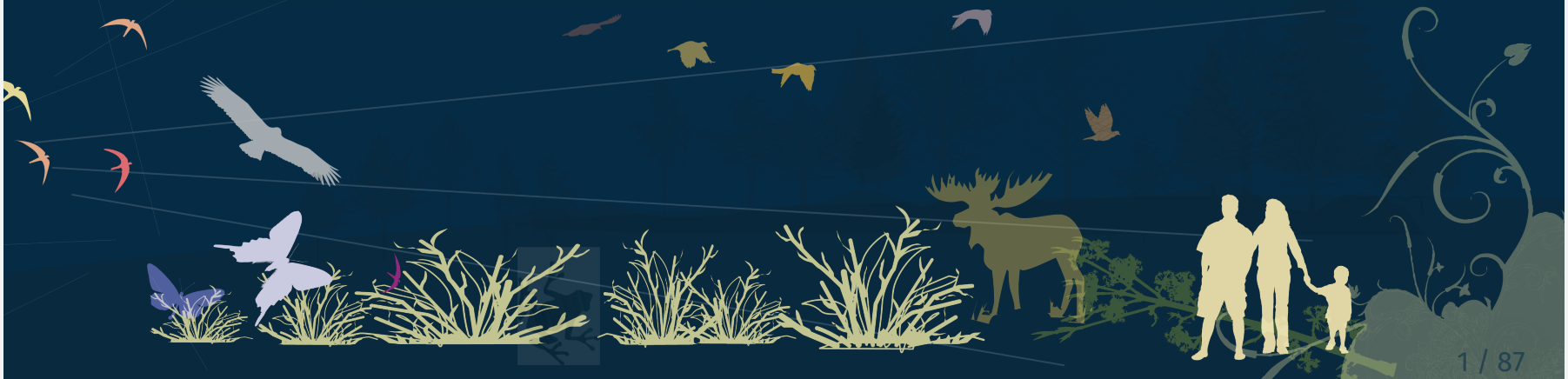
Atelier 5: Programmation en R

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

Rappel : Objets

Révision : Vecteurs

Souvenez-vous de l'atelier n°1 ?

Vecteurs numériques

```
  num.vector <- c(1, 4, 3,
                  9, 32, -4)
  num.vector
# [1]  1  4  3  9 32 -4
```

Vecteur de caractères

```
char_vector <- c("bleu",
                 "rouge",
                 "vert")
char_vector
# [1] "bleu"  "rouge" "vert"
```

Vecteur logique

```
bool_vector <- c(TRUE, TRUE, FALSE) # ou c(T, T, F)
bool_vector
# [1]  TRUE  TRUE FALSE
```

Révision : Tableaux de données

Nous pouvons commencer par créer des vecteurs multiples (petit rappel : Atelier n°1) :

```
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")
```

Nous les combinons ensuite en utilisant la fonction `data.frame()`.

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

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

Listes

Nous pouvons également créer des listes en combinant les vecteurs que nous avons créés auparavant.

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

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

Contrôle de flux

Contrôle de flux

En programmation, le **contrôle de flux** (en anglais, *control flow*) est simplement l'ordre dans lequel le programme est exécuté.

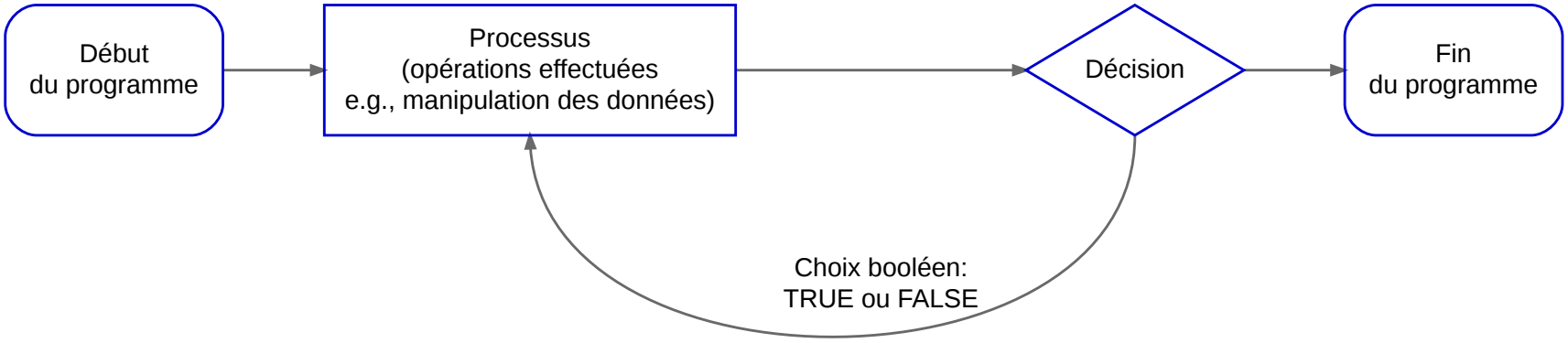
Pourquoi est-il avantageux de structurer nos programmes?

- 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 des 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 {}
```

Les clauses de sélection et d'itéraction peuvent également être contrôlées par des procedures de terminaison et de saut :

Terminaison et saut

```
break
next
```

Route pour le flux de contrôle

les clauses `if` et `if` `else`

↓

la boucle `for`

↓

les clauses `break` et `next`

↓

la boucle `repeat`

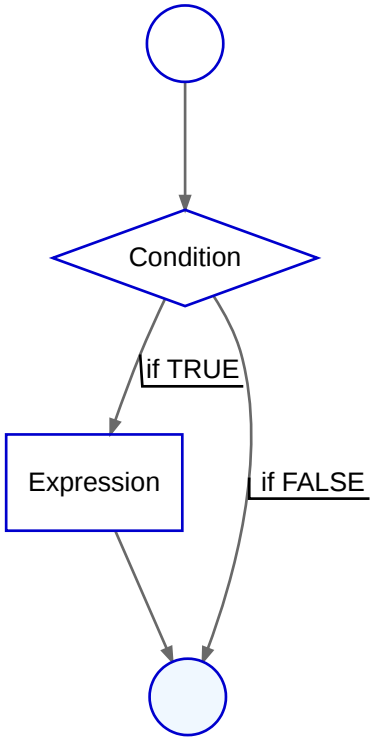
↓

la boucle `while`

Prise de décisions

Condition `if()`

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



Commande `if` `else`

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

```
Error: syntax
error in line 26
    near '1'
```

Comment peut-on tester plus qu'une condition?

- `if()` et `if()` `else` testent une seule condition;
- Mais, on peut aussi utiliser la fonction `ifelse()` pour:
  - tester un vecteur de conditions;
  - exécuter une opération selon certaines conditions.

```
a <- 1:10

ifelse(test = a > 5,
       yes = "oui",
       no = "non")
#  [1] "non" "non" "non" "non"
#  [5] "non" "oui" "oui" "oui"
#  [9] "oui" "oui"
```

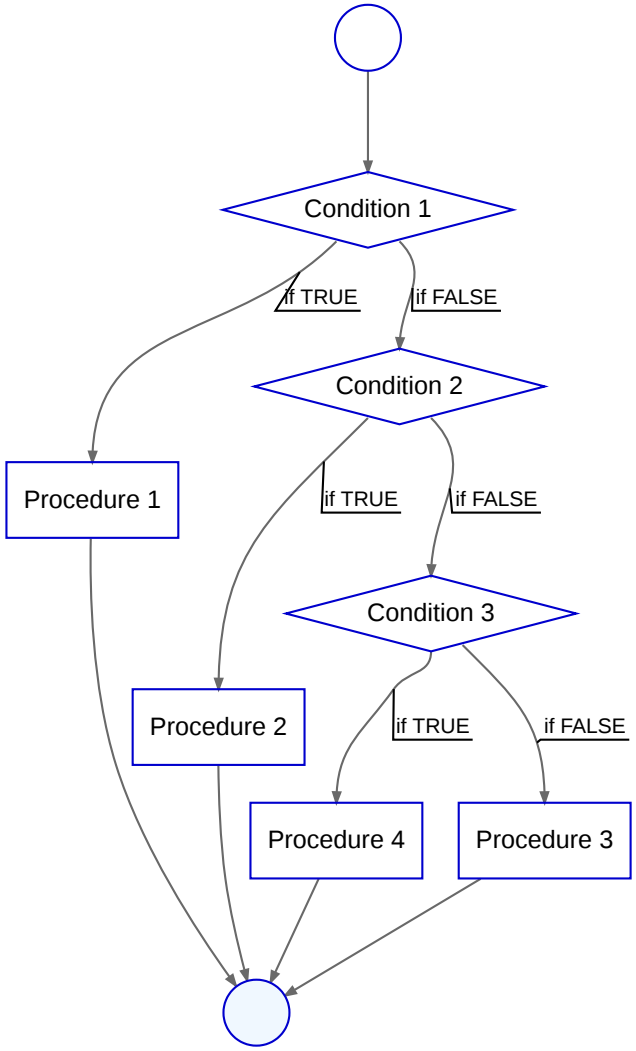
```
a <- (-4):5

sqrt(ifelse(test = a >= 0,
            yes = a,
            no = NA)
     )
#  [1]       NA       NA
#  [3]       NA       NA
#  [5] 0.000000 1.000000
#  [7] 1.414214 1.732051
#  [9] 2.000000 2.236068
```

Commandes `if()` `else` nichées

Alors que les instructions `if()` et `if()` `else` vous laissent exactement deux options, l'instruction imbriquée `if()` `else` vous permet d'envisager d'autres alternatives:

```
if (test_expression1) {
Procedure 1
} else if (test_expression2) {
Procedure 2
} else if (test_expression3) {
Procedure 3
} else {
Procedure 4
}
```



Atttention aux règles d'expression de R !

Que pensez-vous qu'il arrivera si nous essayons le code ci-dessous ?

```
if(2+2) == 4
print("Ouf, tout va bien!")
else
print("C'est la fin du monde!")
```

```
# Error: <text>:1:9: unexpected '=='
# 1: if(2+2) ==
#             ^
```

Cela ne fonctionne pas parce que `R` évalue la première ligne et ne sait pas que vous allez utiliser `else`.

Utilisez les parenthèses bouclées `{}` afin que `R` sache qu'il faut s'attendre à plus de commandes. Essayez :

```
if(2+2 == 4) {
  print("Ouf, tout va bien!")
} else {
  print("C'est la fin du monde!")
}
# [1] "Ouf, tout va bien!"
```

Exercice 1

Considérez les objets suivants :

```
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"
```

Rappel : opérateurs logiques

Opérateur	Signification
`==`	égal à
`!=`	pas égal à
`<`	plus petit que
`<=`	plus petit que ou égal à
`>`	plus grand que
`>=`	plus grand que ou égal à
`x&y`	`x` ET `y`
`x|y`	`x` OU `y`
`isTRUE(x)`	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.

Route pour le flux de contrôle

les clauses `if` et `if` `else`

↓

la boucle `for`

↓

les clauses `break` et `next`

↓

la boucle `repeat`

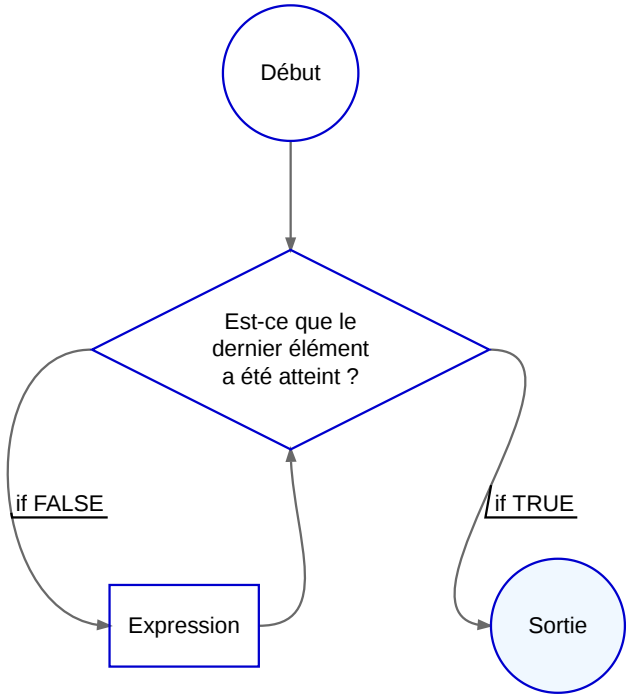
↓

la boucle `while`

Boucles `for()`

Une boucle `for()` exécute un nombre fixe d'itérations:

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



La boucle `for()`

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

```
# 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 l'exemple générique ci-dessous, `R` exécuterait l'expression 5 fois, chacune d'elles en remplaçant séquentiellement `i` par des nombres de 1 à 5 :

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

L'`expression` peut être n'importe quoi :

```
print(i + 1)
```

```
vector.a[i] <- 1 + i
```

```
matrix.b[i, 1] <- matrix.a[i, 1] * 2
```

Dans cet exemple, chaque instance de `m` est remplacé par chaque chiffre entre `1` et `7`, jusqu'au dernier élément de la séquence:

```
for(m in 1:7) {
  print(m*2)
}
```

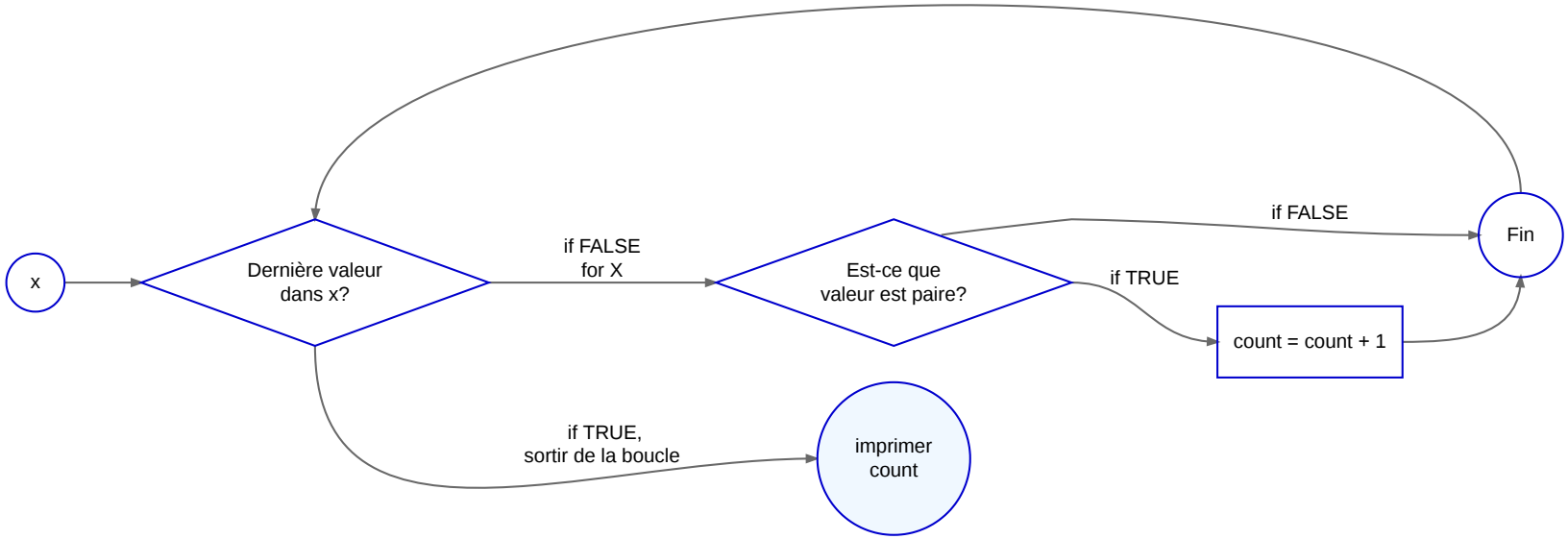
```
# [1] 2
# [1] 4
# [1] 6
# [1] 8
# [1] 10
# [1] 12
# [1] 14
```

Boucle `for()`

Nous allons effectuer des opérations pour les éléments pairs à l'intérieur de `x` en utilisant l'opérateur modulo (`%%`) :

```
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 `CO2`, qui est intégré dans R.

```
data(CO2) # This loads the built in dataset

for(i in 1:length(CO2[,1])) { # for each row in the CO2 dataset
  print(CO2$conc[i]) # print the CO2 concentration
}
```

Les 40 premiers résultats :

```
# [1] 95
# [1] 175
# [1] 250
# [1] 350
# [1] 500
# [1] 675
# [1] 1000
# [1] 95
# [1] 175
# [1] 250
```

```
# [1] 350
# [1] 500
# [1] 675
# [1] 1000
# [1] 95
# [1] 175
# [1] 250
# [1] 350
# [1] 500
# [1] 675
```

```
# [1] 1000
# [1] 95
# [1] 175
# [1] 250
# [1] 350
# [1] 500
# [1] 675
# [1] 1000
# [1] 95
# [1] 175
```

```
# [1] 250
# [1] 350
# [1] 500
# [1] 675
# [1] 1000
# [1] 95
# [1] 175
# [1] 250
# [1] 350
# [1] 500
```

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 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
# élément de la liste
lapply(SimulatedData, mean)
```

```
# $SimpleSequence
# [1] 2.5
#
# $Norm10
# [1] -0.0252946
#
# $Norm20
# [1] 0.9595792
#
# $Norm100
# [1] 5.122061
```

`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
#      2.5000000      0.5840140
#         Norm20        Norm100
#      0.8012317      4.9222527
```

`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 ou facteurs), et lorsqu'on veut appliquer une fonction sur chaque groupe.

```
mtcars[1:10, c("hp", "cyl")]
#                   hp cyl
# Mazda RX4         110   6
# Mazda RX4 Wag     110   6
# Datsun 710         93   4
# Hornet 4 Drive    110   6
# Hornet Sportabout 175   8
# Valiant           105   6
# Duster 360        245   8
# Merc 240D          62   4
# Merc 230           95   4
# Merc 280          123   6
```

```
# Moyenne de hp par cylindres
tapply(mtcars$hp,
       mtcars$cyl,
       FUN = mean)
#        4        6        8
#  82.63636 122.28571 209.21429
```

Exercice 2

En revenant du terrain, vous vous êtes rendu compte que votre outil de mesure de l'absorption de $\mathbf{CO_2}$ n'était pas correctement calibré sur les sites du Québec et que toutes les mesures sont de 2 unités supérieures à ce qu'elles devraient être.

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 $\mathbf{CO_2}$ dans les deux groupes de sites.

> Pour cela, vous devez charger l'ensemble de données $\mathbf{CO_2}$ en utilisant `data(CO2)`, et ensuite utiliser l'objet `CO2`.

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`.

Route pour le flux de contrôle

les clauses `if` et `if` `else`

la boucle `for`

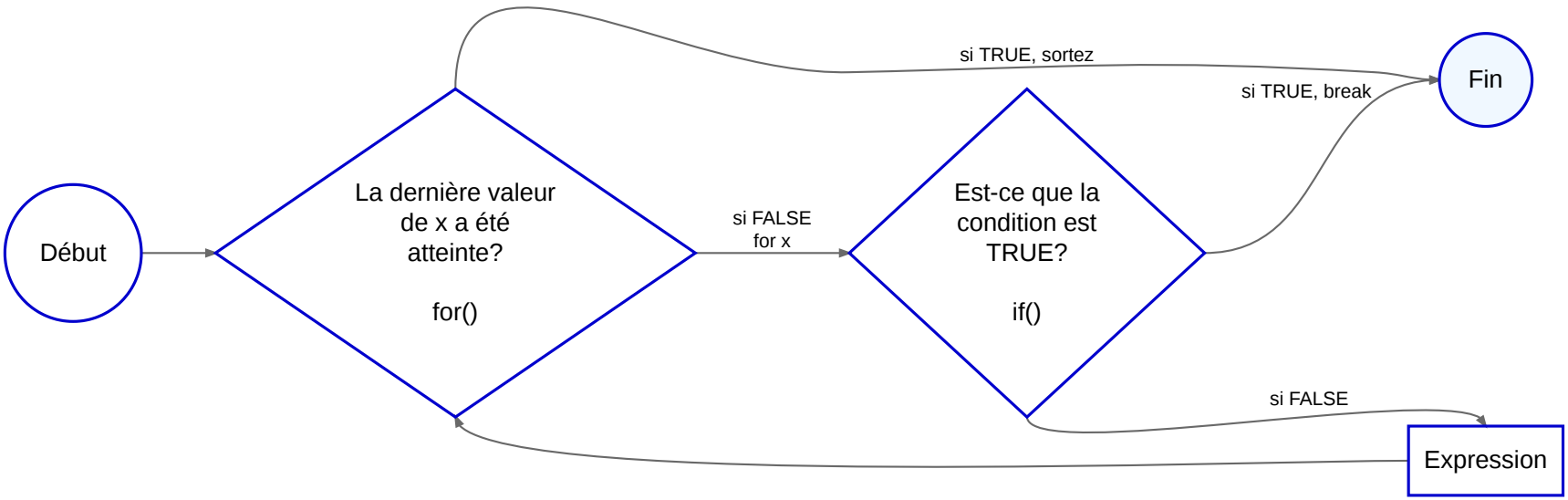
les clauses `break` et `next`

la boucle `repeat`

la boucle `while`

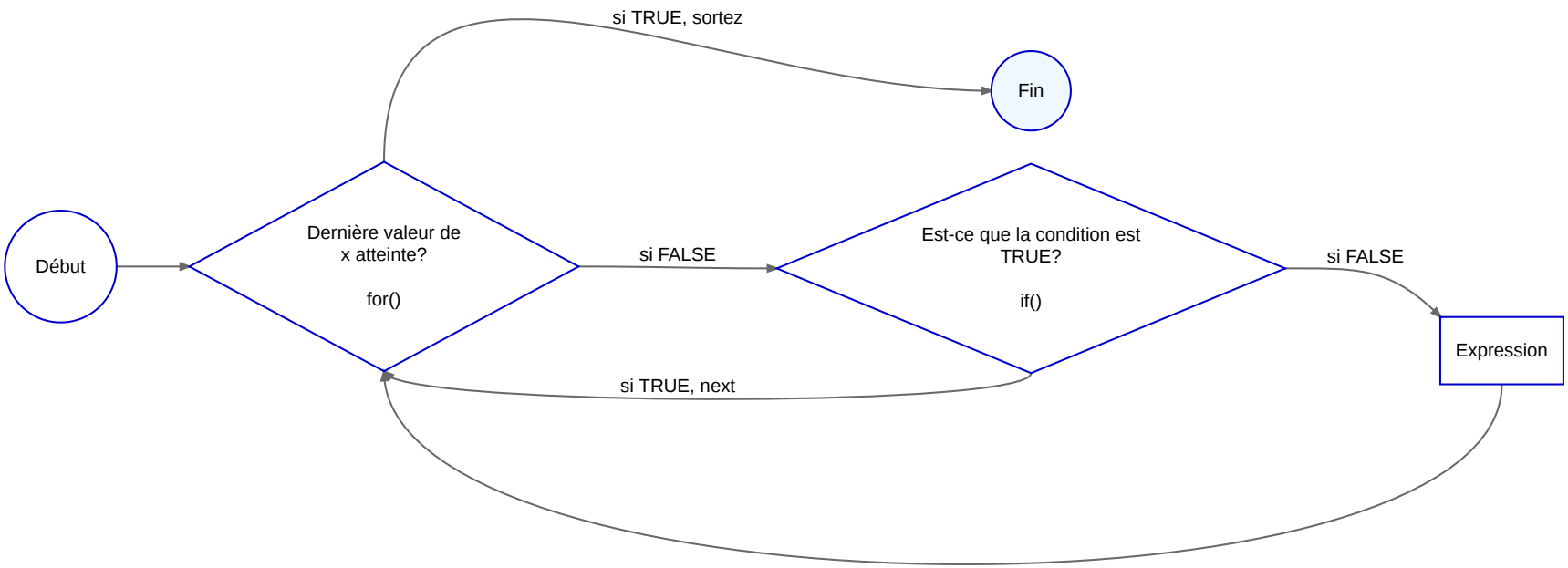
Modifier l'itération: `break`

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



Modifier l'itération: `next`

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



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
```

Route pour le flux de contrôle

les clauses `if` et `if` `else`

la boucle `for`

les clauses `break` et `next`

la boucle `repeat`

la boucle `while`

Modifier l'itération: `repeat`

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)
```

Route pour le flux de contrôle

les clauses `if` et `if` `else`

la boucle `for`

les clauses `break` et `next`

la boucle `repeat`

la boucle `while`

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

Vous vous êtes rendu compte qu'un autre de vos outils ne fonctionnait pas correctement !

Aux sites situés au Mississippi, les concentrations de moins de 300 sont bien mesurés, 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)
```

Exercice 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()`

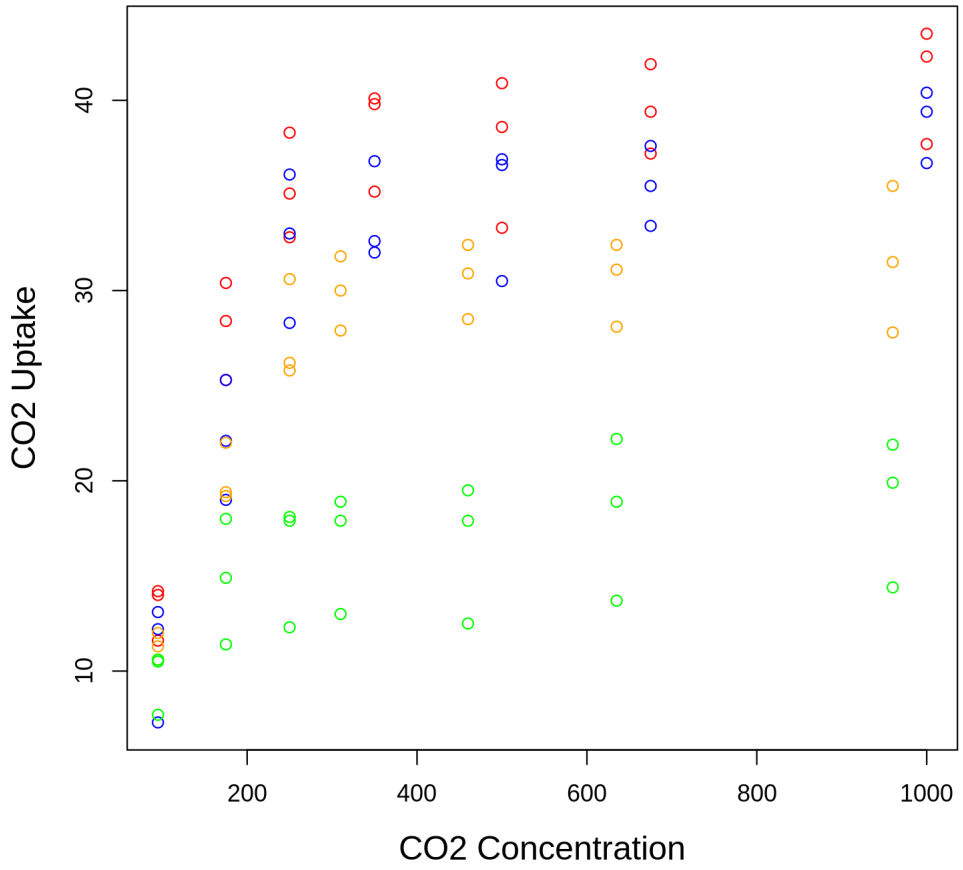
Créons un graphique **uptake** *versus* **concentration** avec des points de couleurs différentes, ouù chaque couleur est associé à un type (*Quebec* ou *Mississippi*) et à un **treatment** (*chilled* ou *nonchilled*):

```
plot(x = CO2$conc, y = CO2$uptake, type = "n",
     cex.lab = 1.4, cex.axis = 1.5,
     cex.main = 1.5, cex.sub = 1.5,
     xlab = "CO2 concentration", ylab = "CO2 uptake")

# type = "n" dit à R de ne pas afficher les points

for (i in 1:length(CO2[,1])) {
  if (CO2$Type[i] == "Quebec" & CO2$Treatment[i] == "nonchilled") {
    points(CO2$conc[i], CO2$uptake[i], col = "red")
  }
  if (CO2$Type[i] == "Quebec" & CO2$Treatment[i] == "chilled") {
    points(CO2$conc[i], CO2$uptake[i], col = "blue")
  }
  if (CO2$Type[i] == "Mississippi" & CO2$Treatment[i] == "nonchilled") {
    points(CO2$conc[i], CO2$uptake[i], col = "orange")
  }
  if (CO2$Type[i] == "Mississippi" & CO2$Treatment[i] == "chilled") {
    points(CO2$conc[i], CO2$uptake[i], col = "green")
  }
```

Visualization de données avec `for()` et `if()`



Exercice 4

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

Points bonus si vous utilisez des boucles nichées!

Étapes :

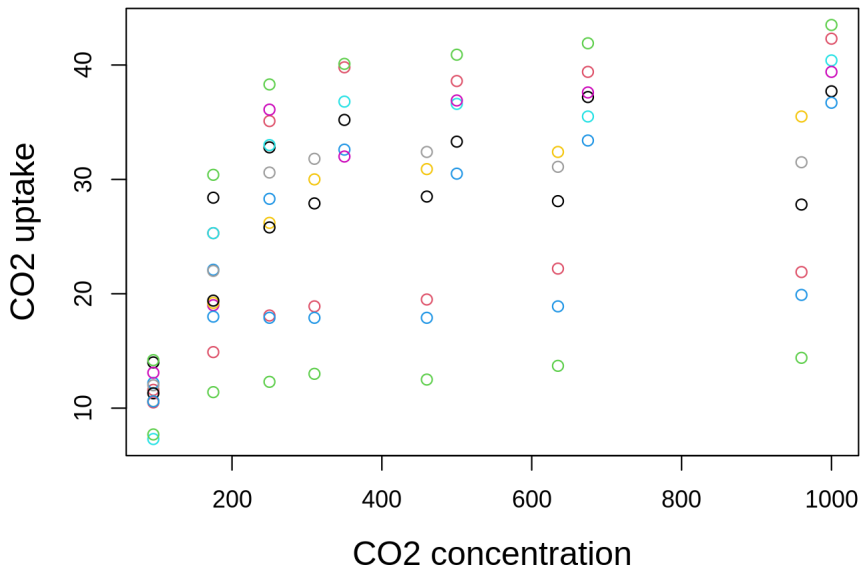
1. *Créez un plot vide ;*
2. *Créez une liste de plantes (indice :* `?unique`*) ;*
3. *Remplissez la parcelle en utilisant les instructions* `for()` *et* `if()`*.*

Exercice 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(valeur)  # Optionnel, pour sortir le résultat de la fonction
}
```

Arguments d'une fonction

```
function_name <- function(argument1, argument2, ...) {
  expression...
  return(valeur)
}
```

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(numero_1, numero_2, numero_3) {
  resultat <- (numero_1 + numero_2) * numero_3
  print(resultat)
}
```

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

Exercice 5

En utilisant ce que vous avez appris précédemment sur le contrôle de flux, créez une fonction `print_animal()` qui prend un `animal` comme argument et donne les résultats suivants :

```
Pitou <- "chien"

Minou <- "chat"

print_animal(Pitou)
# [1] "woof"

print_animal(Minou)
# [1] "miaou"
```

Challenge 5: Solution

```
print_animal <- function(animal) {
  if (animal == "chien") {
    print("woof")
  } else if (animal == "chat") {
    print("miaou")
  }
}
```

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(numero_1, numero_2, numero_3 = 3) {
  resultat <- (numero_1 + numero_2) * numero_3
  print(resultat)
}

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 numero_3
# [1] 6
```

Argument `...`

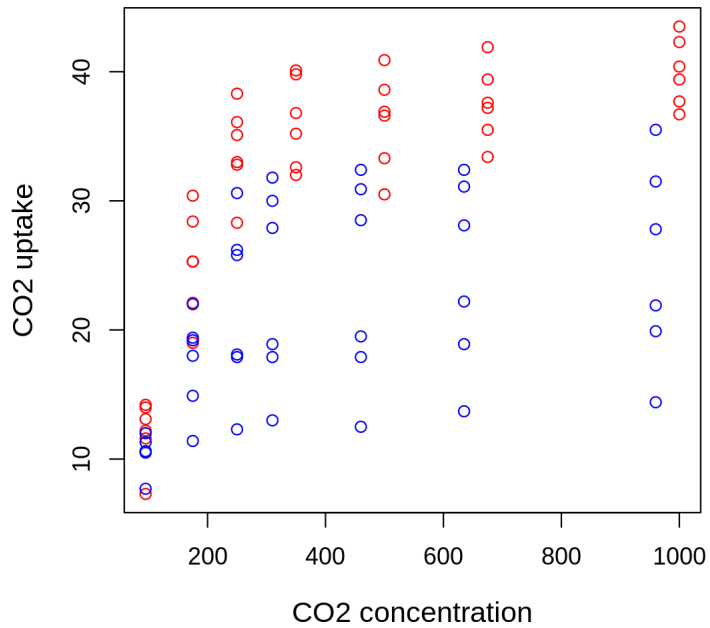
L'argument spécial `...` vous permet de passer des arguments à une autre fonction utilisée à l'intérieur de votre fonction. Ici, on utilise `...` pour passer des arguments à `plot()` et à `points()`.

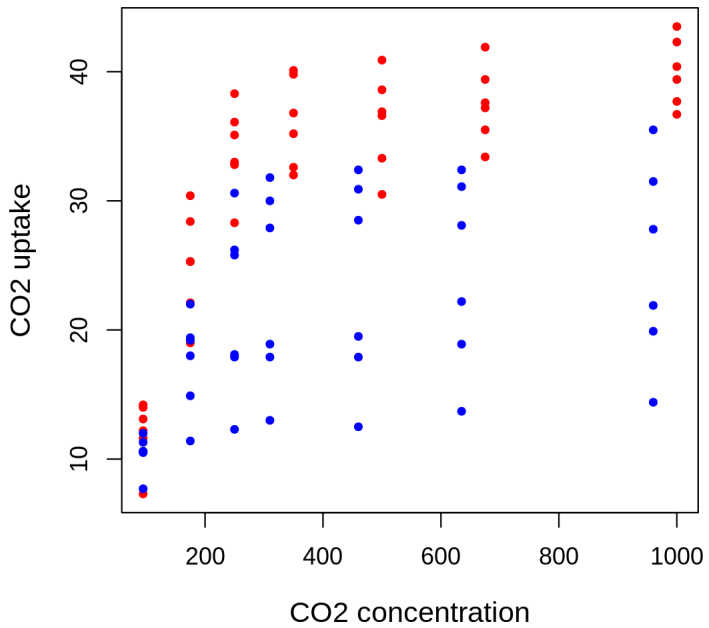
```
plot.CO2 <- function(CO2, ...) {
  plot(x=CO2$conc, y=CO2$uptake, type="n", ...) # On utilise ... pour passer les arg
  for (i in 1:length(CO2[,1])){
     if (CO2$Type[i] == "Quebec") {
       points(CO2$conc[i], CO2$uptake[i], col = "red", type = "p", ...)
     } else if (CO2$Type[i] == "Mississippi") {
       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", pch=20)
```

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 renovoyer une `liste` contenant plusieurs objets.

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

```
simplefun2(1, 2)
```

```
# $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`.

Exercice 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 a 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 de la
#  [1] -4 -3 -2 -1  0  1  2  3
#  [9]  4  5

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 et prend sa
# [1] 8

var1          # var1 a toujours la même valeur à l'extérieur de la fonction
# [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).

```
a <- 3
if (a > 5) {
  b <- 2
}

a + b
```

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

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.

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 sur les 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. Vous pouvez utiliser `=` (parfois), 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;
- Utilisez `Cmd + I` ou `Ctrl + I` dans `RStudio` pour indenter automatiquement le code mis en évidence.

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 print(b)}
```

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 print(b)}
```

```
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 Mississipi 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$conc[i] - 20
  }
}
for (i in 1:length(CO2[,1])) {
  if(CO2$Type[i] == "Quebec") {
    CO2$conc[i] <- CO2$conc[i] + 50
  }
}
```

Ou ceci:

```
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)
}
```

```
newCO2 <- recalibrate(CO2 = CO2,
                      type = "Mississi
                      bias = -20)
newCO2 <- recalibrate(newCO2, "Quebec",
```

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.

Utilisez 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)
}
```

Exercice de groupe

En utilisant ce que vous avez appris, écrivez une déclaration `if()` qui vérifie si une variable numérique `x` est égale à zéro. Si ce n'est pas le cas, elle attribue $cos(x)/x$ à `z`, sinon elle attribue 1 à `z`.

Créez une fonction appelée `ma_fonction()` qui prend la variable `x` en argument et retourne `z`.

Si nous attribuons respectivement 45, 20 et 0 à `x`, laquelle des options suivantes représenterait les résultats ?

1. 0.054, 0.012, et 0;

2. 0.020, 0.054, et 1;

3. 0.012, 0.020, et 1.

Exercice de groupe : Solution

La bonne réponse est l'option **3** (0.12, 0.20, et 1).

```
ma_fonction <- function(x) {
  if(x != 0) {
  z <- cos(x)/x
  } else { z <- 1 }
  return(z)
}
```

```
ma_fonction(45)
# [1] 0.01167382

ma_fonction(20)
# [1] 0.0204041

ma_fonction(0)
# [1] 1
```

Merci d'avoir participé à cet atelier!