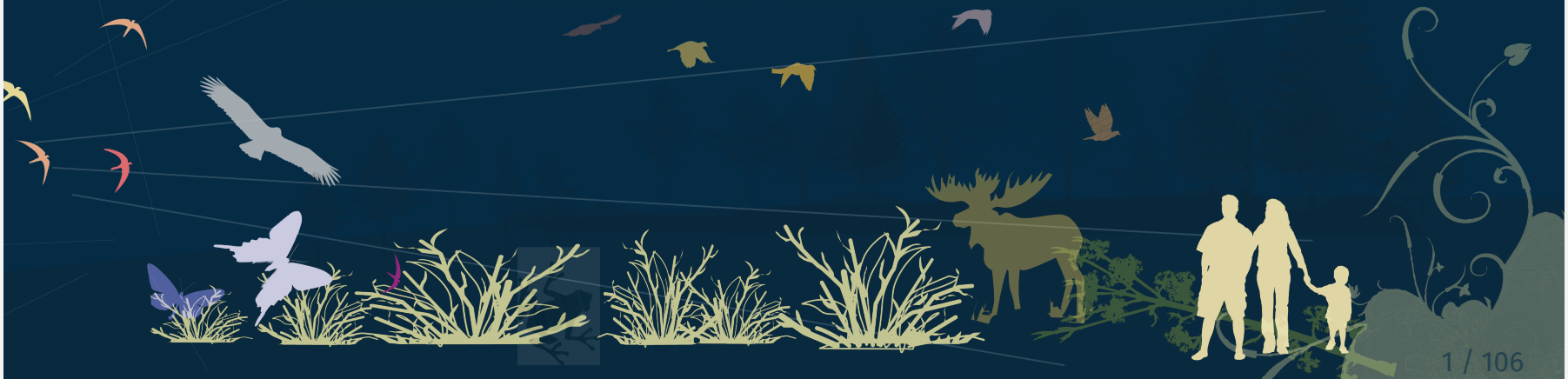
Workshop 5: Programming in R

QCBS R Workshop Series

Québec Centre for Biodiversity Science



About this workshop

Learning Objectives

1. Recognizing **control flow**;
2. Getting comfortable with testing conditions and performing iterations;
3. Developing your first functions in R;
4. Discovering how to accelerate your code;
5. Demonstrating useful R packages for biologists.

Review

Objects

Review: Vectors

Recall Workshop #1?

Numeric vectors

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

Character vector

```
char_vector <- c("blue",
                 "red",
                 "green")
char_vector
# [1] "blue"  "red"   "green"
```

Logical vector

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

Review: Data frames

We can begin by creating multiple vectors (remember Workshop #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")
```

We then combine them using the function `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
```

Review: Lists

We can also create lists by combining the vectors we created before.

```
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"
```

Control flow

Control flow

Program flow control can be simply defined as the order in which a program is executed.

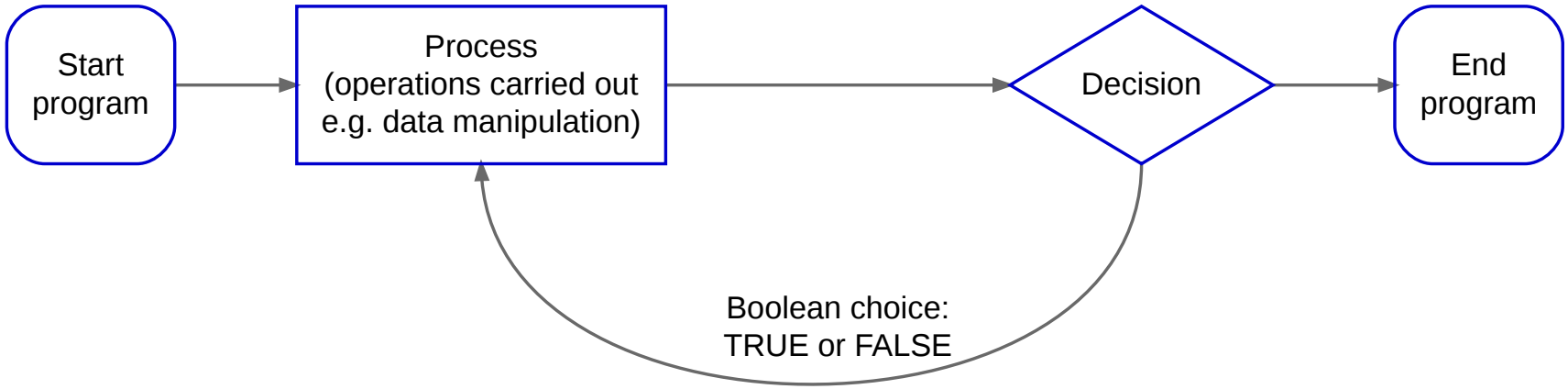
Why is it advantageous to have structured programs?

- It **decreases the complexity** and time of the task at hand;
- This logical structure also means that the code has **increased clarity**;
- It also means that **many programmers can work on one program**.

This means increased productivity.

Control flow

Program flowcharts can be used to plan programs and represent their structure.



Representing structure

The two basic building blocks of codes are the following:

Selection

Program's execution determined by statements

```
if() {}
if() {} else {}
```

Iteration

Repetition, where the statement will **loop** until a criteria is met

```
for() {}
while() {}
repeat {}
```

Selection and iterative statements can also be controlled by termination and jump statements:

Termination and Jump

```
break
next
```

Control flow roadmap

`if` and `if` `else` statements

↓

`for` loop

↓

`break` and `next` statements

↓

`repeat` loop

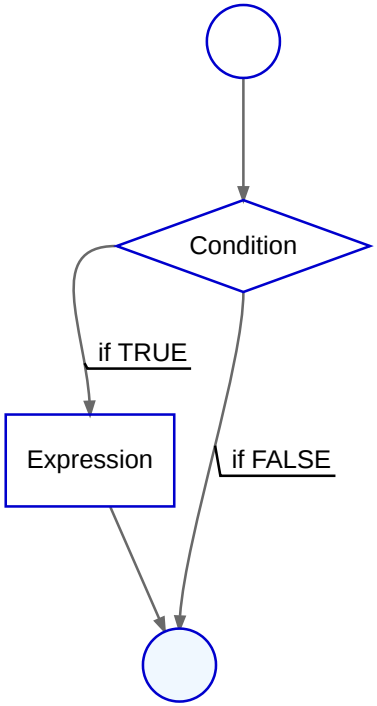
↓

`while` loop

Decision making

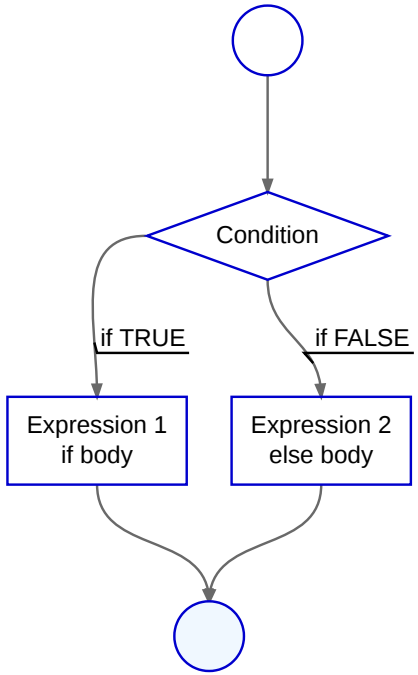
`if()` statement

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



`if()` `else` statement

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



What if you want to test more than one condition?

- `if()` and `if()` `else` test a single condition.
- You can also use the `ifelse()` function to:
  - test a vector of conditions;
  - apply a function only under certain conditions.

```
a <- 1:10

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

```
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
```

Nested `if()` `else` statement

While the `if()` and `if()` `else` statements leave you with exactly two options, nested `if()` `else` statement allows you consider more alternatives.

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

Beware of R's expression parsing!

What do you think will happen if we try the code below?

```
if(2+2) == 4
print("Arithmetic works.")
else
print("Houston, we have a problem.")
```

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

This does not work because R evaluates the first line and does not know that you are going to use an `else` statement

Use curly brackets `{ }` so that R knows to expect more input. Try:

```
if(2+2 == 4) {
  print("Arithmetic works.")
} else {
  print("Houston, we have a problem.")
}
# [1] "Arithmetic works."
```

Challenge 1

Consider the following objects:

```
Paws <- "cat"
Scruffy <- "dog"
Sassy <- "cat"
animals <- c(Paws, Scruffy, Sassy)
```

1. Use an `if()` statement to print "meow" if `Paws` is a "cat".
2. Use an `if()` `else` statement to print `"woof"` if you supply an object that is a `"dog"` and `"meow"` if it is not. Try it out with `Paws` and `Scruffy`.
3. Use the `ifelse()` function to display `"woof"` for `animals` that are dogs and `"meow"` for `animals` that are cats.

Remember the logical operators

Command	Meaning
`==`	equal to
`!=`	not equal to
`<`	less than
`<=`	less than or equal to
`>`	greater than
`>=`	greater than or equal to
`x&y`	`x` AND `y`
`x|y`	`x` OR `y`
`isTRUE(x)`	test if `x` is true

Challenge 1 - Solution

1. Use an `if()` statement to print “meow” if `Paws` is a “cat”.

```
if(Paws == 'cat') {
  print("meow")
}
# [1] "meow"
```

2. Use an `if()` `else` statement to print `“woof”` if you supply an object that is a `“dog”` and `“meow”` if it is not. Try it out with `Paws` and `Scruffy`.

```
x = Paws
# x = Scruffy
if(x == 'cat') {
  print("meow")
} else {
  print("woof")
}
# [1] "meow"
```

Challenge 1 - Solution

3. Use the `ifelse()` function to display `"woof"` for `animals` that are dogs and `"meow"` for `animals` that are cats.

```
animals <- c(Paws, Scruffy, Sassy)

ifelse(animals == 'dog', "woof", "meow")
# [1] "meow" "woof" "meow"
```

Or

```
for(val in 1:3) {
  if(animals[val] == 'cat') {
    print("meow")
  } else if(animals[val] == 'dog') {
    print("woof")
  } else print("what?")
}
# [1] "meow"
# [1] "woof"
# [1] "meow"
```

Iteration

Every time some operations have to be repeated, a loop may come in handy.

Loops are good for:

- Doing something for every element of an object;
- Doing something until the processed data runs out;
- Doing something for every file in a folder;
- Doing something that can fail, until it succeeds;
- Iterating a calculation until it converges.

Control flow roadmap

`if` and `if` `else` statements

`for` loop

`break` and `next` statements

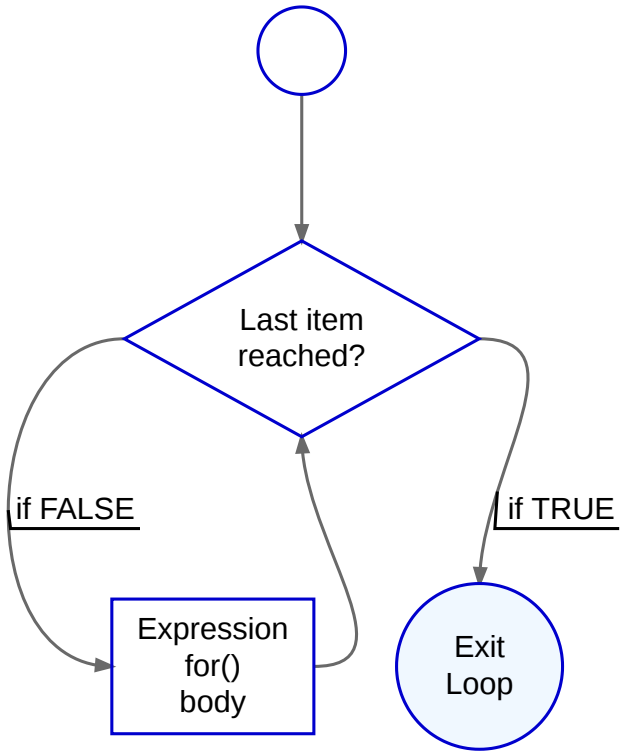
`repeat` loop

`while` loop

`for()` loop

A `for()` loop works in the following way:

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



`for()` loop

A `for()` loop works in the following way:

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

The letter `i` can be replaced with any variable name, `sequence` can be elements or the position of these elements, and `expression` can be anything:

```
for(a in c("Hello",
           "R",
           "Programmers")) {
  print(a)
}
```

```
# [1] "Hello"
# [1] "R"
# [1] "Programmers"
```

```
for(z in 1:4) {
  a <- rnorm(n = 1,
             mean = 5 * z,
             sd = 2)
  print(a)
}
```

```
# [1] 4.611795
# [1] 14.03735
# [1] 15.73001
# [1] 19.36227
```

`for()` loop

A `for()` loop works in the following way:

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

As expected, you can use `for()` loops in different object types and classes, such as a `list`:

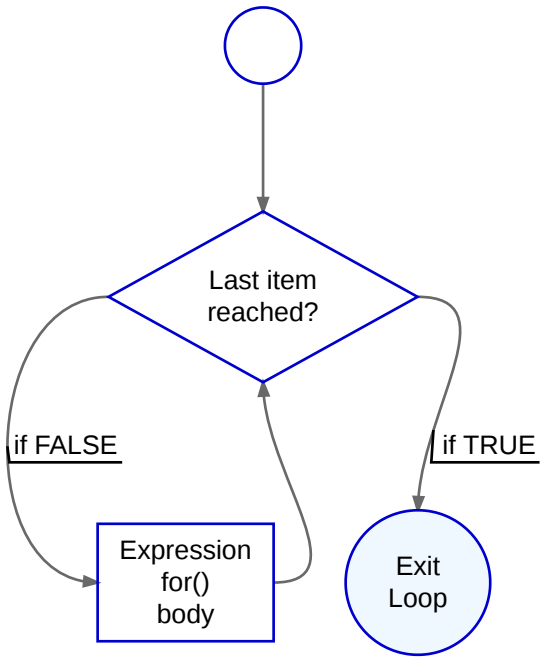
```
(elements <- list(a = 1:3,
                  b = 4:10,
                  c = 7:-1))
# $a
# [1] 1 2 3
#
# $b
# [1]  4  5  6  7  8  9 10
#
# $c
# [1]  7  6  5  4  3  2  1  0 -1
```

```
for(element in elements) {
  print(element*2)
}
# [1] 2 4 6
# [1]  8 10 12 14 16 18 20
# [1] 14 12 10  8  6  4  2  0 -2
```

`for()` loop

Remember our flowchart?

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



We will apply the following example into our flow chart!

Here, every instance of `m` is being replaced by each number between `1` and `7`, until it reaches the last element of the sequence:

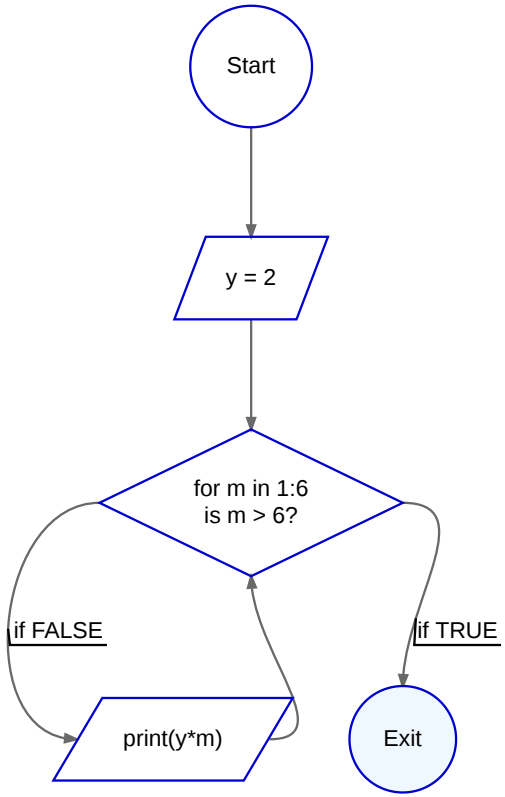
```
y <- 2
for(m in 1:6) {
  print(y*m)
}
```

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

`for()` loop

Remember our flowchart?

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



We will apply the following example into our flow chart!

Here, every instance of `m` is being replaced by each number between `1` and `7`, until it reaches the last element of the sequence:

```
y <- 2
for(m in 1:6) {
  print(y*m)
}
```

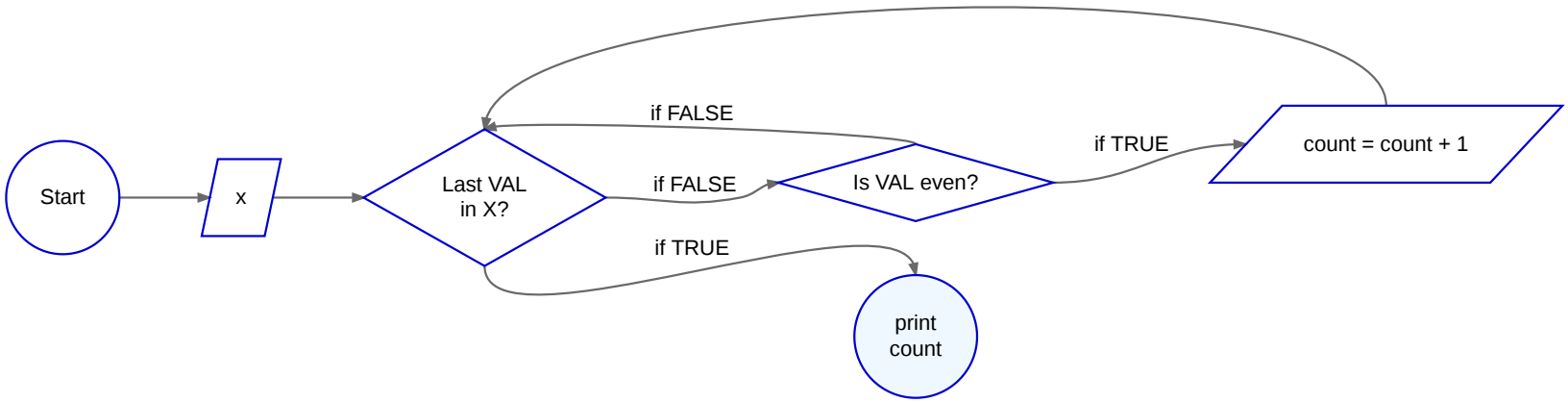
```
# [1] 2
# [1] 4
# [1] 6
# [1] 8
# [1] 10
# [1] 12
```

`for()` loop

Let us perform operations for even elements within `x` using the modulo operator (`%%`):

```
x <- c(2, 5, 3, 9, 6)
count <- 0
```

```
for(val in x) {
  if(val %% 2 == 0) {
    count <- count + 1
  }
}
print(count)
```



`for()` loop

`for()` loops are often used to loop over a dataset. We will use loops to perform functions on the `CO2` dataset which is built in `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
}
```

First 40 outputs:

```
# [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
```

`for()` loop

Another example:

```
for(i in 1:length(CO2[,1])) { # for each row in the CO2 dataset
  if(CO2$Type[i] == "Quebec") { # if the type is "Quebec"
    print(CO2$conc[i]) # print the CO2 concentration
    }
}
```

Outputs:

```
# [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
# [1] 675
# [1] 1000
```

`for()` loop

Tip 1. To loop over the number of rows of a data frame, we can use the function `nrow()`.

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

```
# [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
```

`for()` loop

Tip 2. To perform operations on the elements of one column, we can directly iterate over it.

```
for(p in CO2$conc) {
      # for each element of
      # the column "conc" of
      # the CO2 df
  print(p)
      # print the p-th element
}
```

```
# [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
```

`for()` loop

The expression within the loop can be almost anything and is usually a compound statement containing many commands.

```
for(i in 4:5) { # for i in 4 to 5
  print(colnames(CO2)[i])
  print(mean(CO2[,i])) # print the mean of that column from the CO2 dataset
}
```

Output:

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

Nested `for()` loops within `for()` loops

In some cases, you may want to use nested loops to accomplish a task. When using nested loops, it is important to use different variables as counters for each of your loops. Here we used `i` and `n`:

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

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

Getting good: Using the `apply()` family

`R` disposes of the `apply()` function family, which consists of iterative functions that aim at **minimizing your need to explicitly create loops**.

`apply()` can be used to apply functions to a matrix.

```
height <- 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 = height,
      MARGIN = 1,
      FUN = mean)
# [1] 13.5 14.5 15.5 16.5 17.5
```

```
?apply
```

`lapply()`

`lapply()` applies a function to every element of a `list`.

It may be used for other objects like **dataframes**, **lists** or **vectors**.

The output returned is a `list` (explaining the "`l`" in `lapply`) and has the same number of elements as the object passed to it.

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

# Apply mean to each element
## of the list
lapply(SimulatedData, mean)
```

```
# $SimpleSequence
# [1] 2.5
#
# $Norm10
# [1] 0.01225159
#
# $Norm20
# [1] 0.9533073
#
# $Norm100
# [1] 4.823236
```

`sapply()`

`sapply()` is a 'wrapper' function for `lapply()`, but returns a simplified output as a `vector`, instead of a `list`.

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

```
# Apply mean to each element of the list
sapply(SimulatedData, mean)
# SimpleSequence         Norm10
#      2.5000000     -0.1388967
#         Norm20        Norm100
#      1.0449090      4.9599476
```

`mapply()`

`mapply()` works as a multivariate version of `sapply()`.

It will apply a given function to the first element of each argument first, followed by the second element, and so on. For example:

```
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()` is used to apply a function over subsets of a vector.

It is primarily used when the dataset contains different groups
(*i.e.* levels or factors), and we want to apply a function to each of these groups.

```
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
```

```
# mean hp by cylinder groups
tapply(mtcars$hp,
       mtcars$cyl,
       FUN = mean)
#         4         6         8
#  82.63636 122.28571 209.21429
```

Challenge 2

After coming back from the field, you have realized that your tool for measuring CO_2 uptake was not calibrated properly at Quebec sites and all measurements are 2 units higher than they should be.

Now, you must do the following:

1. Use a loop to correct these measurements for all Quebec sites;
2. Use an `apply()` family function to calculate the average CO_2 uptake in both Québec and Mississipi sampled sites.

> For this, you must load the CO_2 dataset using `data(CO2)`, and then use the object `CO2`.

Get your hands dirty!

Challenge 2: Solution

1.Using `for()` and `if()` to correct the measurements:

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

2.Using `tapply()` to calculate the mean for each group:

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

Modifying iterations

Normally, loops iterate over and over until they finish.

Sometimes you may be interested in breaking this behaviour.

For example, you may want to tell `R` to stop executing the iteration when it reaches a given element or condition.

You may also want `R` to jump certain elements when certain conditions are met.

For this, we will introduce `break`, `next` and `while`.

Control flow roadmap

`if` and `if` `else` statements

↓

`for` loop

↓

`break` and `next` statements

↓

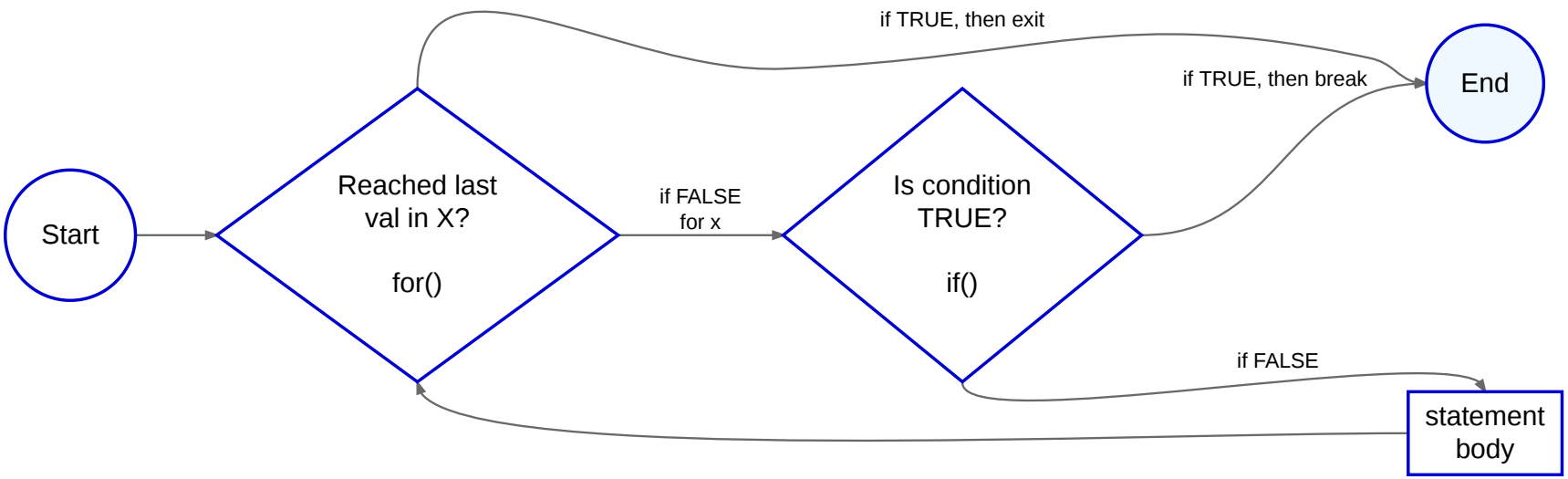
`repeat` loop

↓

`while` loop

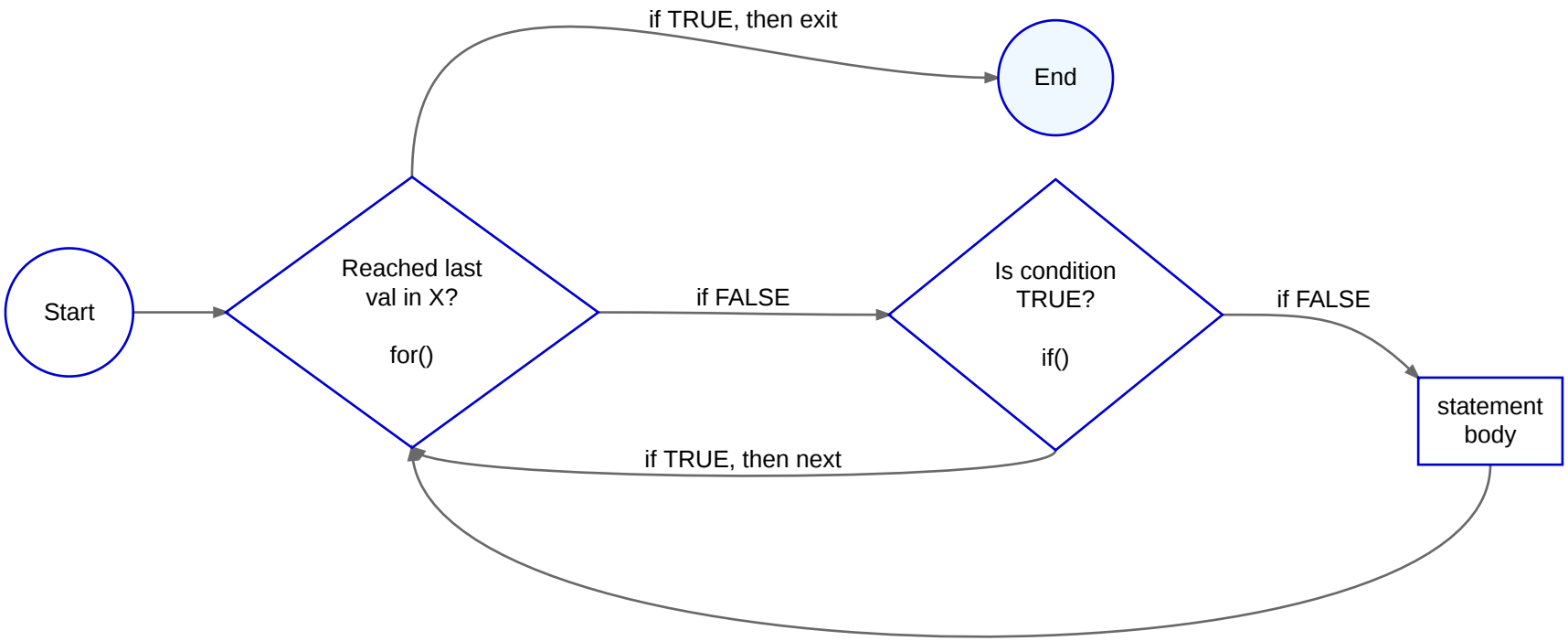
Modifying iterations: `break` statement

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



Modifying iterations: `next` statement

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



Modifying iterations: `next` statement

Print the CO_2 concentrations for *chilled* treatments and keep count of how many replications were done.

```
count <- 0

for(i in 1:nrow(CO2)) {
  if(CO2$Treatment[i] == "nonchilled") next
  # Skip to next iteration if treatment is nonchilled
  count <- count + 1
#  print(CO2$conc[i]) # You can turn this on if you want to
}
print(count) # The count and print command were performed 42 times.
```

```
# [1] 42
```

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

Control flow roadmap

`if` and `if` `else` statements

`for` loop

`break` and `next` statements

`repeat` loop

`while` loop

Modifying iterations: `repeat` loop

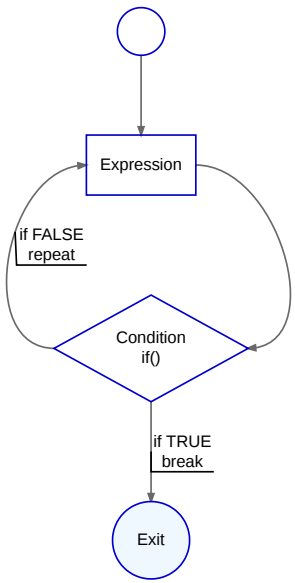
Under `repeat`, a task is performed until it is deliberately stopped:

```
repeat {
  print("Press 'Esc' to stop me!")
}
```

```
[1] "Press 'Esc' to stop me!"
[1] "Press 'Esc' to stop me!"
...
...
[1] "Press 'Esc' to stop me!"
```

You can stop this cycle using a condition and `break`:

```
repeat {
  expression
  if {
    condition
    } break
}
```



Modifying iterations: `repeat` loop

Remember our code that used the `next` statement to print the total counts of CO_2 concentrations taken for *chilled* treatments?

```
count <- 0

for(i in 1:nrow(CO2)) {
  if(CO2$Treatment[i] == "nonchilled")
  count <- count + 1
}

print(count)
# [1] 42
```

It could had been equivalently written using `repeat` and `break`:

```
count <- 0
i <- 0

repeat {
      i <- i + 1
      if(CO2$Treatment[i] == "nonchille
      count <- count + 1
      if(i == nrow(CO2)) break
}

print(count)
# [1] 42
```

Control flow roadmap

`if` and `if` `else` statements

`for` loop

`break` and `next` statements

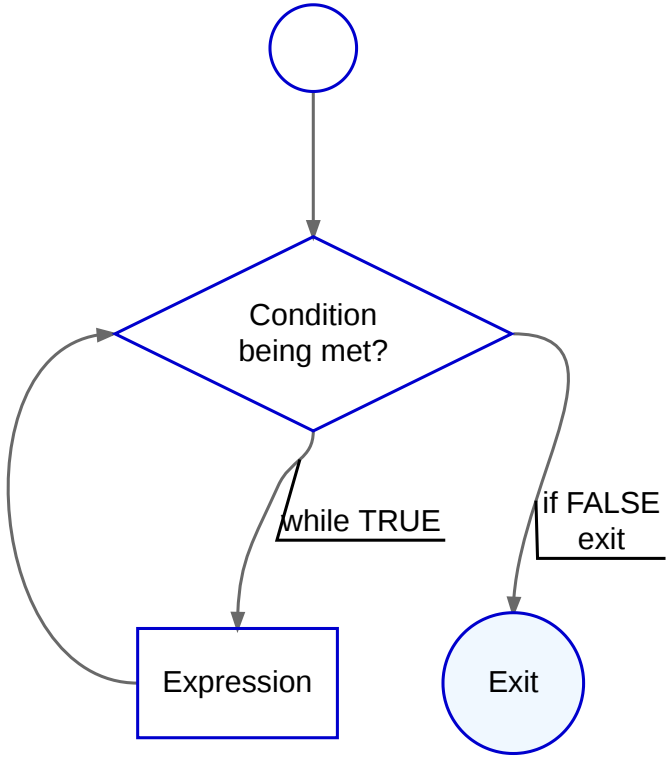
`repeat` loop

`while` loop

Modifying iterations: `while` loop

Within the `while` loop, an expression happens **while** a condition is met.

```
while(condition){
  expression
}
```



Modifying iterations: `while` loop

Again, remember our code that used the `next` statement to print the total counts of CO_2 concentrations taken for *chilled* treatments?

```
count <- 0
for(i in 1:nrow(CO2)) {
  if(CO2$Treatment[i] == "nonchilled")
  count <- count + 1
}
print(count)
```

It could had been equivalently written using `repeat` and `break`:

```
count <- 0
i <- 0
repeat {
      i <- i + 1
      if(CO2$Treatment[i] == "nonchille
      count <- count + 1
      if(i == nrow(CO2)) break
}
print(count)
```

Or, using a `while` loop:

```
i <- 0
count <- 0
while(i < nrow(CO2))
{
  i <- i + 1
  if(CO2$Treatment[i] == "nonchilled") next
  count <- count + 1
}
```

Challenge 3

You have realized that another of your tools was not working properly!

At **Mississippi** sites, **concentrations** less than 300 were measured correctly, but concentrations equal or higher than 300 were overestimated by 20 units!

Your *mission* is to use a loop to correct these measurements for all Mississippi sites.

Tip. Make sure you reload the CO_2 data so that we are working with the raw data for the rest of the exercise:

```
data(CO2)
```

Challenge 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: We could also have written it in this way, which is more concise and clearer:

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

Writing functions

Why write functions?

Imagine that we would like to rescale variables to the range of `0` to `1`.

Our dataset has four variables:

```
our.dataset <- data.frame(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
```

The equation for doing that is:

$$x_{\text{new}} = \frac{x_i - \min(x)}{\max(x) - \min(x)}$$

We can rescale these four variables to `0` and `1` by doing this:

```
our.dataset$a <- (our.dataset$a - min(our.dataset$a, na.rm = TRUE)) /
  (max(our.dataset$a, na.rm = TRUE) - min(our.dataset$a, na.rm = TRUE))
our.dataset$b <- (our.dataset$b - min(our.dataset$b, na.rm = TRUE)) /
  (max(our.dataset$b, na.rm = TRUE) - min(our.dataset$a, na.rm = TRUE))
our.dataset$c <- (our.dataset$c - min(our.dataset$c, na.rm = TRUE)) /
  (max(our.dataset$c, na.rm = TRUE) - min(our.dataset$c, na.rm = TRUE))
our.dataset$d <- (our.dataset$d - min(our.dataset$d, na.rm = TRUE)) /
  (max(our.dataset$d, na.rm = TRUE) - min(our.dataset$d, na.rm = TRUE))
```

What if our dataset had 31 variables? *Should we make this into a challenge?*

Why write functions?

Imagine that we would like to rescale variables to the range of `0` to `1`.

Our dataset has four variables:

```
our.dataset <- data.frame(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
```

The equation for doing that is:

$$x_{\text{new}} = \frac{x_i - \min(x)}{\max(x) - \min(x)}$$

Repeating that equation and this chunk of code 31 times could be a bit tedious:

```
our.dataset$a <- (our.dataset$a - min(our.dataset$a, na.rm = TRUE)) /
  (max(our.dataset$a, na.rm = TRUE) - min(our.dataset$a, na.rm = TRUE))
```

But, we can see that, except from the *input*, the code was pratically the same among the variables. We can then use our ninja abilities and **create a function** that will do that task for us:

```
# our
# secret
# hidden
```

```
rescale01(our.dataset$a)
rescale01(our.dataset$b)
rescale01(our.dataset$c)
```

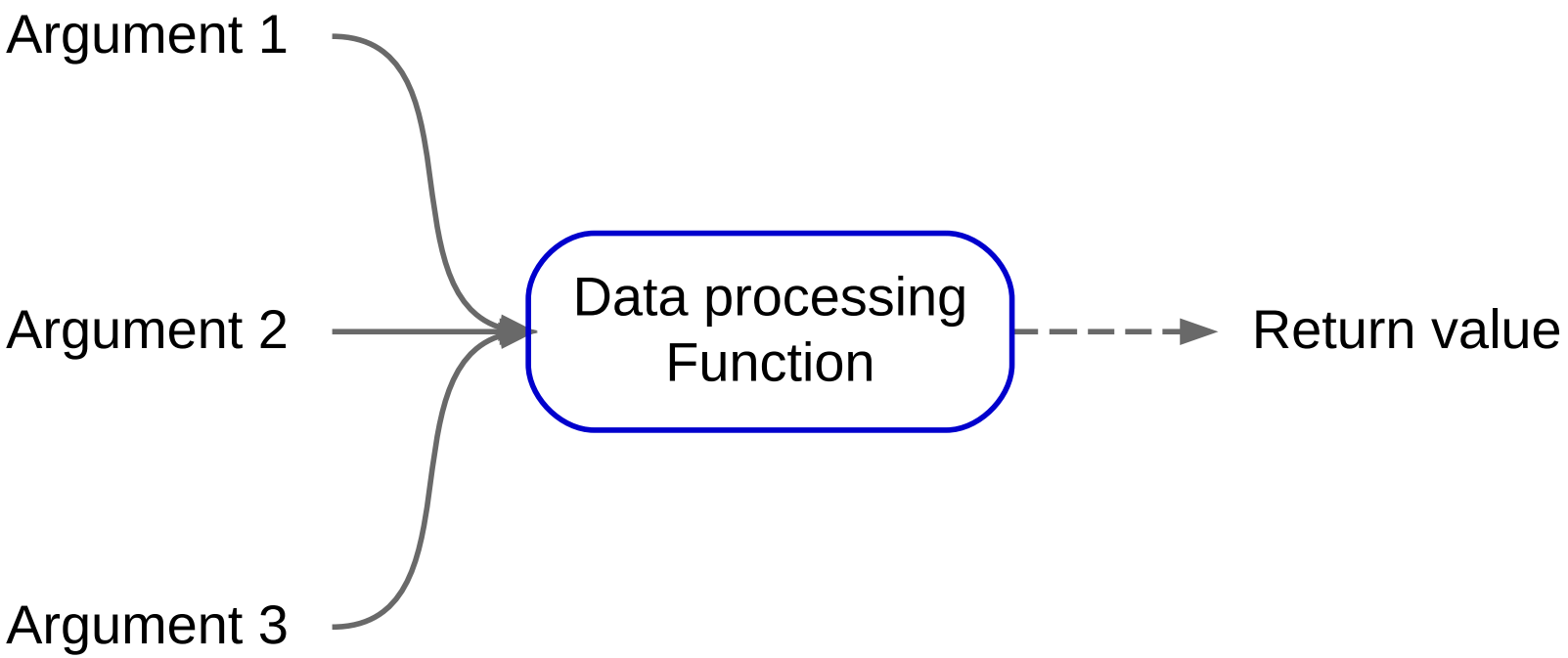
Why write functions?

Much of the heavy lifting in R is done by functions!

Functions are useful for:

1. Performing a task repeatedly, but configurably;
2. Making code more readable;
3. Making code easier to modify and maintain;
4. Sharing code between different analyses;
5. Sharing code with other people;
6. Modifying R's built-in functionality.

What is a function?



Syntax of a function: `function()`

```
function_name <- function(argument1, argument2, ...) {
  body  # What we want the function to do
  return(values)  # Optional
}
```

- `function_name` is the name of the function, and will be stored in the `R` environemnt as an object with this name;
- `argument`s take the defined values that can be used within the function;
- `body` contains the statements that define what the function does;
- `output` contains the returned value from the function. If `return()` is absent, then the last expression is returned.

Arguments of a `function()`

```
function_name <- function(argument1, argument2, ...) {
  body  # What we want the function to do
  return(values)  # Optional
}
```

Arguments are the *input* values of your function and will have the information your function needs to be able to perform correctly.

A function can have between zero and an infinity of arguments. See the following example:

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

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

Challenge 4

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

```
Scruffy <- "dog"
Paws <- "cat"

print_animal(Scruffy)
# [1] "woof"

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

Challenge 4: Solution

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

```
Scruffy <- "dog"
Paws <- "cat"

print_animal(Scruffy)
# [1] "woof"

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

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

Default argument values in a function

Arguments can be provided with a **default value**, or even be optional.

Default values are useful when using a function with the same settings. The flexibility to depart from default values is still there, if needed.

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

operations(number1 = 1, number2 = 2, number3 = 3)
# [1] 9

# is equivalent to
operations(1, 2)
# [1] 9

operations(1, 2, 2) # we can still change the value of number3 if needed
# [1] 6
```

The ellipsis argument: `...`

The special argument `...` allows you to pass arguments from other undefined functions, *i.e.* allowing for an indefinite number of arguments to be inputed.

```
paste_anything_fun <- function(...) {
  arguments <- list(...)
  paste0(arguments)
}
```

```
paste_anything_fun("I",
                   "want",
                   "a break!")
# [1] "I"        "want"
# [3] "a break!"
```

```
percentages <- function(x, mult = 100,
 percent <- round(x * mult, ...)
 paste(percent, "%", sep = "")
}
```

```
percentages(c(.543, .534, .466))
# [1] "54%" "53%" "47%"
```

```
# ?round

percentages(c(.543, .534, .466), digits
# [1] "54.3%" "53.4%" "46.6%"
```

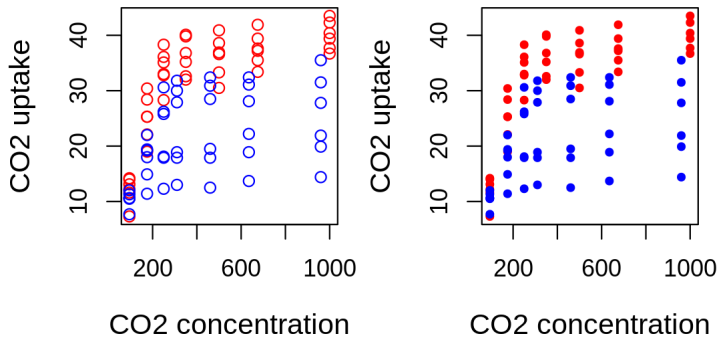
The ellipsis argument: `...`

The special argument `...` allows you to pass on arguments to another function used inside your function. Here we use `...` to pass on arguments to `plot()` and `points()`.

```
plot.CO2 <- function(CO2, ...) {
  plot(x=CO2$conc, y = CO2$uptake, type = "n", ...)
  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 = "CO2 concentration",
         ylab = "CO2 uptake")
```

```
plot.CO2(CO2,
         cex.lab = 1.2, pch = 20,
```



Return values

The last expression evaluated in a `function` becomes the return value:

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

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

`function()` itself returns the last evaluated value even without including `return()` function.

Return values

It can be useful to explicitly `return()` if the routine should end early, jump out of the function and return a value.

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

Functions can return only a single object (and text). But this is not a limitation because you can return a `list` containing any number of objects.

```
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
```

Challenge 5

Using what you have just learned on functions and control flow, create a function named `bigsum` that takes two arguments `a` and `b` and:

1. Returns 0 if the sum of `a` and `b` is strictly less than 50;
2. Else, returns the sum of `a` and `b`.

Challenge 5: Solution

Using what you have just learned on functions and control flow, create a function named `bigsum` that takes two arguments `a` and `b` and:

1. Returns `0` if the sum of `a` and `b` is strictly less than `50`;
2. Else, returns the sum of `a` and `b`.

Answer 1

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

Answer 2

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

Accessibility of variables

It is essential to always keep in mind where your variables are, and whether they are defined and accessible:

1. Variables defined **inside** a function are not accessible outside from it!
2. Variables defined **outside** a function are accessible inside, and are not modified, even if they have the same name.

```
out_val <- 3
vartest <- function() {
  in_val <- 4
  print(in_val)
  print(out_val)
}
vartest()
# [1] 4
# [1] 3
```

```
in_val; out_val
# Error in eval(expr, envir, enclos): o
# [1] 3
```

```
out_val_2 <- 3
vartest <- function(out_val_2) {
  print(out_val_2)
}

vartest(8)
# [1] 8
```

```
out_val_2
# [1] 3
```

What happens in the function club, stays in the function club.

Accessibility of variables

```
var1 <- 3
vartest <- function() {
  a <- 4       # 'a' is defined inside
  print(a)     # print 'a'
  print(var1) # print var1
}

a              # we cannot print 'a' as it exists only inside the function
# [1] 19.36227

vartest()      # calling vartest() will print a and var1
# [1] 4
# [1] 3

rm(var1)       # remove var1
vartest()      # calling the function again does not work anymore
# [1] 4
# Error in print(var1): object 'var1' not found
```

Accessibility of variables

Tip. Be very careful when creating variables inside a conditional statement as the variable may never have been created and cause (sometimes imperceptible) errors.

Tip. It is good practice to define variables outside the conditions and then modify their values to avoid any problems.

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

a + b
```

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

If you had `b` already assigned in your environment, with a different value, you could have had a **bigger** problem!

No error would have been shown and `a + b` would have meant another thing!

Additional good programming practices

Why should I care about programming practices?

- It makes your life easier;
- It helps you achieve greater readability and makes sharing and reusing your code a lot less painful;
- It helps reduce the time you will spend remembering and understanding your own code.

Pay attention to the next tips!

Keep a clean and nice code

Proper indentation and spacing is the first step to get an easy to read code:

- Use **spaces** between and after your operators;
  - `x>=1&x<=10` is more difficult to read then `x >= 1 & x <= 10`
- Use consistently the same assignation operator;
  - `<-` is often preferred. `=` is sometimes OK, but do not switch all the time between the two.
- Use brackets and returns when using flow control statements;
  - Inside brackets, indent by *at least* two returns;
  - Put closing brackets on a separate line, except when preceding an `else` statement.
- Define each variable on its own line;
- Use `Cmd + I` or `Ctrl + I` in RStudio to indent the highlighted code automatically;

Nay!

```
if((a[x,y]>1.0)&(a[x,y]<2.0)){print("Be
```

Yay!

```
if((a[x, y] > 1.0) & (a[x, y] < 2.0))
   print("Between 1 and 2")
 }
```

Keep a clean and nice code

On the left, code is not spaced, nor indented. All brackets are in the same line, and it looks "messy".

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

Keep a clean and nice code

On the left, code is not spaced, nor indented. All brackets are in the same line, and it looks "messy". On the right, it looks more organized, no?

```
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)
  }
}
```

Use functions to simplify your code

Write your own function:

1. When portion of the code is repeated more than a few times in your script;
2. If only a part of the code changes and includes options for different arguments.

This would also reduce the number of potential errors done by copy-pasting, and the time needed to correct them.

Use functions to simplify your code

Let's modify the example from **Challenge 3** and suppose that all CO_2 uptake from Mississipi plants was overestimated by 20 and Quebec underestimated by 50.

We could write this:

```
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
  }
}
```

Or this:

```
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",
```

Use meaningful names for functions

Same function as before, but with vague names:

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

That being said:

Nat "superstar" Alison 🍑
@tesseralis

programmers: nooooo you have to give your variables descriptive names

mathematicians: haha function go f(r,r,r,r,r,r,r,r,r,r)

What is `c` and `rc`?

Whenever possible, avoid using names of existing `R` functions and variables to avoid confusion and conflits.

Use comments:

Final tip. Add comments to describe what your code does, how to use its arguments or a detailed step-by-step description of the function.

```
# Recalibrates the CO2 dataset by modifying the CO2 uptake concentration
# by a fixed amount depending on the region of sampling.

# 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 uptake

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

Challenge 6: Group exercise

Using what you learned, write an `if()` statement that tests whether a numeric variable `x` is `0`. If not, it assigns $cos(x)/x$ to `z`, otherwise it assigns 1 to `z`.

Create a function called `my_function()` that takes the variable `x` as argument and returns `z`.

If we assign 45, 20, and 0 to `x` respectively, which of the following options would represent the results?

1. 0.054, 0.012, and 0;

2. 0.020, 0.054, and 1;

3. 0.012, 0.020, and 1.

In addition to this, discuss with your group about a function that you would like to create to apply (it can or it may not be related to your research). Be prepared to briefly describe it to us!

Group exercise: Solution

Correct answer is option **3** (0.12, 0.20, and 1).

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

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

my_function(20)
# [1] 0.0204041

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

Thank you for attending this workshop!