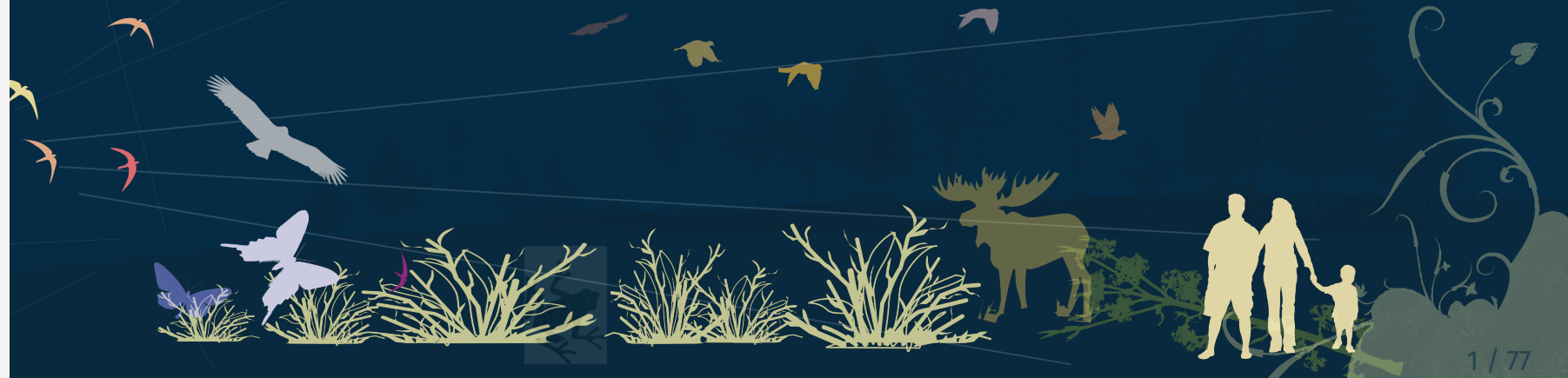




Atelier 5: Programmation en R

Série d'ateliers R du CSBQ

Centre des sciences de la biodiversité du Québec



À propos de cet atelier



Objectifs d'apprentissage

1. Prendre connaissance de **structures de contrôle**
2. Écrire des fonctions dans R
3. Réduire le temps d'exécution des codes
4. Paquets R utiles

Rappel

Objets

Vecteurs

Vecteurs numériques

```
num.vector ← c(1, 4, 3, 98, 32, -76, -4)
```

```
num.vector  
# [1] 1 4 3 98 32 -76 -4
```

Tableaux de données

vecteurs

```
siteID ← c("A1.01", "A1.02", "B1.01", "B1.02")
soil_pH ← c(5.6, 7.3, 4.1, 6.0)
num.sp ← c(17, 23, 15, 7)
treatment ← c("Fert", "Fert", "No_fert", "No_fert")
```

On combine ces vecteurs à l'aide de la fonction `data.frame`

```
my.first.df ← data.frame(siteID, soil_pH, num.sp, treatment)
```

```
my.first.df
```

```
#   siteID soil_pH num.sp treatment
# 1  A1.01    5.6    17      Fert
# 2  A1.02    7.3    23      Fert
# 3  B1.01    4.1    15  No_fert
# 4  B1.02    6.0     7  No_fert
```

Listes

```
my.first.list ← list(siteID, soil_pH, num.sp, treatment)
```

```
my.first.list
# [[1]]
# [1] "A1.01" "A1.02" "B1.01" "B1.02"
#
# [[2]]
# [1] 5.6 7.3 4.1 6.0
#
# [[3]]
# [1] 17 23 15 7
#
# [[4]]
# [1] "Fert"      "Fert"      "No_fert" "No_fert"
```


Contrôle de flux

Contrôle de flux

En programmation, le contrôle de flux est simplement l'ordre dans lequel le programme est exécuté.

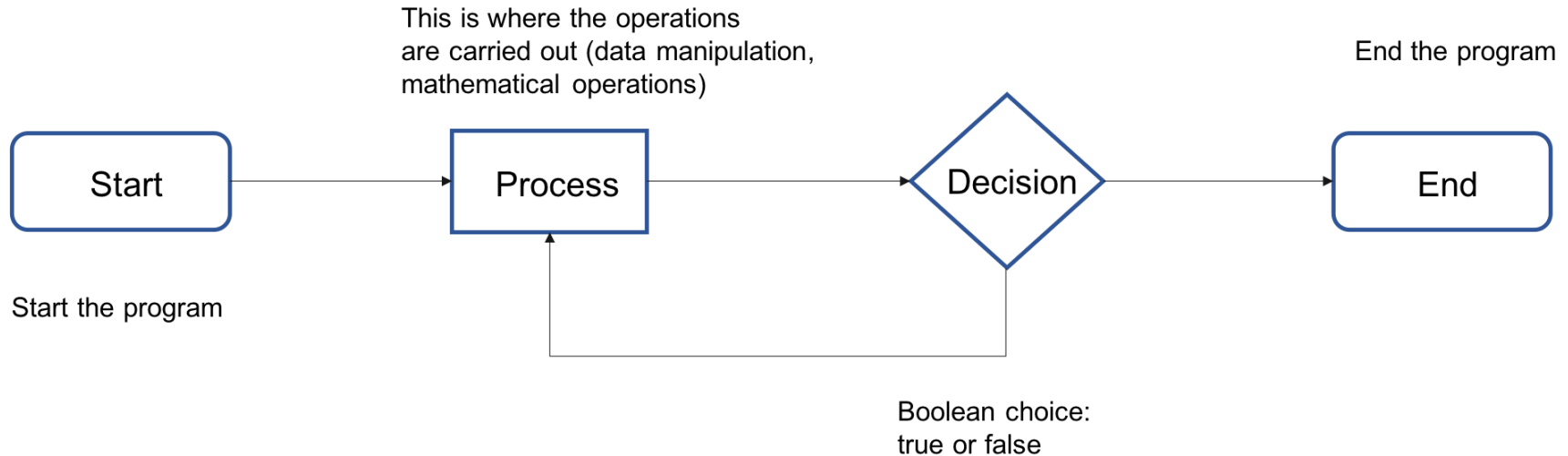
Pourquoi est-il avantageux de structurer nos programmes?

- On **réduit la complexité** et la durée de la tâche;
- Une structure logique **améliore la clarté** du code;
- **Plusieurs programmeurs peuvent alors travailler sur un même programme.**

Tout ceci augmente la productivité!

Contrôle de flux

On peut utiliser des organigrammes pour planifier et représenter la structure de programmes.



Représenter la structure

Les deux composantes de base de programmation sont:

La sélection

Exécuter des commandes
conditionnellement en utilisant:

```
if  
if else
```

L'itération

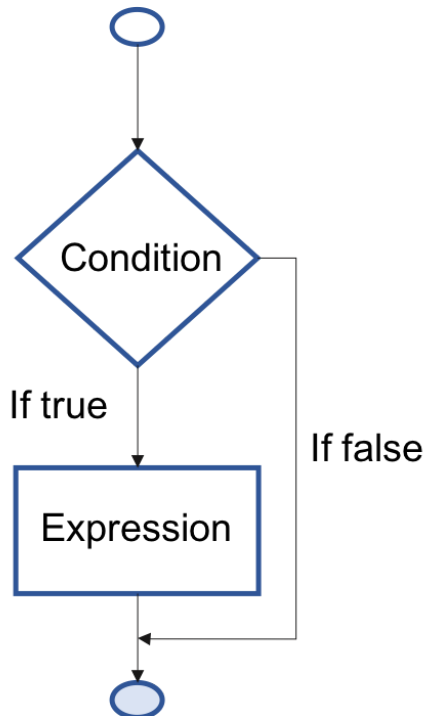
Répéter l'exécution d'une
commande **en boucle** tant qu'une
condition n'est pas satisfaite.

```
for  
while  
repeat
```

Prise de décisions

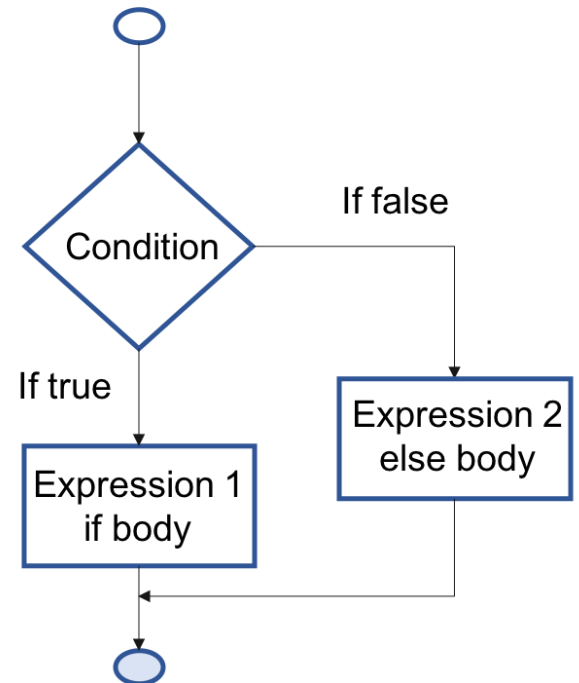
Commande **if**

```
if(condition) {  
    expression  
}
```



Commande **if else**

```
if(condition) {  
    expression 1  
} else {  
    expression 2  
}
```



Comment peut-on tester plus qu'une condition?

- `if` et `if else` testent une seule condition
- On peut aussi utiliser la commande `ifelse` pour:
 - tester un vecteur de conditions;
 - exécuter une opération selon certaines conditions.

```
a ← 1:10  
ifelse(a > 5, "yes", "no")  
  
a ← (-4):5  
sqrt(ifelse(a ≥ 0, a, NA))
```

Commandes **if** **else** nichées

```
if (test_expression1) {  
    statement1  
} else if (test_expression2) {  
    statement2  
} else if (test_expression3) {  
    statement3  
} else {  
    statement4  
}
```

Exercice 1



```
Minou ← "chat"  
Pitou ← "chien"  
Filou ← "chat"  
animaux ← c(Minou, Pitou, Filou)
```

1. Utilisez une commande `if` pour afficher `"meow"` si `Minou` est un `"chat"`.
2. Utilisez une commande `if else` pour afficher `"woof"` si un objet a la valeur `"chien"`, et `"meow"` si non. Essayez ceci sur les objets `Pitou` et `Filou`.
3. Utilisez la fonction `ifelse` pour afficher `"woof"` pour les `animaux` qui sont des chiens et `"meow"` pour les `animaux` qui sont des chats.

Exercice 1 - Solution



1- Utilisez une commande `if` pour afficher `"meow"` si `Minou` est un `"chat"`.

```
if(Minou == 'chat') {  
    print("meow")  
}  
# [1] "meow"
```

2- Utilisez une commande `if` `else` pour afficher `"woof"` si un objet a la valeur `"chien"`, et `"meow"` si non. Essayez ceci sur les objets `Pitou` et `Filou`.

```
x = Minou  
# x = Pitou  
if(x == 'chat') {  
    print("meow")  
} else {  
    print("woof")  
}  
# [1] "meow"
```

Exercice 1 - Solution



3- Utilisez la fonction `ifelse` pour afficher "woof" pour les animaux qui sont des chiens et "meow" pour les animaux qui sont des chats.

```
animaux ← c(Minou, Pitou, Filou)

ifelse(animaux == 'chien', "woof", "meow")
# [1] "meow" "woof" "meow"
```

Ou

```
for(val in 1:3) {
  if(animaux[val] == "chat") {
    print("meow")
  } else if(animaux[val] == "chien") {
    print("woof")
  } else print("quoi?")
}

# [1] "meow"
# [1] "woof"
# [1] "meow"
```

Attention à la syntaxe des expressions!

Les accolades `{}` indiquent à R d'exécuter une expression au complet. Par exemple:

```
if ((2 + 1) = 4) print("Les maths, c'est logique!.")  
else print("Houston, on a un problème.")
```

Ceci ne fonctionne pas, parce que R évalue la première ligne sans reconnaître que votre expression continue avec une commande `else` sur la prochaine ligne!

Utilisez plutôt:

```
if (2+2 == 4) {  
  print("Les maths, c'est logique!")  
} else {  
  print("Houston, on a un problème.")  
}  
# [1] "Les maths, c'est logique!"
```

Rappel: opérateurs logiques

| Opérateur | Signification |
|------------------------|--------------------------|
| $=$ | égal à |
| \neq | pas égal à |
| $<$ | plus petit que |
| \leq | plus petit que ou égal à |
| $>$ | plus grand que |
| \geq | plus grand que ou égal à |
| $x \& y$ | x ET y |
| $x y$ | x OU y |
| <code>isTRUE(x)</code> | est-ce que X est vrai? |

Itération

Une boucle permet de répéter une ou plusieurs opérations.

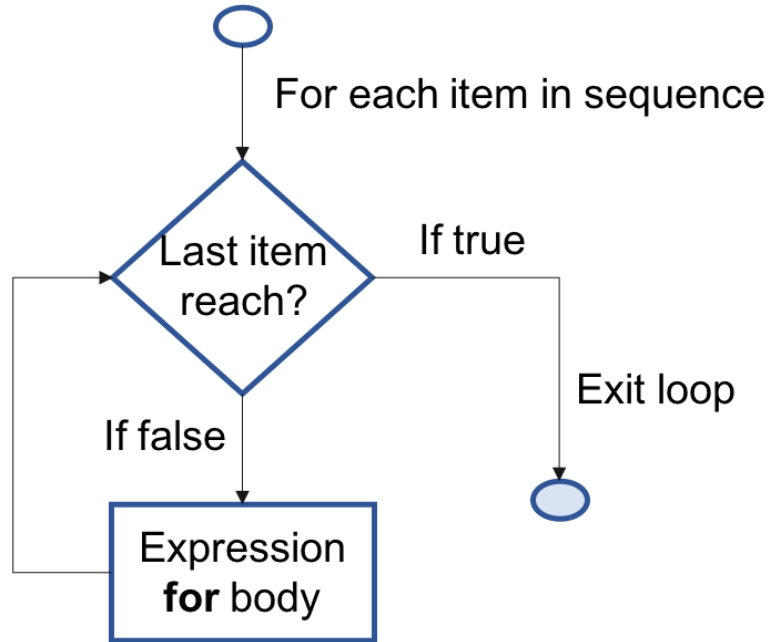
Les boucles sont utiles pour:

- faire quelque chose pour chaque élément d'un objet.
- faire quelque chose jusqu'à la fin des données à traiter.
- faire quelque chose pour chaque fichier dans un répertoire.
- faire quelque chose qui peut échouer, jusqu'à ce que ça fonctionne.
- faire des calculs itératifs jusqu'à convergence.

Boucles **for**

Une boucle **for** exécute un nombre fixe d'itérations:

```
for(i in séquence) {  
    expression  
}
```



for loop

La lettre `i` peut être remplacée par n'importe quelle nom de variable, liste de vecteurs, etc.

Essayez les commandes ci-dessous et observez les résultats:

```
for (a in c("Bonjour", "programmeurs", "en R")) {  
  print(a)  
}
```

```
for (z in 1:30) {  
  a ← rnorm(n = 1, mean = 5, sd = 2)  
  print(a)  
}
```

```
elements ← list(1:3, 4:10)  
for (element in elements) {  
  print(element)  
}
```

Boucle **for**

Dans le prochain exemple, R évalue l'expression 5 fois:

```
for(i in 1:5) {  
  expression  
}
```

Dans cet exemple, chaque instance de **m** est remplacé par chaque chiffre entre 1:10, jusqu'au dernier élément de la séquence:

```
for(m in 1:10) {  
  print(m*2)  
}
```

[1] 2

[1] 4

[1] 6

[1] 8

[1] 10

[1] 12

[1] 14

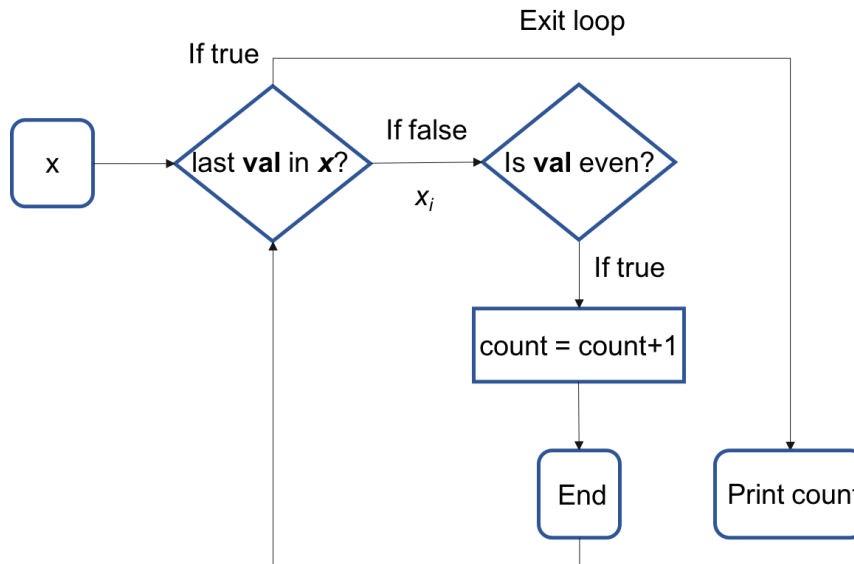
[1] 16

[1] 18

[1] 20

Boucle **for**

```
x ← c(2,5,3,9,6)
count ← 0
for (val in x) {
  if(val %% 2 == 0) {
    count = count+1
  }
}
print(count)
```



Boucle `for`

Les boucles `for` sont souvent utilisées pour exécuter des opérations successivement sur un jeu de données. Nous utiliserons des boucles pour exécuter des fonctions sur le jeu de données `C02`, qui est intégré dans R.

```
data(C02) # ceci charge le jeu de données intégré dans R
```

```
for (i in 1:length(C02[,1])) { # pour chaque ligne du jeu de données C02
  print(C02$conc[i]) # affiche les concentrations de C02
}
```

```
for (i in 1:length(C02[,1])) { # pour chaque ligne du jeu de données C02
  if(C02$Type[i] == "Quebec") { # si le type est "Quebec"
    print(C02$conc[i]) # affiche les concentrations de C02
  }
}
```

Boucle `for`

Truc 1. Pour exécuter une boucle sur chaque ligne d'un jeu de donnée, on utilise la fonction `nrow()`

```
for (i in 1:nrow(CO2)) { # pour chaque ligne du jeu de données CO2
  print(CO2$conc[i]) # affiche les concentrations de CO2
}
```

Truc 2. On peut aussi itérer des opérations sur les éléments d'une colonne.

```
for (p in CO2$conc) { # pour chacune des valeurs de concentration de CO2
  print(p) # afficher cette valeur
}
```

Boucle **for**

L'expression dans la boucle peut contenir plusieurs lignes de commandes différentes.

```
for (i in 4:5) { # pour i de 4 à 5
  print(colnames(CO2)[i])
  print(mean(CO2[,i])) # affiche les moyennes de cette colonne
}
```

Sortie:

```
# [1] "conc"
# [1] 435
# [1] "uptake"
# [1] 27.2131
```

Boucles `for` nichées

Dans certains cas, des boucles nichées peuvent être utiles pour accomplir une tâche. Dans ce cas, il est important d'utiliser un nom de variable d'itération différent pour chaque boucle. Ici, on utilise `i` et `n`:

```
for (i in 1:3) {  
    for (n in 1:3) {  
        print (i*n)  
    }  
}
```

```
# Sortie  
# [1] 1  
# [1] 2  
# [1] 3  
# [1] 2  
# [1] 4  
# [1] 6  
# [1] 3  
# [1] 6  
# [1] 9
```

Encore mieux: la famille `apply()`

La famille de fonctions `apply()` consistent de fonctions vectorisées qui permettent **d'éviter de créer des boucles de façon explicite**.

`apply()` applique des fonctions sur une **matrice**.

```
(hauteur ← matrix(c(1:10, 21:30),  
                  nrow = 5,  
                  ncol = 4))
```

```
#      [,1] [,2] [,3] [,4]  
# [1,]    1    6   21   26  
# [2,]    2    7   22   27  
# [3,]    3    8   23   28  
# [4,]    4    9   24   29  
# [5,]    5   10   25   30
```

```
apply(X = hauteur,  
      MARGIN = 1,  
      FUN = mean)  
# [1] 13.5 14.5 15.5 16.5 17.5
```

```
?apply
```

lapply()

`lapply()` applique une fonction sur chaque élément d'une `liste`.

`lapply()` fonctionne aussi sur d'autres objets, comme des **trames de données** ("**dataframe**") ou des **vecteurs**.

La sortie est une `liste` (d'où le "l" dans `lapply`) ayant le même nombre d'éléments que l'objet d'entrée.

```
SimulatedData <- list(  
  SimpleSequence = 1:4,  
  Norm10 = rnorm(10),  
  Norm20 = rnorm(20, 1),  
  Norm100 = rnorm(100, 5))  
  
# Applique mean() sur chaque élém  
lapply(SimulatedData, mean)
```

```
# $SimpleSequence  
# [1] 2.5  
#  
# $Norm10  
# [1] 0.05189392  
#  
# $Norm20  
# [1] 1.033075  
#  
# $Norm100  
# [1] 4.877695
```

sapply()

`sapply()` est une fonction 'wrapper' pour `lapply()`, qui produit une sortie simplifiée en `vecteur`, au lieu d'une `liste`.

```
SimulatedData <- list(SimpleSequence = 1:4,  
  Norm10 = rnorm(10),  
  Norm20 = rnorm(20, 1),  
  Norm100 = rnorm(100, 5))
```

```
# Applique mean() sur chaque élément de la liste  
sapply(SimulatedData, mean)
```

| # SimpleSequence | Norm10 | Norm20 | Norm100 |
|------------------|------------|-----------|-----------|
| # 2.5000000 | -0.2114442 | 0.8414591 | 4.9776368 |

mapply()

`mapply()` est une version multivariée de `sapply()`.

`mapply()` applique une fonction sur le premier élément de chaque argument, ensuite sur le deuxième élément, et ainsi de suite. Par exemple:

```
lilySeeds ← c(80, 65, 89, 23, 21)
poppySeeds ← c(20, 35, 11, 77, 79)
```

```
# Output
mapply(sum, lilySeeds, poppySeeds)
# [1] 100 100 100 100 100
```

tapply()

`tapply()` applique une fonction sur des sous-ensembles d'un vecteur.

`tapply()` est surtout utilisé quand un jeu de données contient différents groupes (*i.e.* niveaux/facteurs), et lorsqu'on veut appliquer une fonction sur chaque groupe.

```
head(mtcars)
```

| # | mpg | cyl | disp | hp | drat | wt | qsec | vs | am | gear | carb |
|---------------------|------|-----|------|-----|------|-------|-------|----|----|------|------|
| # Mazda RX4 | 21.0 | 6 | 160 | 110 | 3.90 | 2.620 | 16.46 | 0 | 1 | 4 | 4 |
| # Mazda RX4 Wag | 21.0 | 6 | 160 | 110 | 3.90 | 2.875 | 17.02 | 0 | 1 | 4 | 4 |
| # Datsun 710 | 22.8 | 4 | 108 | 93 | 3.85 | 2.320 | 18.61 | 1 | 1 | 4 | 1 |
| # Hornet 4 Drive | 21.4 | 6 | 258 | 110 | 3.08 | 3.215 | 19.44 | 1 | 0 | 3 | 1 |
| # Hornet Sportabout | 18.7 | 8 | 360 | 175 | 3.15 | 3.440 | 17.02 | 0 | 0 | 3 | 2 |
| # Valiant | 18.1 | 6 | 225 | 105 | 2.76 | 3.460 | 20.22 | 1 | 0 | 3 | 1 |

```
# obtient la moyenne de hp par groupe de cylindres
```

```
tapply(mtcars$hp, mtcars$cyl, FUN = mean)
```

```
#           4           6           8
# 82.63636 122.28571 209.21429
```

Exercice 2



Votre outil pour mesurer le l'absorption de CO₂ n'était pas bien calibré aux sites situés au Québec, et toutes les mesures sont donc 2 unités trop élevées.

1. Utilisez une boucle pour corriger les mesures pour tous les sites aux Québec.
2. Utilisez une méthode vectorisée pour calculer la moyenne de l'absorption de CO₂ dans les deux groupes de sites.

Exercice 2: Solution



1. Utiliser `for` et `if` pour corriger les mesures:

```
for (i in 1:dim(CO2)[1]) {  
  if(CO2$Type[i] == "Quebec") {  
    CO2$uptake[i] ← CO2$uptake[i] - 2  
  }  
}
```

1. Utiliser `tapply()` pour calculer la moyenne de chaque groupe de sites:

```
tapply(CO2$uptake, CO2$Type, mean)  
#      Quebec Mississippi  
#    31.54286    20.88333
```

Modifications aux boucles

Habituellement, les boucles itèrent successivement jusqu'à leur fin.

Il est parfois intéressant de modifier ce comportement.

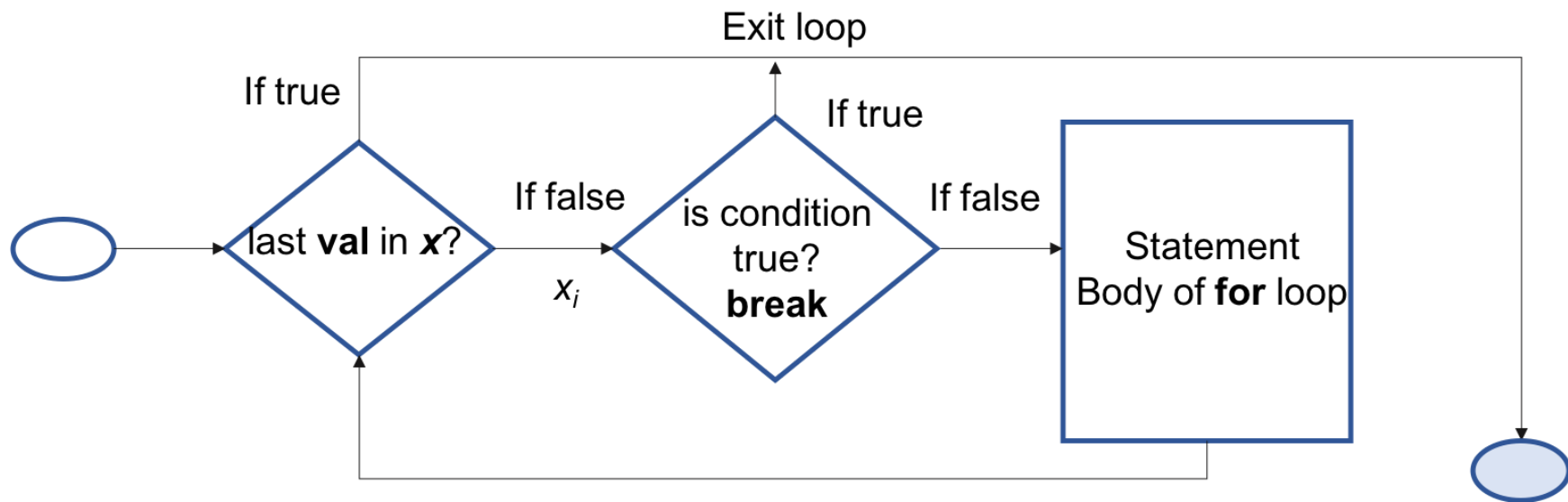
Par exemple, on peut arrêter l'exécution de la boucle quand une certaine condition est satisfaite ou quand l'itération a atteint un certain élément.

On peut aussi sauter certains éléments selon certaines conditions, ou arrêter l'itération pour passer à la boucle suivante.

Pour ceci, on introduit `break`, `next` and `while`.

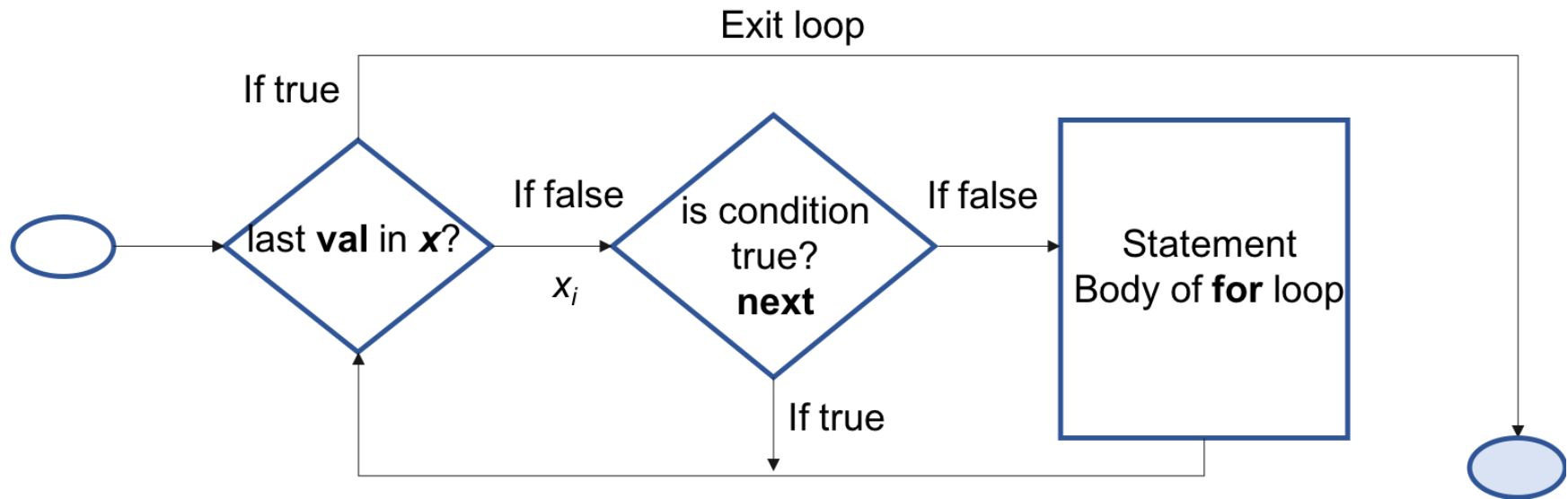
Modifier l'itération: `break`

```
for(val in x) {  
  if(condition) { break }  
  statement  
}
```



Modifier l'itération: **next**

```
for(val in x) {  
  if(condition) { next }  
  statement  
}
```



Modifier l'itération: `next`

Affiche les concentrations CO_2 pour les traitements “chilled” et garder le compte du nombre d'itérations accomplies.

```
count ← 0

for (i in 1:nrow(CO2)) {
  if (CO2$Treatment[i] == "nonchilled") next
  # Passer à l'itération suivante si c'est "nonchilled"
  count ← count + 1
  print(CO2$conc[i])
}
print(count) # Affiche le nombre d'itérations accomplies.
```

```
# [1] 42
```

```
sum(CO2$Treatment == "nonchilled")
# [1] 42
```


Modifier l'itération: `break`

On pourrait aussi accomplir ceci avec une boucle `repeat` et `break`:

```
count ← 0
i ← 0
repeat {
  i ← i + 1
  if (CO2$Treatment[i] == "nonchilled") next # sauter cette itération
  count ← count + 1
  print(CO2$conc[i])
  if (i == nrow(CO2)) break          # rompre la boucle
}
print(count)
```

Modifier l'itération: `while`

On pourrait aussi utiliser une boucle `while`:

```
i ← 0
count ← 0
while (i < nrow(CO2))
{
  i ← i + 1
  if (CO2$Treatment[i] == "nonchilled") next # sauter cette itération
  count ← count + 1
  print(CO2$conc[i])
}
print(count)
```

Exercice 3



Votre outil pour mesurer la concentration ne fonctionne pas correctement.

Aux sites situés au Mississippi, les concentrations de moins de 300 sont bien mesurées, mais les concentrations de plus de 300 étaient surestimées par 20 unités!

Votre *mission* est d'écrire une boucle pour corriger ces mesures pour les sites du Mississippi.

Truc. Assurez-vous de charger les données originales avant de faire l'exercice:

```
data(CO2)
```

Exercise 3: Solution



```
for (i in 1:nrow(CO2)) {  
  if(CO2$Type[i] == "Mississippi") {  
    if(CO2$conc[i] < 300) next  
    CO2$conc[i] ← CO2$conc[i] - 20  
  }  
}
```

Note: On peut écrire ceci de façon plus claire et concise:

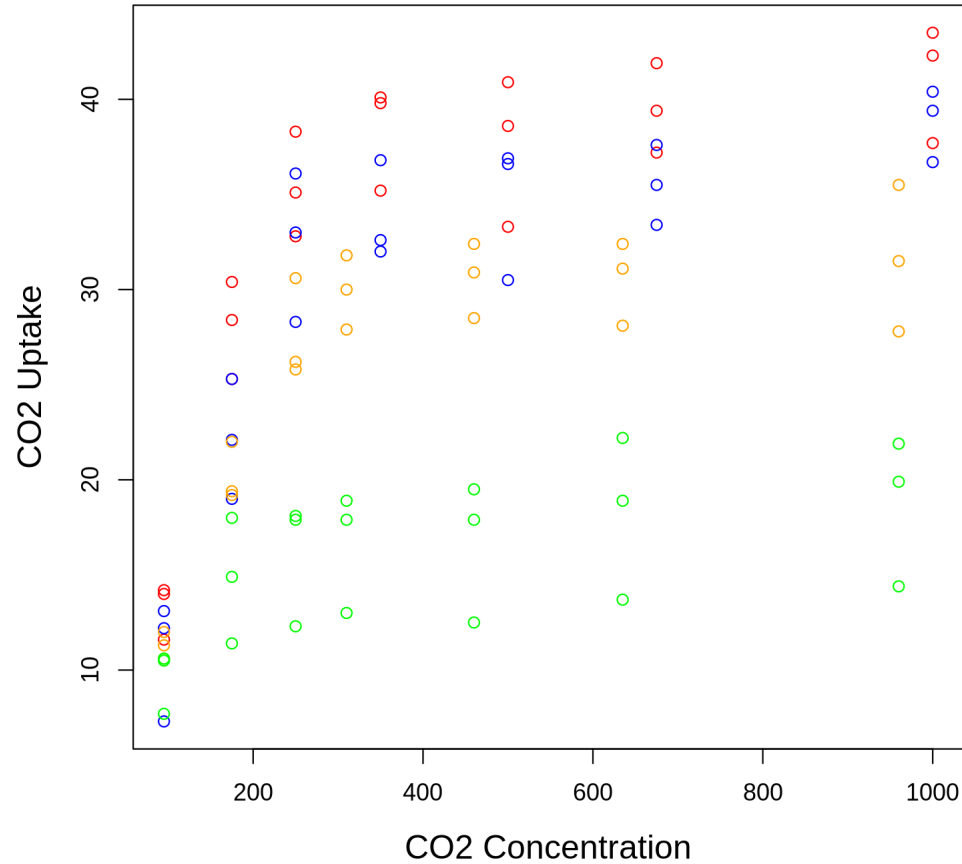
```
for (i in 1:nrow(CO2)) {  
  if(CO2$Type[i] == "Mississippi" && CO2$conc[i] ≥ 300) {  
    CO2$conc[i] ← CO2$conc[i] - 20  
  }  
}
```

Visualization de données avec `for` et `if`

Créons un graphique **uptake** vs **concentration** avec des points de couleurs différentes, où chaque couleur est associé à un type (*Quebec* ou *Mississippi*) et à un **treatment** (*chilled* ou *nonchilled*).

```
plot(x = C02$conc, y = C02$uptake, type = "n", cex.lab=1.4,
     xlab = "C02 concentration", ylab = "C02 uptake")
# Type "n" dit à R de ne pas afficher les points
for (i in 1:length(C02[,1])) {
  if (C02$Type[i] == "Quebec" & C02$Treatment[i] == "nonchilled") {
    points(C02$conc[i], C02$uptake[i], col = "red")
  }
  if (C02$Type[i] == "Quebec" & C02$Treatment[i] == "chilled") {
    points(C02$conc[i], C02$uptake[i], col = "blue")
  }
  if (C02$Type[i] == "Mississippi" & C02$Treatment[i] == "nonchilled") {
    points(C02$conc[i], C02$uptake[i], col = "orange")
  }
  if (C02$Type[i] == "Mississippi" & C02$Treatment[i] == "chilled") {
    points(C02$conc[i], C02$uptake[i], col = "green")
  }
}
```

Create a plot using `for` loop and `if`



Exercice 4



Créez un graphique **concentration** vs **uptake**, où chaque plante est représentée par des points de **couleurs** différentes.

Points bonus si vous utilisez des boucles nichées!

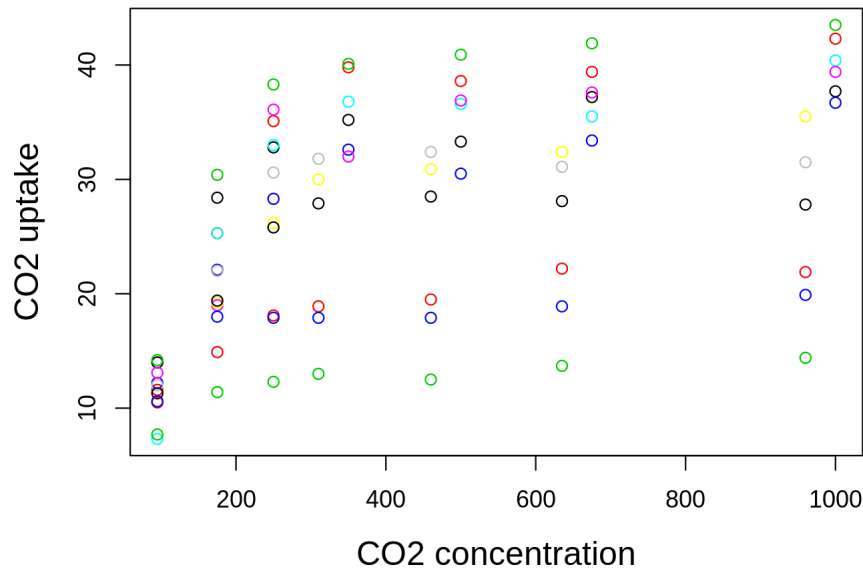
Exercise 4: Solution



```
plot(x = CO2$conc, y = CO2$uptake, type = "n", cex.lab=1.4,  
     xlab = "CO2 concentration", ylab = "CO2 uptake")
```

```
plants ← unique(CO2$Plant)
```

```
for (i in 1:nrow(CO2)){  
  for (p in 1:length(plants)) {  
    if (CO2$Plant[i] == plants[p]) {  
      points(CO2$conc[i], CO2$uptake[i], col = p)  
    }  
  }  
}
```



Écrire des fonctions

Pourquoi écrire une fonction??

Le gros du travail dans R est fait par des fonctions. Elles sont utiles pour:

1. Répéter une même tâche, mais en changeant ses paramètres;
2. Rendre votre code plus lisible;
3. Rendre votre code plus facile à modifier et à maintenir;
4. Partager du code entre différentes analyses;
5. Partager votre code avec d'autres personnes;
6. Modifier les fonctionnalités par défaut de R.

Mais qu'est ce qu'une fonction?



Syntaxe d'une fonction

```
function_name ← function(argument1, argument2, ... ) {  
  expression...  # Ce que la fonction fait  
  return(value)  # Optionnel, pour sortir le résultat de la fonction  
}
```

Arguments d'une fonction

```
function_name ← function(argument1, argument2, ... ) {  
  expression ...  
  return(value)  
}
```

Les arguments sont les données fournies en entrée à votre fonction et contiennent l'information nécessaire pour que la fonction opère correctement.

Une fonction peut avoir entre 0 et une infinité d'arguments. Par exemple:

```
operations ← function(number1, number2, number3) {  
  result ← (number1 + number2) * number3  
  print(result)  
}
```

```
operations(1, 2, 3)  
# [1] 9
```

Exercice 5



Using what you learned previously on flow control, create a function `print_animal` that takes an `animal` as argument and gives the following results:

À l'aide de vos nouvelles connaissances sur les structures de contrôle, créez une fonction appelée `print_animal` qui prend un `animal` en argument et sort les résultats suivants:

```
Scruffy ← "dog"
Paws ← "cat"

print_animal(Scruffy)
# [1] "woof"

print_animal(Paws)
# [1] "meow"
```

Challenge 5: Solution



```
print_animal ← function(animal) {  
  if (animal == "dog") {  
    print("woof")  
  } else if (animal == "cat") {  
    print("meow")  
  }  
}
```

Valeurs par défaut dans une fonction

Les arguments peuvent aussi être optionnels, auquel cas on peut leur donner une valeur par défaut.

Ceci peut s'avérer utile si on utilise souvent une fonction avec les mêmes paramètres, mais qu'on veut tout de même garder la possibilité de changer leur valeur si nécessaire.

```
operations ← function(number1, number2, number3 = 3) {  
  result ← (number1 + number2) * number3  
  print(result)  
}
```

```
operations(1, 2, 3) # est équivalent à
```

```
# [1] 9
```



```
operations(1, 2)
```

```
# [1] 9
```

```
operations(1, 2, 2) # on peut toujours changer la valeur de number3
```

```
# [1] 6
```

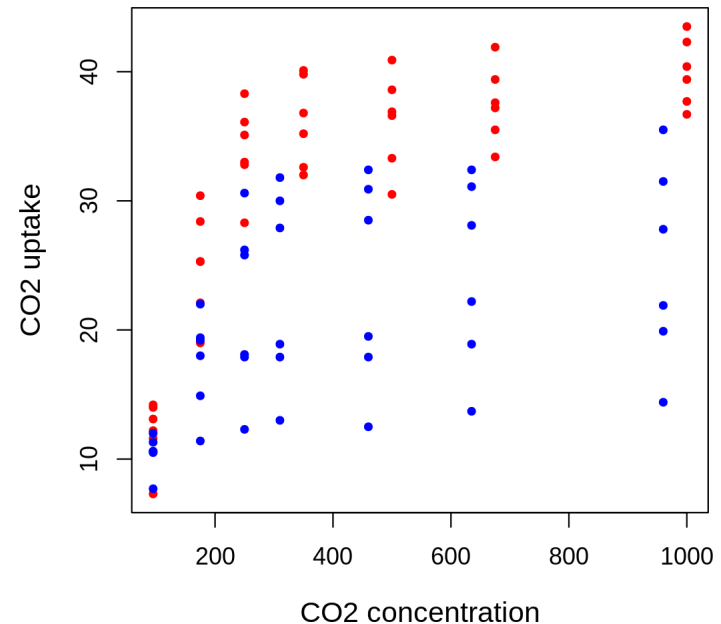
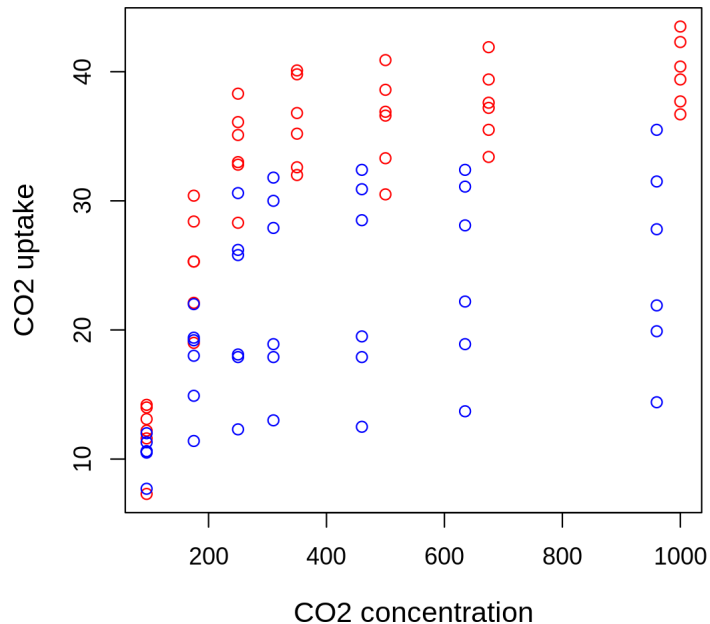

Argument

L'argument spécial  vous permet de passer des arguments à une autre fonction utilisée à l'intérieur de votre fonction. Ici, on utilise  pour passer des arguments à `plot()` et à `points()`.

```
plot.CO2 <- function(CO2, ... ) {  
  plot(x=CO2$conc, y=CO2$uptake, type="n", ...) # On utilise ... pour pas  
  for (i in 1:length(CO2[,1])){  
    if (CO2$Type[i] == "Quebec") {  
      points(CO2$conc[i], CO2$uptake[i], col = "red", type = "p", ... )  
    } else if (CO2$Type[i] == "Mississippi") {  
      points(CO2$conc[i], CO2$uptake[i], col = "blue", type = "p", ... )  
    }  
  }  
}  
  
plot.CO2(CO2, cex.lab=1.2, xlab="Concentration CO2", ylab="CO2 uptake")  
plot.CO2(CO2, cex.lab=1.2, xlab="Concentration CO2", ylab="CO2 uptake", p
```

Argument ...

L'argument spécial `...` vous permet de passer des arguments à une autre fonction utilisée à l'intérieur de votre fonction. Ici, on utilise `...` pour passer des arguments à `plot()` et à `points()`.



Argument

L'argument spécial  permet d'entrer un nombre indéfini d'arguments.

```
sum2 ← function( ... ){  
  args ← list( ... )  
  result ← 0  
  for (i in args) {  
    result ← result + i  
  }  
  return (result)  
}
```

```
sum2(2, 3)
```

```
# [1] 5
```

```
sum2(2, 4, 5, 7688, 1)
```

```
# [1] 7700
```

Valeurs de retour

La dernière expression évaluée dans une `fonction` devient la valeur de sortie.

```
myfun ← function(x) {  
  if (x < 10) {  
    0  
  } else {  
    10  
  }  
}
```

```
myfun(5)  
# [1] 0  
myfun(15)  
# [1] 10
```

`function()` sort la dernière valeur évaluée, même sans inclure la fonction `return()`.

Valeurs de retour

Utiliser `return()` de façon explicite peut être utile si la boucle doit terminer tôt, sortir de la fonction, et sortir une valeur.

```
simplefun1 ← function(x) {  
  if (x<0)  
    return(x)  
}
```

Un seul objet (ou texte) de retour peut être renvoyé par une fonction. Par contre, ceci n'est pas une limite: on peut renvoyer une `liste` contenant plusieurs objets.

```
simplefun2 ← function(x, y) {  
  z ← x + y  
  return(list("result" = z,  
             "x" = x,  
             "y" = y))  
}
```

```
simplefun2(1,
```

```
# $result  
# [1] 3  
#  
# $x  
# [1] 1  
#  
# $y  
# [1] 2
```

Exercice 6



En utilisant vos nouvelles connaissances de fonctions et de structures de contrôle, créez une fonction `bigsum` qui prend 2 arguments `a` et `b`, et:

1. Sort 0 si la somme de `a` et `b` est strictement inférieure à 50;
2. Sinon, sort la somme de `a` et `b`.

Exercise 6: Solution



Solution 1

```
bigsum ← function(a, b) {  
  result ← a + b  
  if (result < 50) {  
    return(0)  
  } else {  
    return (result)  
  }  
}
```

Solution 2

```
bigsum ← function(a, b) {  
  result ← a + b  
  if (result < 50) {  
    0  
  } else {  
    result  
  }  
}
```

Accessibilité des variables

Il est essentiel de pouvoir situer nos variables, et de savoir si elles sont définies et accessibles.

➔ Les variables définies **à l'intérieur** d'une fonction ne sont pas accessibles à l'extérieur de la fonction!

➔ Les variables définies **à l'extérieur** d'une fonction sont accessibles à l'intérieur, mais ce n'est jamais une bonne idée! Votre fonction ne fonctionnera plus si la variable extérieure est effacée!

Accessibilité des variables

```
var1 ← 3      # 'var1' est définie à l'extérieur de la fonction
vartest ← function() {
  a ← 4      # 'a' est définie à l'intérieur
  print(a)   # affiche 'a'
  print(var1) # affiche 'var1'
}
```

```
a      # on ne peut pas afficher 'a', car 'a' n'existe qu'à l'intérieur
# Error in eval(expr, envir, enclos): object 'a' not found
```

```
vartest()    # cvartest() affiche 'a' et 'var1'
# [1] 4
# [1] 3
```

```
rm(var1)     # supprime 'var1'
vartest()    # la fonction ne fonctionne plus, car 'var1' n'existe plus!
# [1] 4
# Error in print(var1): object 'var1' not found
```

Accessibilité des variables

Truc obligatoire. Utilisez donc des arguments!

De plus, à l'intérieur d'une fonction, les noms d'arguments remplaceront les noms des autres variables.

```
var1 ← 3      # var1 est définie à l'extérieur de la fonction
vartest ← function(var1) {
  print(var1) # affiche var1
}
```

```
vartest(8)    # Dans notre fonction, var1 est maintenant notre argument e
# [1] 8
```

```
var1          # var1 a toujours la même valeur à l'extérieur de la foncti
# [1] 3
```

Accessibilité des variables

Truc. Faites très attention lorsque vous créez des variables à l'intérieur d'une condition, car la variable pourrait ne jamais être créée et causer des erreurs (parfois imperceptibles).

Truc. Une bonne pratique serait de définir les variables à l'extérieur de la condition, puis ensuite de modifier leurs valeurs pour éviter ces problèmes.

```
a ← 3
if (a > 5) {
  b ← 2
}

a + b
```

```
# Error: object 'b' not found
```

Si `b` avait déjà une valeur différente assignée dans l'environnement, on aurait **gros** problème!

R ne trouverait pas d'erreur, et la valeur de `a + b` serait entièrement différente!

Bonnes pratiques de programmation

Pourquoi devrais-je me soucier de bonnes pratiques de programmation?

- Pour vous faciliter la vie;
- Pour améliorer la lisibilité et faciliter le partage et la réutilisation de votre code;
- Pour réduire le temps que vous passeriez à essayer de comprendre votre code.

Faites attention aux conseils suivants!

Gardez un code beau et propre

Les indentations et les espaces sont une première étape vers un code lisible:

- Utilisez des **espaces** avant et après vos opérateurs;
- Utilisez toujours le même opérateur d'assignation. `<=>` est préférable. `=` fonctionne, mais ne changez pas entre les deux;
- Utilisez des crochets pour encadrer vos structures de contrôle de flux:
 - À l'intérieur des crochets, faites une indentation *d'au moins 2* espaces;
 - Les crochets de fermeture occupent généralement leur propre ligne, sauf s'ils précèdent une condition `else`.
- Définissez chaque variable sur sa propre ligne.

Gardez un code beau et propre

À gauche, le code n'est pas espacé. Tous les crochets sont sur une ligne, et le code paraît malpropre.

```
a←4;b=3
if(a<b){
if(a=0)print("a zero")}else{
if(b=0){print("b zero")}else pri
```

Gardez un code beau et propre

À gauche, le code n'est pas espacé. Tous les crochets sont sur une ligne, et le code paraît malpropre. Le code à droite paraît mieux organisé, non?

```
a ← 4; b = 3
if(a < b){
if(a = 0)print("a zero")} else{
if(b = 0){print("b zero")} else pri
```

```
a ← 4
b ← 3
if(a < b){
    if(a = 0) {
        print("a zero")
    }
} else {
    if(b = 0){
        print("b zero")
    } else {
        print(b)
    }
}
```


Utilisez des fonctions pour simplifier le code

Écrivez une fonction:

1. Quand une portion du code est répété à plus de 2 reprises dans ton script;
2. Quand seulement une partie du code change et inclut des options pour différents arguments.

Ceci vous aidera à réduire le nombre d'erreurs de copier/coller, et réduira le temps passé à les corriger.

Utilisez des fonctions pour simplifier le code

Modifions l'exemple de l'**Exercice #3** et supposons que toutes les concentrations de CO_2 du Mississippi étaient surestimées de 20 et que celles du Québec étaient sous-estimées de 50.

On pourrait écrire ceci:

```
for (i in 1:length(CO2[,1])) {  
  if(CO2$Type[i] == "Mississippi")  
    CO2$conc[i] ← CO2$conc[i] -  
  }  
}  
for (i in 1:length(CO2[,1])) {  
  if(CO2$Type[i] == "Quebec") {  
    CO2$conc[i] ← CO2$conc[i] +  
  }  
}
```

Ou ceci:

```
recalibrate ← function(CO2, type)  
  for (i in 1:nrow(CO2)) {  
    if(CO2$Type[i] == type) {  
      CO2$conc[i] ← CO2$conc[i]  
    }  
  }  
  return(CO2)  
}
```

```
newCO2 ← recalibrate(CO2, "Missi  
newCO2 ← recalibrate(newCO2, "Qu
```

Noms de fonctions informatifs

Voici notre fonction de l'exemple précédent avec un nom vague.

```
rc ← function(c, t, b) {  
  for (i in 1:nrow(c)) {  
    if(c$Type[i] == t) {  
      c$conc[i] ← c$conc[i] + b  
    }  
  }  
  return (c)  
}
```

Qu'est ce que `c` et `rc`?

Quand possible, évitez d'utiliser des noms de fonctions `R` et de variables qui existent déjà pour éviter la confusion et les conflits.

Utiliser des commentaires

Truc final. Ajoutez des commentaires pour décrire tout ce que votre code fait, que ce soit le but de la fonction, comment utiliser ses arguments, ou une description détaillée de la fonction étape par étape.

```
# Recalibre le jeu de données CO2 en modifiant la concentration de CO2
# d'une valeur fixe selon la region

# Arguments
# CO2: the CO2 dataset
# type: the type ("Mississippi" or "Quebec") that need to be recalibrated
# bias: the amount to add or remove to the concentration

recalibrate ← function(CO2, type, bias) {
  for (i in 1:nrow(CO2)) {
    if(CO2$Type[i] == type) {
      CO2$conc[i] ← CO2$conc[i] + bias
    }
  }
  return(CO2)
}
```

Merci d'avoir participé à cet atelier!

