre taille en mètres et définissons l'IMC
e(taille_m = taille_cm/100,
  IMC = (poids_kg / (taille_m)^2)) %>%
ectionnons uniquement l'IMC
t(IMC)

_IMC
```

Rappelez-vous les catégories suivantes pour l'IMC:

- Si l'IMC est inférieur à 18,5, la personne est considérée comme étant en sous-poids.
- Un IMC normal est supérieur ou égal à 18,5 et inférieur à 25.
- Un IMC en surpoids est supérieur ou égal à 25 et inférieur à 30.
- Un IMC obèse est supérieur ou égal à 30.

La condition `IMC >= 18.5 & IMC < 25` pour définir `Poids normal` est une condition composée car elle a *deux* comparateurs : `>=` et `<`.

```
_IMC <-  yaounde_IMC %>%
e(classification_IMC = case_when(IMC < 18.5 ~'Sous-poids',
                                 IMC >= 18.5 & IMC < 25 ~ 'Poids normal',
                                 IMC >= 25 & IMC < 30 ~ 'Surpoids',
                                 IMC >= 30 ~ 'Obèse'))
_IMC
```

Utilisons `tabyl()` pour jeter un coup d'œil à nos données :

```
_IMC %>%
(classification_IMC)
```

Mais vous pouvez voir que les niveaux d'IMC sont définis par ordre alphabétique de Obèse à Surpoids, au lieu de aller du plus léger (Sous-poids) au plus lourd (Obèse). Rappelez-vous que si vous voulez avoir un certain ordre, vous pouvez faire de `classification_IMC` un facteur en utilisant `mutate()` et définir ses niveaux.

```
_IMC %>%
e(classification_IMC = factor(classification_IMC, levels=c("Obèse",
                                                           "Surpoids",
                                                           "Poids normal",
                                                           "Sous-poids"))) %>%

(classification_IMC)
```

Avec les conditions composées, il faut se rappeler d'entrer le nom de la variable *à chaque fois* qu'il y a un comparateur. Les apprenants R oublient souvent cela et essaieront d'exécuter du code qui ressemble à ceci :

```
_IMC %>%
e(classification_IMC = case_when(IMC < 18.5 ~'Sous-poids',
                                 IMC >= 18.5 & < 25 ~ 'Poids
 normal',
                                 IMC >= 25 & < 30 ~ 'Surpoids',
                                 IMC >= 30 ~ 'Obèse'))
```

Les définitions pour les catégories "Poids normal" et "Surpoids" sont erronées. Voyez-vous le problème ? Essayez d'exécuter le code pour repérer l'erreur.

Avec les données `liste_influ`, créez une nouvelle colonne, appelée `adolescent`, qui a la valeur "Oui" pour les personnes âgées de 10 à 19 ans (au moins 10 ans et moins de 20 ans), et "Non" pour tous les autres.

```
étez le code avec votre réponse :
scent_grouping <-
_influ %>% ____________________
```

Conditions multiples sur variables multiples

Dans tous les exemples vus jusqu'à présent, vous n'avez utilisé que des conditions impliquant une seule variable à la fois. Mais les conditions de LHS se réfèrent souvent à plusieurs variables à la fois.

Voyons un exemple simple avec l'âge et le sexe dans le jeu de données `yaounde`. Tout d'abord, nous sélectionnons uniquement ces deux variables pour une illustration facile :

```
_age_sexe <-
de %>%
(age_annees, sexe)

_age_sexe
```

Maintenant, imaginons que nous voulons recruter des femmes et des hommes dans le groupe d'âge 20-29 ans dans deux études. Pour cela, nous aimerions créer une colonne, appelée `recruit`, avec le schéma suivant :

- Les femmes âgées de 20 à 29 ans devraient avoir la valeur "Recruter pour l'étude féminine"
- Les hommes âgés de 20 à 29 ans devraient avoir la valeur "Recruter pour l'étude masculine"
- Tous les autres devraient avoir la valeur "Ne pas recruter"

Pour faire cela, nous exécutons l'instruction case_when suivante :

```
_age_sexe %>%
e(recruit = case_when(
e == "Female" & age_annees >= 20 & age_annees <= 29 ~ "Recruter pour l'étude
féminine",
e == "Male" & age_annees >= 20 & age_annees <= 29 ~ "Recruter pour l'étude
masculine",
~ "Ne pas recruter"
```

Vous pourriez également ajouter des paires supplémentaires de parenthèses autour des critères d'âge dans chaque condition :

```
_age_sexe %>%
e(recruit = case_when(
e == "Female" & (age_annees >= 20 & age_annees <= 29) ~ "Recruter pour l'étude
féminine",
e == "Male" & (age_annees >= 20 & age_annees <= 29) ~ "Recruter pour l'étude
masculine",
~ "Ne pas recruter"
```

Cette paire supplémentaire de parenthèses ne change pas la sortie du code, mais elle améliore la cohérence parce que le lecteur peut voir visuellement que votre condition est composée de deux parties, une pour le sexe, `sexe == "Female"`, et une autre pour l'âge, `(age_annees >= 20 & age_annees <= 29)`.

Avec les données `liste_influ`, créez une nouvelle colonne, appelée `recruter` avec le schéma suivant :

- Les personnes âgées de 30 à 59 ans (au moins 30 ans, moins de 60 ans) de la province du Jiangsu devraient avoir la valeur "Recruter pour l'étude Jiangsu"

- Les personnes âgées de 30 à 59 ans de la province du Zhejiang devraient avoir la valeur “Recruter pour l’étude Zhejiang”
- Tous les autres devraient avoir la valeur “Ne pas recruter”

```
étez le code avec votre réponse :
rovince_grouping <-
_influ %>%
e(recruter = _____________________________)
```

Ordre de priorité des conditions dans `case_when()`

Notez que l’ordre des conditions est important, car les conditions listées en haut de votre instruction `case_when()` sont prioritaires sur les autres.

Pour comprendre cela, exécutez l’exemple ci-dessous :

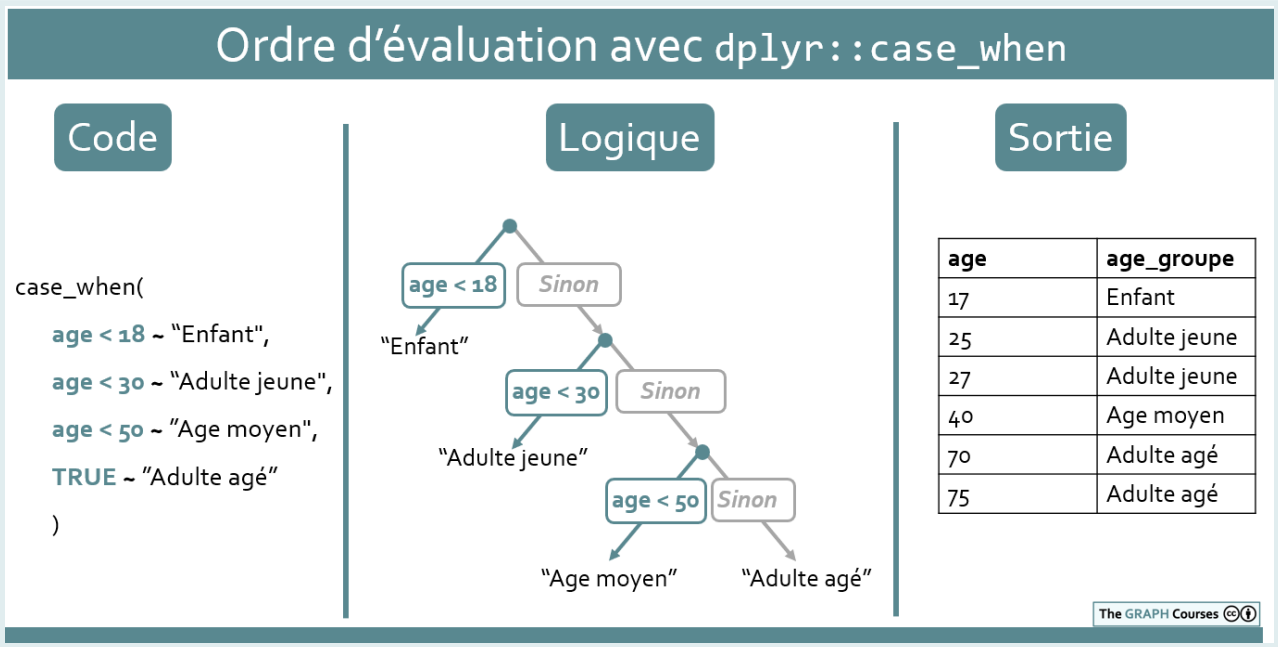
```
_age_sexe %>%
e(age_groupe = case_when(age_annees < 18 ~ "Enfant",
                         age_annees < 30 ~ "Adulte jeune",
                         age_annees < 50 ~ "Adulte moyen",
                         age_annees < 120 ~ "Adulte agé"))
```

Au premier abord, cela ressemble à une instruction `case_when()` défectueuse car les conditions d’âge se chevauchent. Par exemple, l’instruction `age_annees < 120 ~ "Adulte agé"` (qui se lit “si l’âge est inférieur à 120, saisissez ‘Adulte agé’”) suggère que *n’importe qui* entre 0 et 120 ans (même un bébé de 1 an !) serait codé comme “Adulte agé”.

Mais comme vous l’avez vu, le code fonctionne bien ! Les personnes de moins de 18 ans sont toujours codées comme “Enfant”.

Que se passe-t-il ? Essentiellement, l’instruction `case_when()` est interprétée comme une série d’étapes logiques de branchement, en commençant par la première condition. Cette instruction particulière peut donc se lire comme suit : “Si l’âge est inférieur à 18 ans, saisissez ‘Enfant’, *sinon*, si l’âge est inférieur à 30 ans, saisissez ‘Adulte jeune’, *sinon*, si l’âge est inférieur à 120 ans, saisissez ‘Adulte agé’”.

Ceci est illustré dans le schéma ci-dessous :

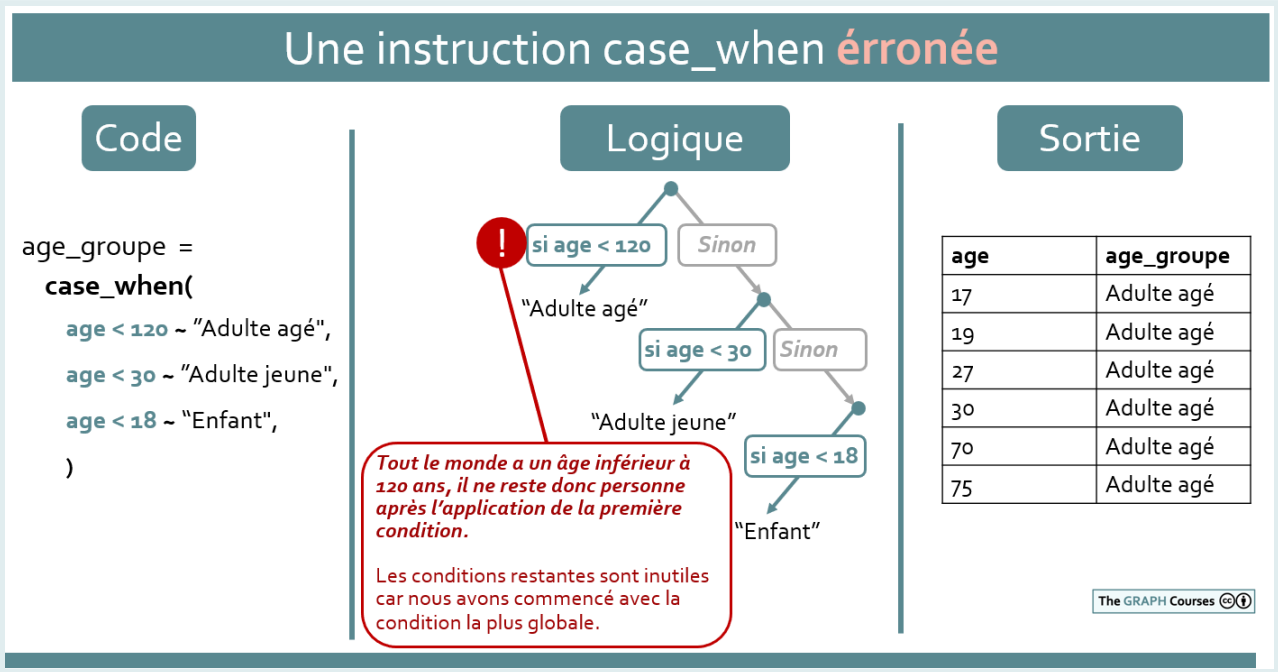


Cela signifie que si vous inversez l'ordre des conditions, vous obtiendrez une instruction `case_when()` erronée:

```
_age %>%
e(age_groupe = case_when(age_annees < 120 ~ "Adulte agé",
                         age_annees < 50 ~ "Adulte moyen",
                         age_annees < 30 ~ "Adulte jeune",
                         age_annees < 18 ~ "Enfant"))
```

Comme vous pouvez le constater, tout le monde est codé comme "Adulte agé". Cela se produit parce que la première condition correspond à tout le monde, il ne reste donc plus personne pour correspondre aux conditions suivantes. L'instruction peut se lire "Si l'âge est inférieur à 120 ans, saisissez 'Adulte agé', *sinon* si l'âge est inférieur à 30 ans…". Mais il n'y a pas de "sinon" car tout le monde a déjà été mis en correspondance !

Ceci est illustré dans le diagramme ci-dessous :



Bien que nous ayons passé beaucoup de temps à expliquer l'importance de l'ordre des conditions, dans cet exemple précis, il y aurait une façon beaucoup plus claire d'écrire ce code qui ne dépendrait pas de l'ordre des conditions. Plutôt que de laisser les groupes d'âge ouverts comme ceci :

```
age_annees < 120 ~ "Adulte agé"
```

vous devriez en fait utiliser des limites d'âge *fermées* comme ceci :

```
age_annees >= 30 & age_annees < 120 ~ "Adulte agé"
```

qui se lit : "si l'âge est supérieur ou égal à 30 ans et inférieur à 120 ans, saisissez 'Adulte agé'".

Avec de telles conditions fermées, l'ordre des conditions n'a plus d'importance. Vous obtenez le même résultat quel que soit l'arrangement des conditions :

```
ncer avec la condition "Adulte agé"
_age %>%
ivons des conditions de bornes fermées sur l'âge
e(age_groupe = case_when(age_annees >= 30 & age_annees < 120 ~ "Adulte agé",
                         age_annees >= 18 & age_annees < 30 ~ "Adulte jeune",
                         age_annees >= 0 & age_annees < 18 ~ "Enfant"))
```

```
ncer avec la condition "Enfant"
_age %>%  mutate(age_group = case_when(age_annees >= 0 & age_annees < 18 ~
 "Enfant",
                         age_annees >= 18 & age_annees < 30 ~ "Adulte jeune",
                         age_annees >= 30 & age_annees < 120 ~ "Adulte agé"))
```

Bien net et clair !

Alors pourquoi avons-nous passé autant de temps à expliquer l'importance de l'ordre des conditions si vous pouvez simplement éviter les catégories ouvertes et ne pas avoir à vous soucier de l'ordre des conditions ?

Une raison est que la compréhension de l'ordre des conditions devrait maintenant vous aider à voir pourquoi il est important de placer la condition `TRUE` comme dernière ligne de votre instruction `case_when()`. La condition `TRUE` correspond à *chaque ligne qui n'a pas encore été mise en correspondance*, donc si vous l'utilisez en premier dans le `case_when()`, elle correspondra à *tout le monde* !

L'autre raison est qu'il existe certains cas où vous voudrez *peut-être* utiliser des conditions qui se chevauchent et ouvertes, et vous devrez donc faire attention à l'ordre des conditions. Voyons un tel exemple maintenant : l'identification des symptômes de type COVID. Notez qu'il s'agit d'un matériel un peu avancé, probablement un peu au-dessus de vos besoins actuels. Nous l'introduisons maintenant pour que vous en soyez conscient et que vous restiez vigilant avec `case_when()` à l'avenir.

Conditions superposées avec `case_when()`

Nous voulons identifier les symptômes semblables à ceux de la COVID-19 dans nos données. Considérons les colonnes de symptômes dans le jeu de données `yaounde`, qui indiquent quels symptômes ont été ressentis par les répondants sur une période de 6 mois :

```
%>%
(starts_with("symp_"))
```

Nous aimerions utiliser cela pour évaluer si une personne a pu avoir la COVID-19, en suivant partiellement les directives recommandées par l'OMS.

- Les individus avec toux doivent être classés comme "cas possibles de COVID-19"
- Les individus avec anosmie/agueusie (perte d'odeur ou de goût) doivent être classés comme "cas probables de COVID-19".

Maintenant, en gardant ces critères à l'esprit, considérons une personne, appelons-la Osma, qui a à la fois de la toux ET de l'anosmie/agueusie ? Comment devrions-nous classer Osma ?

Elle remplit les critères pour être un "cas possible de COVID-19" (parce qu'elle a de la toux), mais elle remplit *aussi* les critères pour être un "cas probable de COVID-19" (parce qu'elle a de l'anosmie/agueusie). Alors, dans quel groupe devrait-elle être classée, "cas possible de COVID-19" ou "cas probable de COVID-19" ? Pensez-y pendant une minute.

Vous avez probablement deviné qu'elle devrait être classée comme un "cas probable de COVID-19". "Probable" est plus probable que "Possible" ; et le symptôme de l'anosmie/agueusie est plus *significatif* que le symptôme de la toux. On pourrait dire

que le critère pour “cas probable de COVID-19” a une spécificité plus élevée ou une *préséance* plus élevée que le critère pour “cas possible de COVID-19”.

Par conséquent, lors de la construction d’une déclaration `case_when()`, la condition “cas probable de COVID-19” devrait également prendre une préséance plus élevée - elle devrait venir *en premier* dans les conditions fournies à `case_when()`. Voyons cela maintenant.

D’abord, nous sélectionnons les variables pertinentes, pour une illustration facile. Nous identifions également et `slice()` des lignes spécifiques qui sont utiles pour la démonstration :

```
_symptomes_slice <-
le %>%
(symp_toux, symp_anosmie_agueusie) %>%
ce de lignes spécifiques utiles pour la démo
fois que vous trouvez le bon code, vous devriez supprimer ce slice
(32, 711, 625, 651 )

_symptomes_slice
```

Maintenant, la déclaration `case_when()` correcte, qui a la condition “COVID-19 probable” en premier :

```
_symptomes_slice %>%
e(statut_covid = case_when(
_anosmie_agueusie == "Yes" ~ "COVID-19 Probable",
_toux == "Yes"  ~ "COVID-19 Possible"
```

Cette déclaration `case_when()` peut être lue en termes simples comme ‘Si la personne a de l’anosmie/agueusie, inscrire “COVID-19 Probable”, sinon, si la personne a de la toux, inscrire “COVID-19 Possible”’.

Maintenant, passez du temps à examiner le jeu de données de sortie, en particulier les trois derniers individus. L’individu de la ligne 2 remplit le critère pour être un “cas possible de COVID-19” parce qu’il a de la toux (`symp_toux` == “Yes”), et l’individu de la ligne 3 remplit le critère pour être un “cas probable de COVID-19” parce qu’il a de l’anosmie/agueusie (`symp_anosmie_agueusie` == “Yes”).

L’individu de la ligne 4 est Osma, qui remplit à la fois les critères pour être un “cas possible de COVID-19” *et* pour un “cas probable de COVID-19”. Et parce que nous avons organisé nos conditions `case_when()` dans le bon ordre, elle est correctement codée comme “COVID-19 probable”. Super !

Mais remarquez ce qui se passe si nous échangeons l’ordre des conditions :

```
_symptomes_slice %>%
e(statut_covid = case_when(
_toux == "Yes"  ~ "COVID-19 Possible",
_anosmie_agueusie == "Yes" ~ "COVID-19 Probable"
```

Oh non ! Osma à la ligne 4 est maintenant mal classée comme “COVID-19 Possible” alors qu’elle a le symptôme plus significatif d’anosmie/agueusie. C’est parce que la première condition `symp_toux == "Yes"` l’a correspondue en premier, et donc la deuxième condition n’a pas pu la correspondre !

Vous voyez maintenant pourquoi vous devez parfois réfléchir profondément à l’ordre de vos conditions `case_when()`. C’est un point mineur, mais il peut vous mordre à des moments inattendus. Même les analystes expérimentés ont tendance à faire des erreurs qui peuvent être attribuées à un mauvais arrangement des déclarations `case_when()`.

CHALLENGE

En réalité, il *existe* encore une autre solution pour éviter de mal classer la personne qui a de la toux et de l’anosmie/agueusie. C’est d’ajouter `symp_anosmie_agueusie != "Yes"` (n’est pas égal à “Yes”) aux conditions pour “COVID-19 Possible”. Pouvez-vous penser à pourquoi cela fonctionne ?

```
_symptomes_slice %>%
e(statut_covid = case_when(
_toux == "Yes" & symp_anosmie_agueusie != "Yes" ~ "COVID-19
 Possible",
_anosmie_agueusie == "Yes" ~ "COVID-19 Probable"))
```

PRACTICE (in RMD)

Avec le jeu de données `liste_influ`, créez une nouvelle colonne appelée `priorite_de_suivi` qui implémente le schéma suivant :

- Les femmes doivent être considérées comme “Haute priorité”
- Tous les enfants (de moins de 18 ans) de n’importe quel sexe doivent être considérés comme “Priorité la plus élevée”.
- Tous les autres doivent avoir la valeur “Pas de priorité”

```
étez le code avec votre réponse :
_priorite_groupes <-
_influ %>%
e(priorite_de_suivi = _________________)
```

Conditions binaires : `dplyr::if_else()`

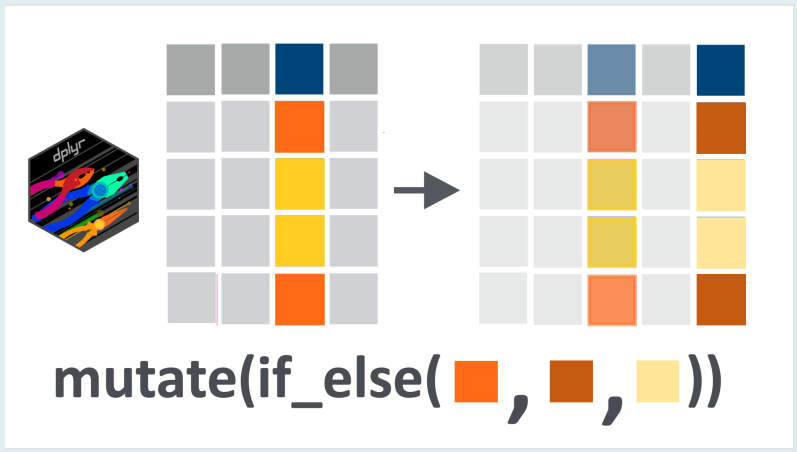


Fig: les conditions if_else().

Il existe une autre fonction {dplyr} similaire à `case_when()` pour lorsque nous voulons appliquer une condition binaire à une variable : `if_else()`. Une condition binaire est soit `TRUE` soit `FALSE`.`if_else()` a une application similaire à `case_when()` : si la condition est vraie, une opération est appliquée, si la condition est fausse, l'alternative est appliquée. La syntaxe est : `if_else(CONDITION, SI_VRAI, SI_FAUX)`. Comme vous pouvez le voir, cela ne permet que d'appliquer une condition binaire (et non des cas multiples, comme avec `case_when()`).

Si nous prenons l'un des premiers exemples sur le recodage de la variable `highest_education`, nous pouvons l'écrire soit avec `case_when()` soit avec `if_else()`.

Voici la version que nous avons déjà explorée :

```
educ %>%
(edu_haute =
    case_when(
      edu_haute %in% c("University", "Doctorate") ~ "Post-secondary",
      TRUE ~ edu_haute
      ))
```

Et voici comment nous l'écririons en utilisant `if_else()` :

```
educ %>%
(edu_haute =
    if_else(
      edu_haute %in% c("University", "Doctorate"),
      # si TRUE alors on recodifie
      "Post-secondary",
      # si FALSE alors on garde la valeur par défaut
      edu_haute
      ))
```

Comme vous pouvez le voir, nous obtenons le même résultat, que nous utilisions `if_else()` ou `case_when()`.

Avec les données `liste_influ`, créez une nouvelle colonne, appelée `age_groupe`, qui a la valeur “Moins de 50” pour les personnes de moins de 50 ans et “50 ans et plus” pour les personnes âgées de 50 ans et plus. Utilisez la fonction `if_else()`.

C’est exactement la même question que votre première question de pratique, mais cette fois vous devez utiliser `if_else()`.

```
étez le code avec votre réponse :

roupe_if_else <-
_influ %>%
e(age_groupe = if_else(_____________________________))
```

Récapitulatif

Modifier ou construire vos variables en fonction de conditions sur d’autres variables est l’une des tâches de nettoyage de données les plus répétées. À tel point que cela méritait sa propre leçon !

J’espère maintenant que vous vous sentirez à l’aise pour utiliser `case_when()` et `if_else()` dans `mutate()` et que vous êtes enthousiaste à l’idée d’apprendre des opérations {dplyr} plus complexes comme le groupement de variables et leur résumé.À la prochaine !

Fig: Les conditions if_else() et `case_when()`.

Contributeurs

L'équipe suivante a contribué à cette leçon :

LAURE VANCAUWENBERGHE

Data analyst, the GRAPH Network
A firm believer in science for good, striving to ally programming, health and education

KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement

GUY WAFEU

R Instructor and Public Health Physician
Committed to improving the quality of data analysis

SABINA RODRIGUEZ VELÁSQUEZ

Project Manager and Scientific Collaborator, The GRAPH Network
Infectiously enthusiastic about microbes and Global Health

Références

Certains matériaux de cette leçon ont été adaptés des sources suivantes :

- Horst, A. (2022). *Dplyr-learnr*. https://github.com/allisonhorst/dplyr-learnr (Œuvre originale publiée en 2020)
- *Créer, modifier, et supprimer des colonnes — Mutate*. (s.d.). Consulté le 21 février 2022, à partir de https://dplyr.tidyverse.org/reference/mutate.html

L'artwork a été adapté de :

- Horst, A. (2022). *Illustrations R & stats par Allison Horst*. https://github.com/allisonhorst/stats-illustrations (Œuvre originale publiée en 2018)