Notes de leçon : Introduction aux fonctions et aux conditionnelles

Introduction
Objectifs d'apprentissage
Packages
  Bases d'une fonction
Quand écrire une fonction en R
Fonctions avec Plusieurs Arguments
Passage d'Arguments aux Fonctions Internes
  L'Argument Ellipse, `...`
Compréhension de la Portée en R
Introduction aux Conditionnels : `if`, `else if` et `else`
Vérification des arguments avec des conditionnels
Conditionnels Vectorisés
Où stocker vos fonctions
Conclusion !
Corrigés

Introduction

Les deux composants principaux du langage R sont les objets et les fonctions. Les objets sont les structures de données que nous utilisons pour stocker des informations, et les fonctions sont les outils que nous utilisons pour manipuler ces objets. Citant John Chambers, qui a joué un rôle clé dans le développement du langage R, tout ce qui "existe" dans un environnement R est un objet, et tout ce qui "se passe" est une fonction.

Jusqu'à présent, vous avez principalement utilisé des fonctions écrites par d'autres. Dans cette leçon, vous apprendrez à écrire vos propres fonctions en R.

Écrire des fonctions vous permet d'automatiser des tâches répétitives, d'améliorer l'efficacité et de réduire les erreurs dans votre code.

Dans cette leçon, nous apprendrons les fondamentaux des fonctions avec des exemples simples. Puis, dans une leçon future, nous écrirons des fonctions plus complexes qui peuvent automatiser de grandes parties de votre flux de travail d'analyse de données.

Objectifs d'apprentissage

À la fin de cette leçon, vous serez capable de :

1. Créer et utiliser vos propres fonctions en R.
2. Concevoir des arguments de fonction et définir des valeurs par défaut.
3. Utiliser une logique conditionnelle telle que `if`, `else if`, et `else` au sein des fonctions.
4. Vérifier et valider les arguments de fonction pour prévenir les erreurs.
5. Gérer la portée des fonctions et comprendre les variables locales vs. globales.

6. Gérer des données vectorisées dans les fonctions.
7. Organiser et stocker vos fonctions personnalisées pour une réutilisation facile.

Packages

Exécutez le code suivant pour installer et charger les packages nécessaires pour cette leçon :

```
if (!require(pacman)) install.packages("pacman")
pacman::p_load(tidyverse, here, NHSRdatasets, medicaldata, outbreaks,
reactable)
```

Bases d'une fonction

Commençons par créer une fonction très simple. Considérez la fonction suivante qui convertit les pounds (une unité de poids) en kilogrammes (une autre unité de poids) :

```
pounds_en_kg <- function(pounds) {
  return(pounds * 0.4536)
}
```

Si vous exécutez ce code, vous créerez une fonction nommée `pounds_en_kg`, qui peut être utilisée directement dans un script ou dans la console :

```
pounds_en_kg(150)
```

```
## [1] 68.04
```

Décortiquons la structure de cette première fonction étape par étape.

Tout d'abord, une fonction est créée en utilisant l'instruction `function`, suivie d'une paire de parenthèses et d'une paire d'accolades.

```
function() {

}
```

À l'intérieur des parenthèses, nous indiquons les **arguments** de la fonction. Notre fonction ne prend qu'un seul argument, que nous avons décidé de nommer `pounds`. C'est la valeur que nous voulons convertir de pounds en kilogrammes.

```
function(pounds) {

}
```

Bien sûr, nous aurions pu nommer cet argument comme nous le voulions.

L'élément suivant, à l'intérieur des accolades, est le **corps** de la fonction. C'est là que nous écrivons le code que nous voulons exécuter lorsque la fonction est appelée.

```
function(pounds) {
  pounds * 0.4536
}
```

Maintenant, nous voulons que notre fonction retourne ce qui est calculé à l'intérieur de son corps. Cela est réalisé via l'instruction `return`.

```
function(pounds) {
  return(pounds * 0.4536)
}
```

Parfois, vous pouvez omettre l'instruction `return` et simplement écrire l'expression à retourner à la fin de la fonction, car R retournera automatiquement la dernière expression évaluée dans la fonction :

```
function(pounds) {
  pounds * 0.4536 # R retournera automatiquement cette expression
}
```

Cependant, il est conseillé d'inclure toujours l'instruction `return`, car cela rend le code plus lisible.

Nous pourrions aussi vouloir d'abord assigner le résultat à un objet puis le retourner :

```
function(pounds) {
  kg <- pounds * 0.4536
  return(kg)
}
```

C'est un peu plus long, mais cela rend la fonction plus claire.

Enfin, pour que notre fonction puisse être appelée et utilisée, nous devons lui donner un nom. Cela revient à stocker une valeur dans un objet. Ici, nous la stockons dans un objet nommé `pounds_to_kg`.

```
pounds_to_kg <- function(pounds) {
  kg <- pounds * 0.4536
  return(kg)
}
```

Avec notre fonction, nous avons donc créé un nouvel objet dans notre environnement appelé `pounds_to_kg`, de classe `function`.

```
class(pounds_to_kg)
```

```
## [1] "function"
```

Nous pouvons maintenant l'utiliser ainsi avec un argument nommé :

```
pounds_to_kg(pounds = 150)
```

```
## [1] 68.04
```

Ou sans un argument nommé :

```
pounds_to_kg(150)
```

```
## [1] 68.04
```

La fonction peut également être utilisée avec un vecteur de valeurs :

```
my_vec <- c(150, 200, 250)
pounds_to_kg(my_vec)
```

```
## [1]  68.04  90.72 113.40
```

Et voilà ! Vous venez de créer votre première fonction en R.

Vous pouvez voir le code source de n'importe quelle fonction en tapant son nom sans parenthèses :

```
pounds_to_kg
```

```
## function(pounds) {
##   kg <- pounds * 0.4536
##   return(kg)
## }
## <bytecode: 0x298ac89d8>
```

Pour voir cela comme un script R, vous pouvez utiliser la fonction `View` :

```
View(pounds_to_kg)
```

Cela ouvrira un nouvel onglet dans RStudio avec le code source de la fonction.

Cette méthode fonctionne pour n'importe quelle fonction, pas seulement celles que vous créez. Pour un exemple à quoi ressemble une *vraie* fonction dans la pratique, essayez de

```
View(reactable)
```

Fonction Age Mois

Créez une fonction simple appelée `years_to_months` qui transforme l'âge en années en âge en mois.

Essayez-la avec `years_to_months(12)`

```
# Votre code ici
years_to_months <- ...
```

Écrivons maintenant une fonction un peu plus complexe, pour un peu plus de pratique. La fonction que nous allons écrire convertira une température en Fahrenheit (utilisée aux États-Unis) en température en Celsius. La formule pour cette conversion est :

$$C = \frac{5}{9} \times (F - 32)$$

Et voici la fonction :

```
fahrenheit_to_celsius <- function(fahrenheit) {
  celsius <- (5 / 9) * (fahrenheit - 32)
  return(celsius)
}

fahrenheit_to_celsius(32) # point de congélation de l'eau. Devrait être 0
```

```
## [1] 0
```

Testons la fonction sur une colonne du jeu de données `airquality`, qui est l'un des jeux de données intégrés dans R :

```
airquality %>%
  select(Temp) %>%
  mutate(Temp = fahrenheit_to_celsius(Temp)) %>%
  head()
```

```
##       Temp
## 1 19.44444
## 2 22.22222
## 3 23.33333
## 4 16.66667
## 5 13.33333
## 6 18.88889
```

Super !

Quand écrire une fonction en R

Dans R, de nombreuses opérations peuvent être complétées en utilisant des fonctions existantes ou en combinant quelques-unes. Cependant, il y a des occasions où il est avantageux de créer sa propre fonction :

- **Réutilisabilité** : Si vous vous retrouvez à écrire le même code à plusieurs reprises, il peut être bénéfique de l'encapsuler dans une fonction. Par exemple, si vous convertissez fréquemment des températures de Fahrenheit en Celsius, créer une fonction `fahrenheit_to_celsius` rendrait votre code plus épuré et améliorerait l'efficacité.

- **Lisibilité** : Les fonctions peuvent améliorer la lisibilité du code, en particulier lorsqu'elles ont des noms descriptifs. Avec des fonctions simples comme `fahrenheit_to_celsius`, les avantages ne sont pas toujours évidents. Cependant, au fur et à mesure que les fonctions deviennent plus complexes, l'importance des noms descriptifs devient de plus en plus cruciale.

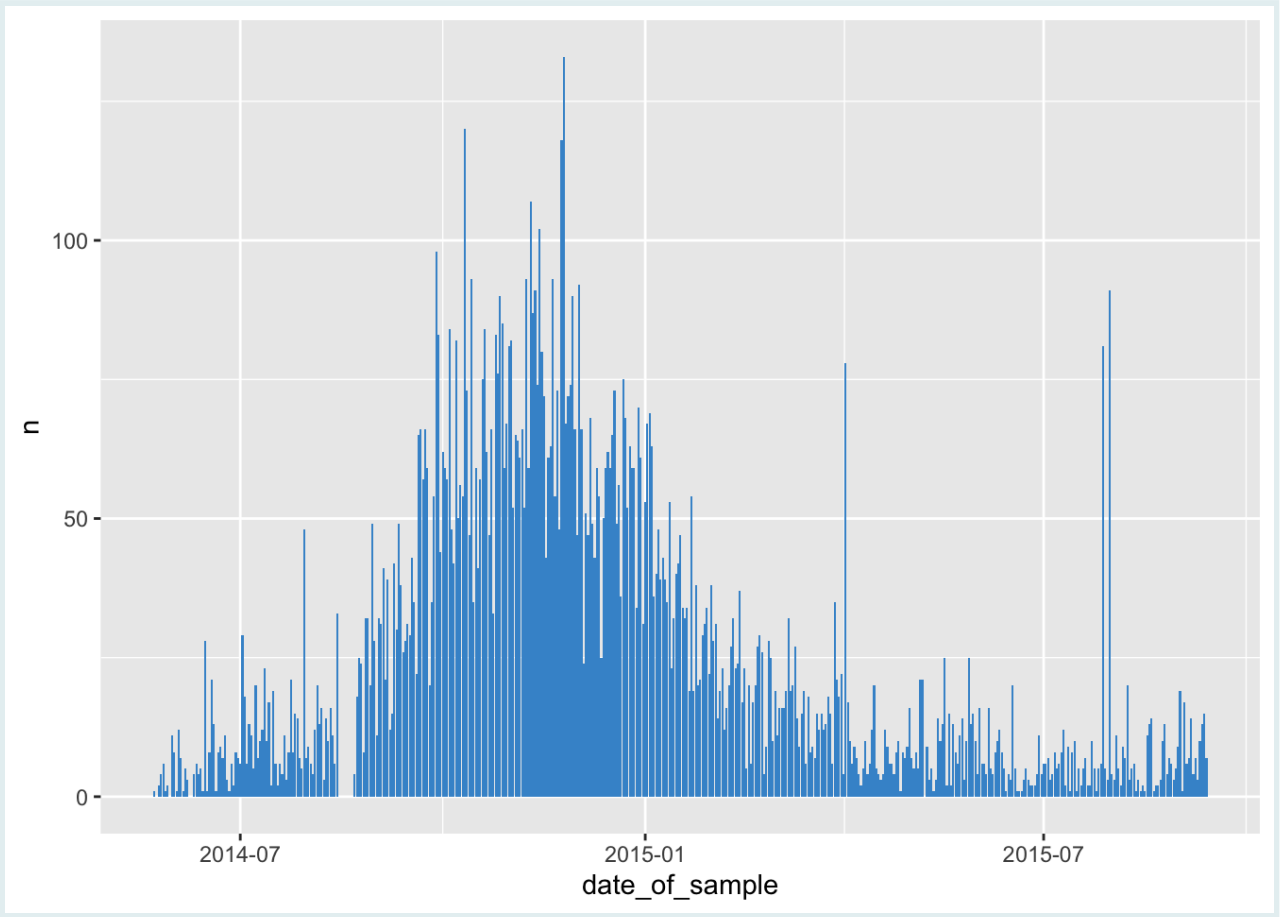
- **Partage** : Les fonctions facilitent le partage du code. Elles peuvent être distribuées soit dans le cadre d'un package, soit comme des scripts autonomes. Bien que la création d'un package soit plus complexe et au-delà du cadre de ce cours, partager des fonctions plus simples est assez direct. Nous parlerons des options pour cela plus tard dans la leçon.

::: note latérale **Fonctions pour les jeux de données et les Graphiques**

Les fonctions les plus utiles que vous écrirez probablement concerneront la manipulation de jeux de données et de graphiques. Voici une fonction qui prend une liste linéaire de cas et retourne un graphique de la courbe épidémique :

```
# Fonction pour tracer une courbe épidémique
tracer_courbe_epidemie <- function(donnees, colonne_date) {
  donnees %>%
    count({{ colonne_date }}) %>%
    complete({{ colonne_date }} := seq(min({{ colonne_date }}),
                                       max({{ colonne_date }}), by="day")) %>%
    ggplot(aes(x = {{ colonne_date }}, y = n)) +
    geom_col(fill = "#4395D1")
}

# Exemple d'utilisation
tracer_courbe_epidemie(outbreaks::ebola_sierraleone_2014, date_of_sample)
```



Cette leçon abordera des fonctions plus complexes plus tard. Pour l'instant, nous nous concentrerons sur les bases de la rédaction de fonctions, en utilisant comme exemples des fonctions simples de manipulation de vecteurs. :::

Fonction de Conversion Celsius en Fahrenheit

Créez une fonction nommée `celsius_en_fahrenheit` qui convertit la température de Celsius en Fahrenheit. Voici la formule pour cette conversion :

$$Fahrenheit = Celsius \times 1.8 + 32$$

```
# Votre code ici
celsius_en_fahrenheit <- ...
```

Ensuite, testez votre fonction sur la colonne `temp` du jeu de données intégré `beaver1` dans R :

```
beaver1 %>%
  select(temp) %>%
  head()
```

Fonctions avec Plusieurs Arguments

La plupart des fonctions prennent plusieurs arguments plutôt qu'un seul. Examinons un exemple de fonction qui prend trois arguments :

```
calculate_calories <- function(carb_grams, protein_grams, fat_grams) {
  result <- (carb_grams * 4) + (protein_grams * 4) + (fat_grams * 9)
  return(result)
}

calculate_calories(carb_grams = 50, protein_grams = 25, fat_grams = 10)
```

```
## [1] 390
```

La fonction `calculate_calories` calcul le total des calories basé sur les grammes de glucides, protéines, et lipides. On estime que les glucides et les protéines ont 4 calories par gramme, tandis que les lipides ont 9 calories par gramme.

Si vous essayez d'utiliser la fonction sans fournir tous les arguments, cela entraînera une erreur.

```
calculate_calories(carb_grams = 50, protein_grams = 25)
```

```
Erreur dans calculate_calories(carb_grams = 50, protein_grams = 25) :
  l'argument "fat_grams" est manquant, avec aucune valeur par défaut
```

Vous pouvez définir des **valeurs par défaut** pour les arguments de votre fonction. Si un argument est **appelé** sans qu'une **valeur ne lui soit attribuée**, alors cet argument prend sa valeur par défaut.

Voici un exemple où `fat_grams` se voit attribuer une valeur par défaut de `0`.

```
calculate_calories <- function(carb_grams, protein_grams, fat_grams = 0) {
  result <- (carb_grams * 4) + (protein_grams * 4) + (fat_grams * 9)
  return(result)
}

calculate_calories(50, 25)
```

```
## [1] 300
```

Dans cette version révisée, `carb_grams` et `protein_grams` sont des arguments obligatoires, mais nous pourrions rendre tous les arguments optionnels en leur donnant tous des valeurs par défaut :

```r
calculate_calories <- function(carb_grams = 0, protein_grams = 0, fat_grams =
0) {
  result <- (carb_grams * 4) + (protein_grams * 4) + (fat_grams * 9)
  return(result)
}
```

Maintenant, nous pouvons appeler la fonction sans arguments :

```r
calculate_calories()
```

```
## [1] 0
```

Nous pouvons également l'appeler avec certains arguments :

```r
calculate_calories(carb_grams= 50, protein_grams = 25)
```

```
## [1] 300
```

Et cela fonctionne comme prévu.

Fonction de Calcul de l'IMC

Créez une fonction nommée `calc_imc` qui calcule l'Indice de Masse Corporelle (IMC) pour une ou plusieurs personnes. Gardez à l'esprit que l'IMC est calculé comme le poids en kg divisé par le carré de la taille en mètres. Par conséquent, cette fonction nécessite deux arguments obligatoires : le poids et la taille.

```r
# Votre code ici
calc_imc <- ...
```

Ensuite, appliquez votre fonction au jeu de données `medicaldata::smartpill` pour calculer l'IMC de chaque personne :

```r
medicaldata::smartpill %>%
  as_tibble() %>%
  select(Weight, Height) %>%
  mutate(IMC = calc_imc(Weight, Height))
```

Passage d'Arguments aux Fonctions Internes

Lors de l'écriture de fonctions en R, vous pourriez avoir besoin d'utiliser des fonctions existantes au sein de votre fonction personnalisée. Par exemple, considérons notre fonction familière qui convertit les livres en kilogrammes :

```r
pounds_en_kg <- function(pounds) {
  kg <- pounds * 0.4536
  return(kg)
}
```

Il pourrait être utile de pouvoir arrondir le résultat à un nombre spécifié de décimales sans appeler une fonction séparée.

Pour cela, nous pouvons intégrer directement la fonction `round` dans notre fonction personnalisée. La fonction `round` a deux arguments : `x`, le nombre à arrondir, et `digits`, le nombre de décimales à arrondir :

```r
round(x = 1.2345, digits = 2)
```

```
## [1] 1.23
```

Maintenant, nous pouvons ajouter un argument à notre fonction appelé `arrondir_a` qui sera passé à l'argument `digits` de la fonction `round`.

```r
pounds_en_kg <- function(pounds, arrondir_a = 2) {
  kg <- pounds * 0.4536
  kg_arrondi <- round(x = kg, digits = arrondir_a)
  return(kg_arrondi)
}
```

Dans la fonction ci-dessus, nous avons ajouté un argument appelé `arrondir_a` avec une valeur par défaut de 3.

Maintenant, lorsque vous passez une valeur à l'argument `arrondir_a`, elle sera utilisée par la fonction `round`.

```r
pounds_en_kg(10) # sans argument passé à arrondir_a, la valeur par défaut de 2 est utilisée
```

```
## [1] 4.54
```

```r
pounds_en_kg(10, arrondir_a = 1)
```

```
## [1] 4.5
```

```r
pounds_en_kg(10, arrondir_a = 3)
```

```
## [1] 4.536
```

L'Argument Ellipse, `...`

Parfois, il y a de nombreux arguments à passer à une fonction interne. Par exemple, considérez la fonction `format()` en R, qui a de nombreux arguments :

```
format(x = 12364.2345,
       big.mark = " ", # séparateur de milliers
       decimal.mark = ",", # point décimal à la française !
       nsmall = 2, # nombre de chiffres après la virgule
       scientific = FALSE # utiliser la notation scientifique ?
       )
```

```
## [1] "12 364,23"
```

Vous pouvez voir tous les arguments en tapant `?format` dans la console.

Si nous voulons que notre fonction puisse passer tous ces arguments à la fonction `format`, nous utiliserons l'argument ellipse, `...`. Voici un exemple :

```
pounds_en_kg <- function(pounds, ...) {
  kg <- pounds * 0.4536
  kg_formate <- format(x = kg, ...)
  return(kg_formate)
}
```

Maintenant, lorsque nous passons des arguments à la fonction `livres_en_kg`, ils seront transmis à la fonction `format`, même si nous ne les avons pas explicitement définis dans notre fonction.

```
pounds_en_kg(10000.234)
```

```
## [1] "4536.106"
```

```
pounds_en_kg(10000.234, big.mark = " ", decimal.mark = ",")
```

```
## [1] "4 536,106"
```

Super !

Pratique avec l'Argument ...

Considérez notre fonction `calculer_calories()`.

```r
calculate_calories <- function(carb_grams, protein_grams, fat_grams) {
  result <- (carb_grams * 4) + (protein_grams * 4) + (fat_grams * 9)
  return(result)
}
```

Améliorez cette fonction pour accepter les arguments de formatage en utilisant le mécanisme `...`.

Compréhension de la Portée en R

La portée se réfère à la visibilité des variables et objets dans différentes parties de votre code R. Il est important de comprendre la portée lors de l'écriture de fonctions.

Les objets créés à l'intérieur d'une fonction ont une **portée locale** au sein de cette fonction (par opposition à une **portée globale**) et ne sont pas accessibles en dehors de la fonction. Illustrons cela avec la fonction `pounds_en_kg` :

Imaginez que vous voulez convertir un poids en pounds en kilogrammes et vous écrivez une fonction pour cela :

```r
pounds_en_kg <- function(pounds) {
  kg <- pounds * 0.4536
}
```

Vous pourriez être tenté d'essayer d'accéder à la variable `kg` en dehors de la fonction, mais vous obtiendrez une erreur :

```r
pounds_en_kg(50)
kg
```

```
Erreur : objet 'kg' introuvable
```

C'est parce que `kg` est une variable locale à l'intérieur de la fonction `livres_en_kg` et n'est pas accessible dans l'environnement global.

Pour utiliser une valeur générée à l'intérieur d'une fonction, nous devons nous assurer qu'elle est retournée par la fonction :

```r
pounds_en_kg <- function(pounds) {
  kg <- pounds * 0.4536
  return(kg)
}
```

Et ensuite, nous pouvons stocker le résultat de la fonction dans une variable globale :

```r
kg <- pounds_en_kg(50)
```

Maintenant, nous pouvons accéder à l'objet `kg` :

```r
kg
```

```
## [1] 22.68
```

L'Opérateur de Superassignation, `<<-`

Bien que nous ayons dit que les objets créés au sein d'une fonction ont une portée locale, il est en réalité **possible** de créer des variables globales depuis l'intérieur d'une fonction en utilisant l'opérateur spécial `<<-`.

Considérez l'exemple suivant :

```r
test <- function() {
  nouvel_objet <<- 15
}
```

Maintenant, si nous exécutons la fonction, `nouvel_objet` sera créé dans l'environnement global, avec une valeur de 15 :

```r
test()
nouvel_objet
```

```
## [1] 15
```

Bien que cela soit techniquement possible, cela n'est généralement pas recommandé (surtout pour les non-experts) en raison des effets secondaires potentiels et des défis de maintenance.

Introduction aux Conditionnels : `if`, `else if` et `else`

Les conditionnels, qui sont utilisés pour contrôler le flux d'exécution du code, sont une partie essentielle de la programmation, en particulier lors de l'écriture de fonctions. En R, les conditionnels sont mis en œuvre à l'aide des instructions `if`, `else` et `else if`.

Lorsque nous utilisons `if`, nous spécifions que nous voulons que certaines parties du code s'exécutent uniquement si une condition spécifique est vraie.

Voici la structure d'une instruction `if` :

```r
if (condition) {
  # code à exécuter si la condition est vraie
}
```

Remarquez que cela ressemble à la structure d'une fonction.

Maintenant, voyons une instruction `if` en action, pour convertir une température de Celsius en Fahrenheit :

```r
celsius <- 20
convertir_en <- "fahrenheit"

if (convertir_en == "fahrenheit") {
  fahrenheit <- (celsius * 9/5) + 32
  print(fahrenheit)
}
```

```
## [1] 68
```

Dans ce fragment de code, si la variable `convertir_en` est égale à `"fahrenheit"`, la conversion est réalisée et le résultat est imprimé.

Voyons maintenant ce qui se passe lorsque `convertir_en` est défini à une valeur autre que `"fahrenheit"` :

```r
convertir_en <- "kelvin"

if (convertir_en == "fahrenheit") {
  fahrenheit <- (celsius * 9/5) + 32
  print(fahrenheit)
}
```

Dans ce cas, rien n'est imprimé car la condition n'est pas satisfaite.

Pour gérer les situations où `convertir_en` n'est pas `"fahrenheit"`, nous pouvons ajouter une instruction `else` :

```r
convertir_en <- "kelvin"

if (convertir_en == "fahrenheit") {
  fahrenheit <- (celsius * 9/5) + 32
  print(fahrenheit)
} else {
  print(celsius)
}
```

```
## [1] 20
```

Ici, si la variable `convertir_en` ne correspond pas à `"fahrenheit"`, le code dans le bloc `else` s'exécute, imprimant la valeur originale en Celsius.

Pour vérifier plusieurs conditions spécifiques, nous pouvons utiliser `else if` :

```
convertir_en <- "kelvin"

if (convertir_en == "fahrenheit") {
  fahrenheit <- (celsius * 9/5) + 32
  print(fahrenheit)
} else if (convertir_en == "kelvin") {
  temp_kelvin <- celsius + 273.15
  print(temp_kelvin)
} else {
  print(celsius)
}
```

```
## [1] 293.15
```

Ici, le code gère trois possibilités :

- Convertir en Fahrenheit si `convertir_en` est "fahrenheit".
- Si `convertir_en` n'est PAS "fahrenheit", vérifie si c'est "kelvin". Si c'est le cas, convertir en Kelvin.
- Si `convertir_en` n'est ni "fahrenheit" ni "kelvin", imprimer la valeur originale en Celsius.

Notez que vous pouvez avoir autant d'instructions `else if` que nécessaire, tandis que vous ne pouvez avoir qu'une seule instruction `else` attachée à une instruction `if`.

Finalement, nous pouvons encapsuler cette logique dans une fonction. Nous allons assigner le résultat à l'intérieur de chaque conditionnel à une variable appelée `sortie` et retourner `sortie` à la fin de la fonction :

```
convertir_celsius <- function(celsius, convertir_en) {
  if (convertir_en == "fahrenheit") {
    sortie <- (celsius * 9/5) + 32
  } else if (convertir_en == "kelvin") {
    sortie <- celsius + 273.15
  } else {
    sortie <- celsius
  }
  return(sortie)
}
```

Testons la fonction :

```
convertir_celsius(20, "fahrenheit")
```

```
## [1] 68
```

```
convertir_celsius(20, "kelvin")
```

```
## [1] 293.15
```

Un problème avec la fonction actuelle est que si une valeur invalide est passée à `convert_to`, nous n'obtenons aucun message informatif à ce sujet :

```
convertir_celsius(20, "celsius")
```

```
## [1] 20
```

```
convertir_celsius(20, "foo")
```

```
## [1] 20
```

C'est un problème courant avec les fonctions qui utilisent des conditionnels. Nous discuterons de la manière de gérer cela dans la section suivante.

Débogage d'une fonction avec une logique conditionnelle

Une fonction nommée `check_negatives` est conçue pour analyser un vecteur de nombres en R et imprimer un message indiquant si le vecteur contient des nombres négatifs. Cependant, la fonction contient actuellement des erreurs de syntaxe.

```
check_negatives <- function(numbers) {
  x <- numbers

  if any(x < 0) {
    print("x contient des nombres négatifs")
  else {
    print("x ne contient pas de nombres négatifs")
}
```

Identifiez et corrigez les erreurs de syntaxe dans la fonction `check_negatives`. Après avoir corrigé la fonction, testez-la avec les vecteurs suivants pour vous assurer qu'elle fonctionne correctement : 1. `c(8, 3, -2, 5)` 2. `c(10, 20, 30, 40)`

Vérification des arguments avec des conditionnels

Lors de l'écriture de fonctions en R, il est souvent utile de s'assurer que les entrées fournies sont sensées et dans le domaine attendu. Sans vérifications appropriées, une fonction peut retourner des résultats incorrects ou échouer silencieusement, ce qui peut être source de confusion et de perte de temps pour le débogage. C'est là que la vérification des arguments intervient.

Considérez le scénario suivant avec notre fonction de conversion de température `convertir_celsius()`, qui convertit une température de Celsius en fahrenheit ou kelvin

```
convertir_celsius (30, "centigrade")
```

```
## [1] 30
```

Dans ce cas, l'utilisateur essaie de convertir une température de Kelvin en "centigrade". Mais notre fonction échoue silencieusement, retournant la valeur Celsius d'origine au lieu d'un message d'erreur. Cela est dû au fait que l'argument `convertir_en` n'est pas vérifié pour sa validité.

Pour améliorer notre fonction, nous pouvons introduire une vérification des arguments pour valider `convertir_de` et `convertir_en`. La fonction `stop()` en R nous permet de terminer l'exécution d'une fonction et d'imprimer un message d'erreur. Voici comment nous pouvons utiliser `stop()` pour vérifier des valeurs valides de `convertir_de` et `convertir_en` :

```
# Tester stop() en dehors d'une fonction, pour l'intégrer plus tard dans
conv_temp()
convertir_en <- "mauvaise échelle"

if (!(convertir_en %in% c("fahrenheit", "kelvin"))) {
  stop("convertir_en doit être fahrenheit ou kelvin")
}
```

```
Erreur : convertir_en doit être celsius, fahrenheit, ou kelvin
```

Intégrons cela dans notre fonction `convertir_celsius()` :

```
convertir_celsius <- function(celsius, convert_en) {
  if (!(convert_en %in% c("fahrenheit", "kelvin"))) {
    stop("convert_en doit être fahrenheit ou kelvin")
  }

  if (convert_en == "fahrenheit") {
    out <- (celsius * 9/5) + 32
  } else if (convert_to == "kelvin") {
    out <- celsius + 273.15
  } else {
    out <- celsius
  }
  return(out)
}
```

Notez que dans cette fonction mise à jour, il n'est plus nécessaire d'avoir une instruction `else`, car la fonction `stop()` mettra fin à l'exécution de la fonction si `convert_en` n'est pas l'une des trois valeurs valides. Ainsi, nous pouvons simplifier la fonction comme suit :

```
convertir_celsius <- function(celsius, convert_en) {
  if (!(convert_en %in%c("fahrenheit", "kelvin"))) {
    stop("convert_en doit être fahrenheit ou kelvin")
  }

  if (convert_en == "fahrenheit") {
    out <- (celsius * 9/5) + 32
  } else if (convert_en == "kelvin") {
    out <- celsius + 273.15
  }
  return(out)
}
```

Maintenant, si nous exécutons la commande problématique originale :

```
convertir_celsius(30, "centigrade")
```

La fonction s'arrêtera immédiatement et fournira un message d'erreur clair, indiquant que "centigrade" n'est pas une échelle de température reconnue.

Bien que la vérification des arguments améliore la fiabilité des fonctions, une utilisation excessive peut ralentir les performances et compliquer le code. Avec le temps, en examinant le code d'autres personnes et grâce à l'expérience, vous développerez un bon sens de l'équilibre à trouver entre rigueur, efficacité et clarté. Pour l'instant, notez qu'il est généralement bon de pencher vers plus de vérifications.

Exercice sur la vérification des arguments

Considérez la fonction `calculate_calories` que nous avons écrite plus tôt :

```
calculate_calories <- function(carb_grams = 0, protein_grams = 0, fat_grams =
0) {
  result <- (carb_grams * 4) + (protein_grams * 4) + (fat_grams * 9)
  return(result)
}
```

Écrire une fonction appelée `calculate_calories2()` qui est identique à `calculate_calories()` sauf qu'elle vérifie si les arguments `carb_grams`, `protein_grams` et `fat_grams` sont numériques. Si l'un d'eux n'est pas numérique, la fonction doit imprimer un message d'erreur en utilisant la fonction `stop()`.

```
calculate_calories2 <- function(carb_grams = 0, protein_grams = 0, fat_grams =
0) {

  # votre code ici

  return(result)
}
```

Conditionnels Vectorisés

Dans les exemples précédents, nous avons ajouté une condition sur une seule valeur, telle que l'argument `convert_en`. Maintenant, explorons comment construire des conditionnels basés sur un vecteur de valeurs. De tels cas nécessitent une gestion spéciale car l'instruction `if` n'est pas vectorisée.

Par exemple, si nous voulions écrire une fonction pour classifier les lectures de température en hypothermie, normal ou fièvre, vous pourriez penser à utiliser une construction if-else comme ceci :

```
classify_temp <- function(temp) {
  if (temp < 36.5) {
    print("hypothermie")
  } else if (temp >= 36.5 & temp <= 37.5) {
    print("normal")
  } else if (temp > 37.5) {
    print("fièvre")
  }
}
```

Cela fonctionne sur une seule valeur :

```r
classify_temp(36)
```

```
## [1] "hypothermie"
```

Mais cela ne fonctionnera pas comme prévu sur un vecteur, car l'instruction `if` n'est pas vectorisée et évalue seulement le premier élément du vecteur. Par exemple :

```r
temp_vec <- c(36, 37, 38)
```

```r
classify_temp(temp_vec)  # Ceci ne fonctionnera pas correctement
```

```
Erreur dans if (temp < 35) { : la condition a une longueur > 1
```

Pour traiter l'ensemble du vecteur, vous devriez utiliser des fonctions vectorisées telles que `ifelse` ou `case_when` du package `dplyr`. Voici comment vous pouvez employer `ifelse` :

```r
classify_temp <- function(temp) {
  out <- ifelse(temp < 36.5, "hypothermie",
                ifelse(temp >= 36.5 & temp <= 37.5, "normal",
                       ifelse(temp > 37.5, "fièvre", NA)))
  return(out)
}

classify_temp(temp_vec)  # Ceci fonctionne comme prévu
```

```
## [1] "hypothermie" "normal"      "fièvre"
```

Pour une alternative plus propre et plus lisible, `dplyr::case_when` peut également être utilisé :

```r
classify_temp <- function(temp) {
 case_when(
    temp < 36.5 ~ "hypothermie",
    temp >= 36.5 & temp <= 37.5 ~ "normal",
    temp > 37.5 ~ "fièvre",
    TRUE ~ NA_character_
  )
}

classify_temp(temp_vec)  # Ceci fonctionne aussi comme prévu
```

```
## [1] "hypothermie" "normal"      "fièvre"
```

Cette fonction fonctionne sans problème avec les jeux de données aussi :

```r
NHSRdatasets::synthetic_news_data %>%
  select(temp) %>%
  mutate(temp_class = classify_temp(temp))
```

```
## # A tibble: 1,000 × 2
##     temp temp_class
##    <dbl> <chr>
##  1  36.8 normal
##  2  35   hypothermie
##  3  36.2 hypothermie
##  4  36.9 normal
##  5  36.4 hypothermie
##  6  35.3 hypothermie
##  7  35.6 hypothermie
##  8  37.2 normal
##  9  35.5 hypothermie
## 10  35.3 hypothermie
## # ℹ 990 more rows
```

Pratique pour classifier la dose d'Isoniazide

Appliquons ces connaissances à un cas pratique. Considérez la tentative suivante pour écrire une fonction qui calcule les dosages du médicament isoniazide pour des adultes pesant plus de 30kg :

```r
calculer_dosage_isoniazide <- function(poids) {
  if (poids < 30) {
    stop("Le poids doit être au moins de 30 kg.")
  } else if (poids <= 35) {
    return(150)
  } else if (poids <= 45) {
    return(200)
  } else if (poids <= 55) {
    return(300)
  } else if (poids <= 70) {
    return(300)
  } else {
    return(300)
  }
}
```

Cette fonction échoue avec un vecteur de poids. Votre tâche est d'écrire une nouvelle fonction `calculate_isoniazid_dosage2()` qui peut gérer les entrées vectorielles. Pour vous assurer que tous les poids sont au-dessus de 30 kg, vous utiliserez la fonction `any()` dans votre vérification des erreurs.

Voici une ébauche pour vous aider à démarrer :

```r
calculate_isoniazid_dosage2 <- function(weight) {
  if (any(weight < 30)) stop("Tous les poids doivent être au moins de 30 kg.")

  # Votre code ici

  return(out)
}

calculate_isoniazid_dosage2(c(30, 40, 50, 100))
```

```
## Error in calculate_isoniazid_dosage2(c(30, 40, 50, 100)): object 'out' not
found
```

Où stocker vos fonctions

Lorsque vous écrivez des scripts en R, décider où stocker vos fonctions est une considération importante pour maintenir un code propre et un flux de travail. Voici quelques stratégies clés :

1) Haut du Script

Placer les fonctions en haut de votre script est une pratique simple et couramment utilisée.

2) Script Séparé qui est utilisé comme source

À mesure que votre projet grandit, vous pouvez avoir plusieurs fonctions. Dans de tels cas, les stocker dans un script R séparé peut permettre de garder votre script d'analyse principal ordonné. Vous pouvez ensuite 'sourcer' ce script pour charger les fonctions.

```r
# Sourcer un script séparé
source("chemin_vers_votre_script_de_fonctions.R")
```

3) GitHub Gist

Pour des fonctions que vous réutilisez fréquemment ou souhaitez partager avec la communauté, les stocker dans un GitHub Gist est une bonne option. Créez un compte sur github, puis créez un gist public à https://gist.github.com/. Ensuite, vous pouvez copier-coller votre fonction dans le gist. Finalement, vous pouvez obtenir l'URL du gist et la sourcer dans votre script R en utilisant la fonction `source_gist()` du package `devtools` :

```r
# Sourcer depuis un GitHub Gist
pacman::p_load(devtools)
devtools::source_gist("https://gist.github.com/kendavidn/a5e1ce486910e6b2dc77a5b
```

Lorsque vous exécutez le code ci-dessus, cela définira une nouvelle fonction appelée `hello_from_gist()` que nous avons créée pour cette leçon.

```
hello_from_gist("Étudiant")
```

```
## [1] "Hello Étudiant! This is a function from a gist!"
```

Vous pouvez voir le code en allant directement à l'URL : https://gist.github.com/kendavidn/a5e1ce486910e6b2dc77a5b6bddf87d0.

Le code dans le gist peut être mis à jour à tout moment, et les changements seront reflétés dans votre script lorsque vous le sourcerez à nouveau.

4) Package

Comme précédemment mentionné, les fonctions peuvent également être stockées dans des packages. C'est une option plus avancée qui nécessite la connaissance du développement de packages R. Pour plus d'informations, voir le manuel Writing R Extensions.

Conclusion !

Félicitations pour avoir suivi la leçon !

Vous avez maintenant les éléments de base pour créer des fonctions personnalisées qui automatisent les tâches répétitives dans vos flux de travail R. Bien sûr, il y a encore beaucoup à apprendre sur les fonctions, mais vous avez maintenant les fondations sur lesquelles construire.

Corrigés

Fonction Âge en Mois

```
annees_en_mois <- function(annees) {
  mois <- annees * 12
  return(mois)
}

# Test
annees_en_mois(12)
```

```
## [1] 144
```

Fonction Celsius en Fahrenheit

```r
celsius_en_fahrenheit <- function(celsius) {
  fahrenheit <- celsius * 1.8 + 32
  return(fahrenheit)
}

# Test
beaver1 %>%
  select(temp) %>%
  mutate(Fahrenheit = celsius_en_fahrenheit(temp))
```

```
##      temp Fahrenheit
## 1   36.33     97.394
## 2   36.34     97.412
## 3   36.35     97.430
## 4   36.42     97.556
## 5   36.55     97.790
## 6   36.69     98.042
## 7   36.71     98.078
## 8   36.75     98.150
## 9   36.81     98.258
## 10  36.88     98.384
## 11  36.89     98.402
## 12  36.91     98.438
## 13  36.85     98.330
## 14  36.89     98.402
## 15  36.89     98.402
## 16  36.67     98.006
## 17  36.50     97.700
## 18  36.74     98.132
## 19  36.77     98.186
## 20  36.76     98.168
## 21  36.78     98.204
## 22  36.82     98.276
## 23  36.89     98.402
## 24  36.99     98.582
## 25  36.92     98.456
## 26  36.99     98.582
## 27  36.89     98.402
## 28  36.94     98.492
## 29  36.92     98.456
## 30  36.97     98.546
## 31  36.91     98.438
## 32  36.79     98.222
## 33  36.77     98.186
## 34  36.69     98.042
## 35  36.62     97.916
## 36  36.54     97.772
```

```
## 37  36.55     97.790
## 38  36.67     98.006
## 39  36.69     98.042
## 40  36.62     97.916
## 41  36.64     97.952
## 42  36.59     97.862
## 43  36.65     97.970
## 44  36.75     98.150
## 45  36.80     98.240
## 46  36.81     98.258
## 47  36.87     98.366
## 48  36.87     98.366
## 49  36.89     98.402
## 50  36.94     98.492
## 51  36.98     98.564
## 52  36.95     98.510
## 53  37.00     98.600
## 54  37.07     98.726
## 55  37.05     98.690
## 56  37.00     98.600
## 57  36.95     98.510
## 58  37.00     98.600
## 59  36.94     98.492
## 60  36.88     98.384
## 61  36.93     98.474
## 62  36.98     98.564
## 63  36.97     98.546
## 64  36.85     98.330
## 65  36.92     98.456
## 66  36.99     98.582
## 67  37.01     98.618
## 68  37.10     98.780
## 69  37.09     98.762
## 70  37.02     98.636
## 71  36.96     98.528
## 72  36.84     98.312
## 73  36.87     98.366
## 74  36.85     98.330
## 75  36.85     98.330
## 76  36.87     98.366
## 77  36.89     98.402
## 78  36.86     98.348
## 79  36.91     98.438
## 80  37.53     99.554
## 81  37.23     99.014
## 82  37.20     98.960
## 83  37.25     99.050
## 84  37.20     98.960
## 85  37.21     98.978
## 86  37.24     99.032
## 87  37.10     98.780
## 88  37.20     98.960
## 89  37.18     98.924
## 90  36.93     98.474
## 91  36.83     98.294
## 92  36.93     98.474
```

```
## 93  36.83     98.294
## 94  36.80     98.240
## 95  36.75     98.150
## 96  36.71     98.078
## 97  36.73     98.114
## 98  36.75     98.150
## 99  36.72     98.096
## 100 36.76     98.168
## 101 36.70     98.060
## 102 36.82     98.276
## 103 36.88     98.384
## 104 36.94     98.492
## 105 36.79     98.222
## 106 36.78     98.204
## 107 36.80     98.240
## 108 36.82     98.276
## 109 36.84     98.312
## 110 36.86     98.348
## 111 36.88     98.384
## 112 36.93     98.474
## 113 36.97     98.546
## 114 37.15     98.870
```

Fonction IMC

```r
calculer_imc <- function(poids, taille) {
  imc <- poids / (taille^2)
  return(imc)
}

# Test
library(medicaldata)
medicaldata::smartpill %>%
  as_tibble() %>%
  select(Weight, Height) %>%
  mutate(IMC = calculer_imc(Weight, Height))
```

```
## # A tibble: 95 × 3
##    Weight Height     IMC
##     <dbl>  <dbl>   <dbl>
##  1  102.    183. 0.00305
##  2  102.    180. 0.00314
##  3   68.0   180. 0.00209
##  4   69.9   175. 0.00227
##  5   44.9   152. 0.00193
##  6   94.8   185. 0.00276
##  7   86.2   188. 0.00244
##  8   76.2   165. 0.00280
##  9   74.4   173. 0.00249
## 10   64.9   170. 0.00224
## # ℹ 85 more rows
```

Pratique avec l'Argument ...

```
calculate_calories <- function(carb_grams, protein_grams, fat_grams, ...) {
  result <- (carb_grams * 4) + (protein_grams * 4) + (fat_grams * 9)
  result_formatted <- format(result, ...)
  return(result)
}
```

Débogage d'une Fonction avec Logique Conditionnelle

```
verifier_negatifs <- function(nombres) {
  if (any(nombres < 0)) {
    print("x contient des nombres négatifs")
  } else {
    print("x ne contient pas de nombres négatifs")
  }
}

# Test
verifier_negatifs(c(8, 3, -2, 5))
```

```
## [1] "x contient des nombres négatifs"
```

```
verifier_negatifs(c(10, 20, 30, 40))
```

```
## [1] "x ne contient pas de nombres négatifs"
```

Pratique de Vérification des Arguments

```
calculate_calories2 <- function(carb_grams = 0, protein_grams = 0, fat_grams =
0) {

  if (!is.numeric(carb_grams)) {
    stop("carb_grams must be numeric")
  }

  if (!is.numeric(protein_grams)) {
    stop("protein_grams must be numeric")
  }

  if (!is.numeric(fat_grams)) {
    stop("fat_grams must be numeric")
  }


  result <- (carb_grams * 4) + (protein_grams * 4) + (fat_grams * 9)
  return(result)
}
```

Pratique de classification du dosage de l'Isoniazide

```
calculer_dosage_isoniazide2 <- function(poids) {
  if (any(poids < 30)) stop("Tous les poids doivent être au moins de 30 kg.")

  dosage <- case_when(
    poids <= 35 ~ 150,
    poids <= 45 ~ 200,
    poids <= 55 ~ 300,
    poids <= 70 ~ 300,
    TRUE ~ 300
  )
  return(dosage)
}

calculer_dosage_isoniazide2(c(30, 40, 50, 100))
```

```
## [1] 150 200 300 300
```

Contributeurs

Les membres de l'équipe suivants ont contribué à cette leçon :

DANIEL CAMARA

Data Scientist at the GRAPH Network and fellowship as Public Health

researcher at Fiocruz, Brazil
Passionate about lots of things, especially when it involves people leading lives with more equality and freedom

EDUARDO ARAUJO

Student at Universidade Tecnologica Federal do Parana
Passionate about reproducible science and education

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

Références

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

- Barnier, Julien. “Introduction à R et au tidyverse.” Consulté le 23 mai 2022. https://juba.github.io/tidyverse
- Wickham, Hadley; Grolemund, Garrett. “R for Data Science.” Consulté le 25 mai 2022. https://r4ds.had.co.nz/