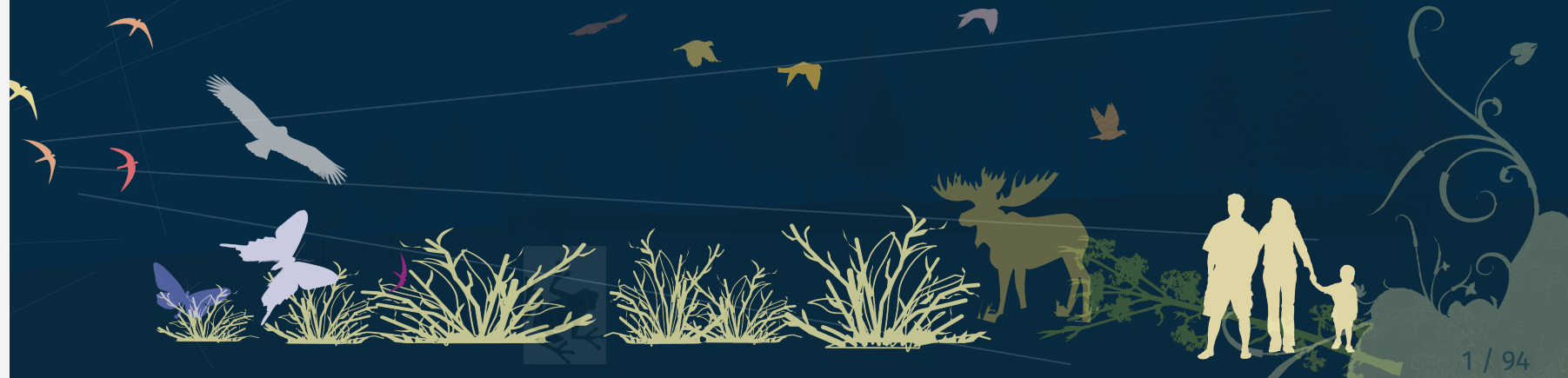




# Workshop 5: Programming in R

QCBS R Workshop Series

Québec Centre for Biodiversity Science



# About this workshop

 REPO	DEV	 WIKI	05	 SLIDES	05	 SLIDES	05	 SCRIPT	05
--	-----	--	----	--	----	--	----	--	----

# Learning Objectives

1. Recognizing **control flow**
2. Developing your first functions in R
3. Accelerating your code
4. 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
```

# 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.

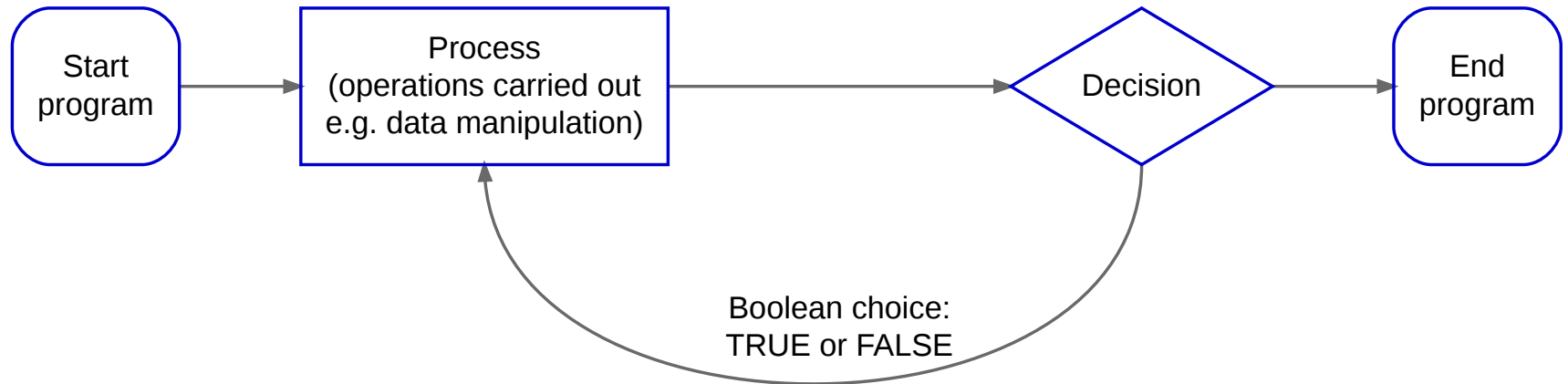
## Why is it advantageous to have structured programs?

- 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

**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

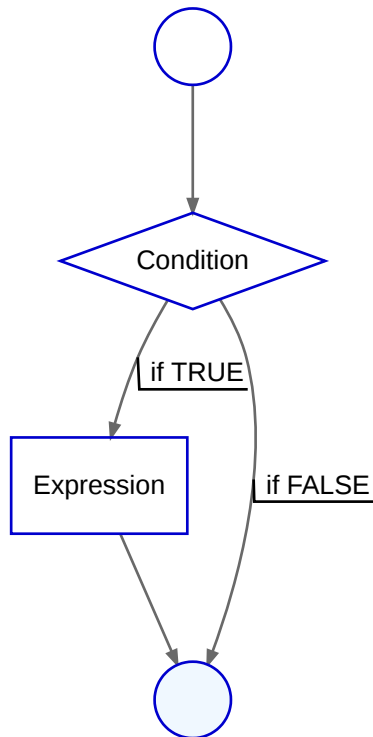


`while` loop

# Decision making

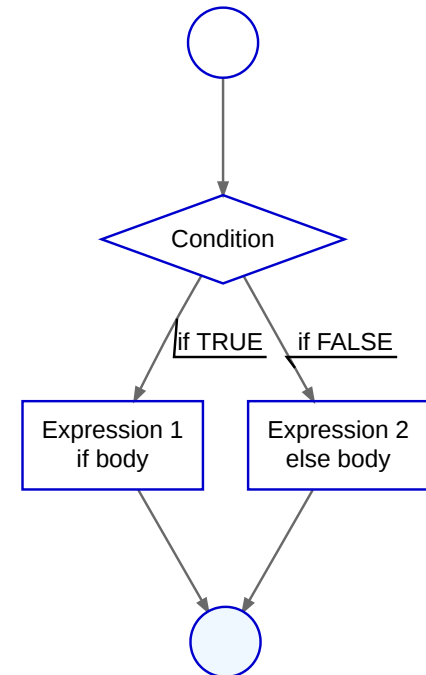
## `if()` statement

```
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

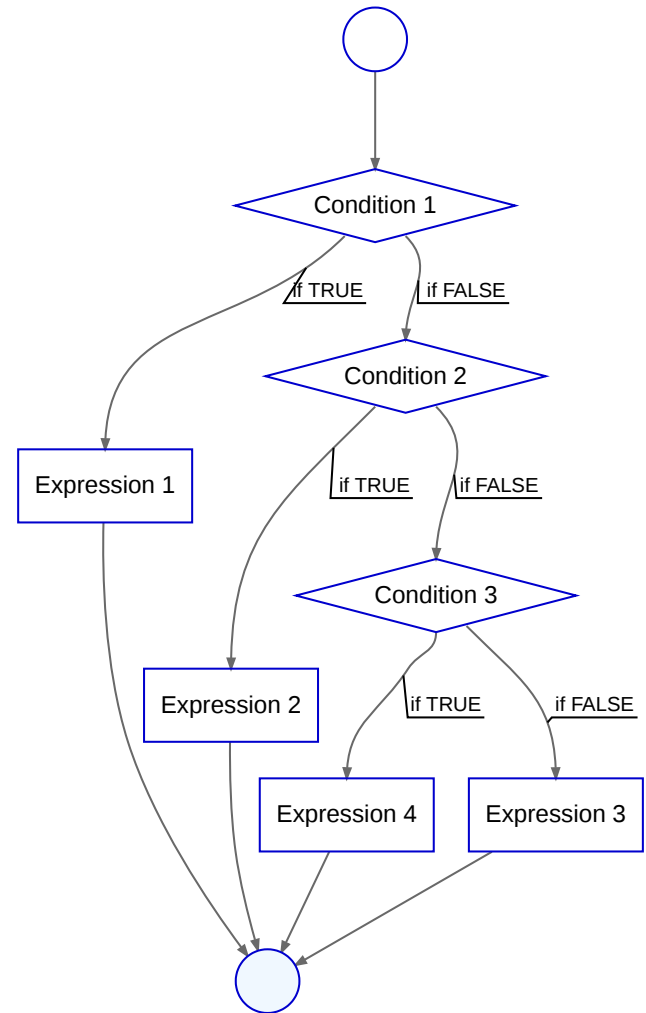
sqrt(ifelse(test = a >= 0,
            yes = a,
            no = NA))

# [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.

# 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"
```

# Remember the logical operators

Command	Meaning
<code>==</code>	equal to
<code>!=</code>	not equal to
<code>&lt;</code>	less than
<code>&lt;=</code>	less than or equal to
<code>&gt;</code>	greater than
<code>&gt;=</code>	greater than or equal to
<code>x&amp;y</code>	<code>x</code> AND <code>y</code>
<code>x y</code>	<code>x</code> OR <code>y</code>
<code>isTRUE(x)</code>	test if <code>x</code> is true

# 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

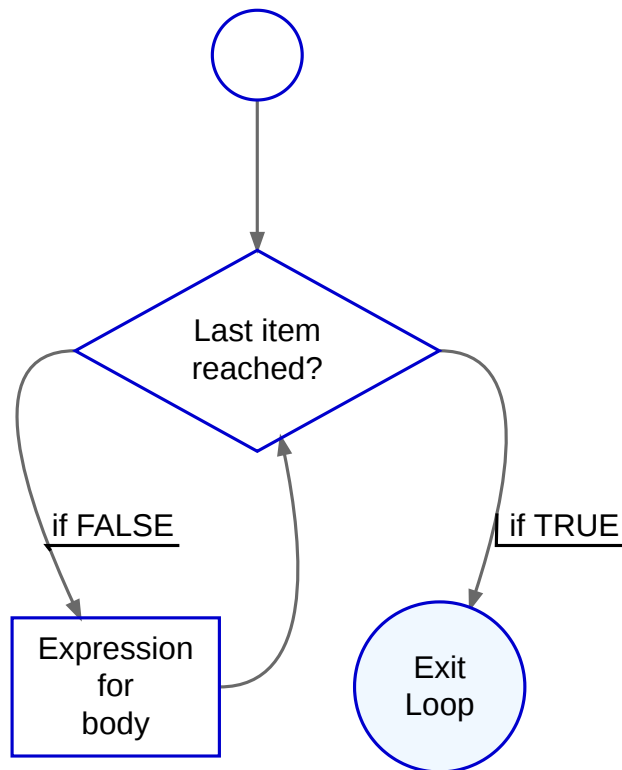


`while` loop

# for() loop

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

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





# for loop

The letter `i` can be replaced with any variable name and the sequence can be almost anything, even a list of vectors.

# Try the commands below and see what happens:

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

# for() loop

In the generic example below, `R` would execute the expression 5 times, each one of them by sequentially replacing `i` by numbers from 1 to 5:

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

`expression` can be anything:

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

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

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

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

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

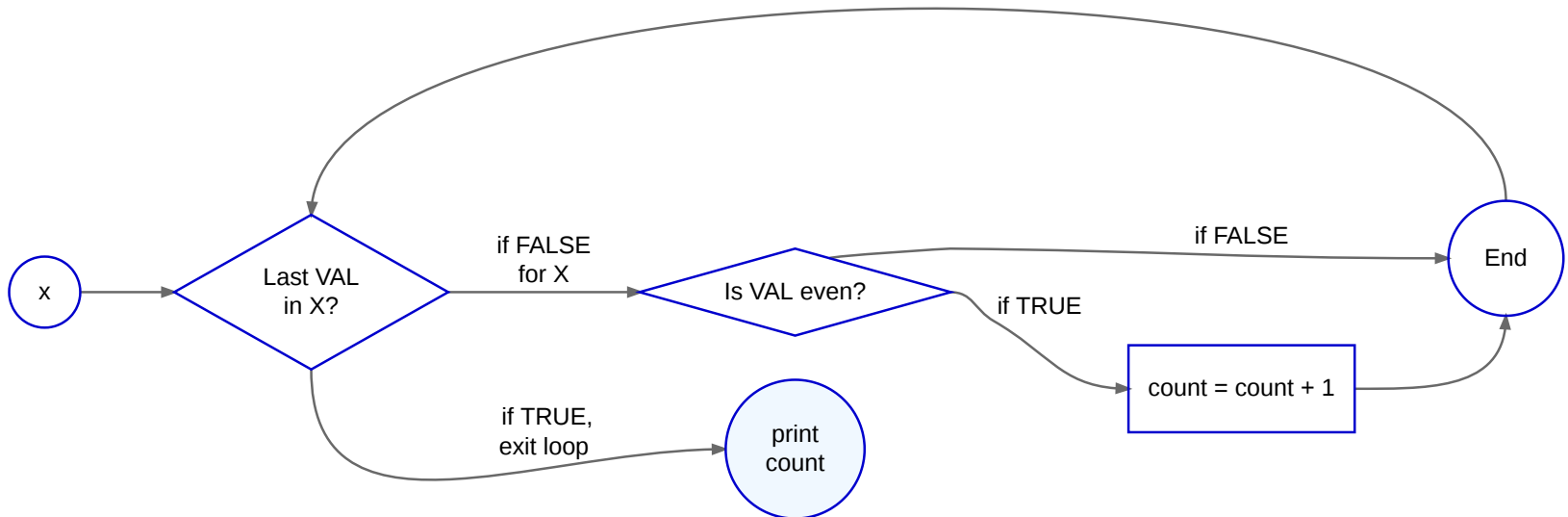
```
# [1] 2  
# [1] 4  
# [1] 6  
# [1] 8  
# [1] 10  
# [1] 12  
# [1] 14
```

# 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 `C02` dataset which is built in `R`.

```
data(C02) # This loads the built in dataset
```

```
for(i in 1:length(C02[,1])) { # for each row in the C02 dataset
  print(C02$conc[i]) # print the C02 concentration
}
```

First 40 outputs:

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

# 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 C02$conc) {  
  # for each element of  
  # the column "conc" of  
  # the C02 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
```



# `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 “1” 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] 0.07491378  
#  
# $Norm20  
# [1] 1.026005  
#  
# $Norm100  
# [1] 4.946315
```

# 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      0.3133652  
#      Norm20        Norm100  
#      0.8849029      4.9426221
```

# 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  $\text{CO}_2$  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  $\text{CO}_2$  uptake in both Québec and Mississippi sampled sites.

For this, you must load the  $\text{CO}_2$  dataset using `data(c02)`, and then use the object `c02`.

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



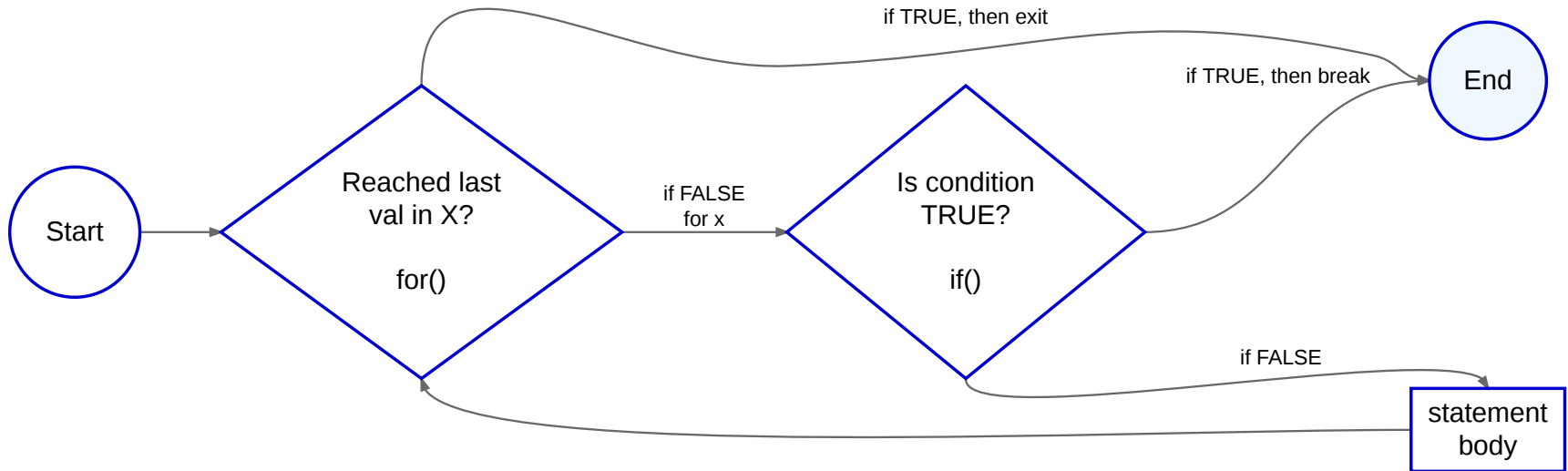
`repeat` loop



`while` loop

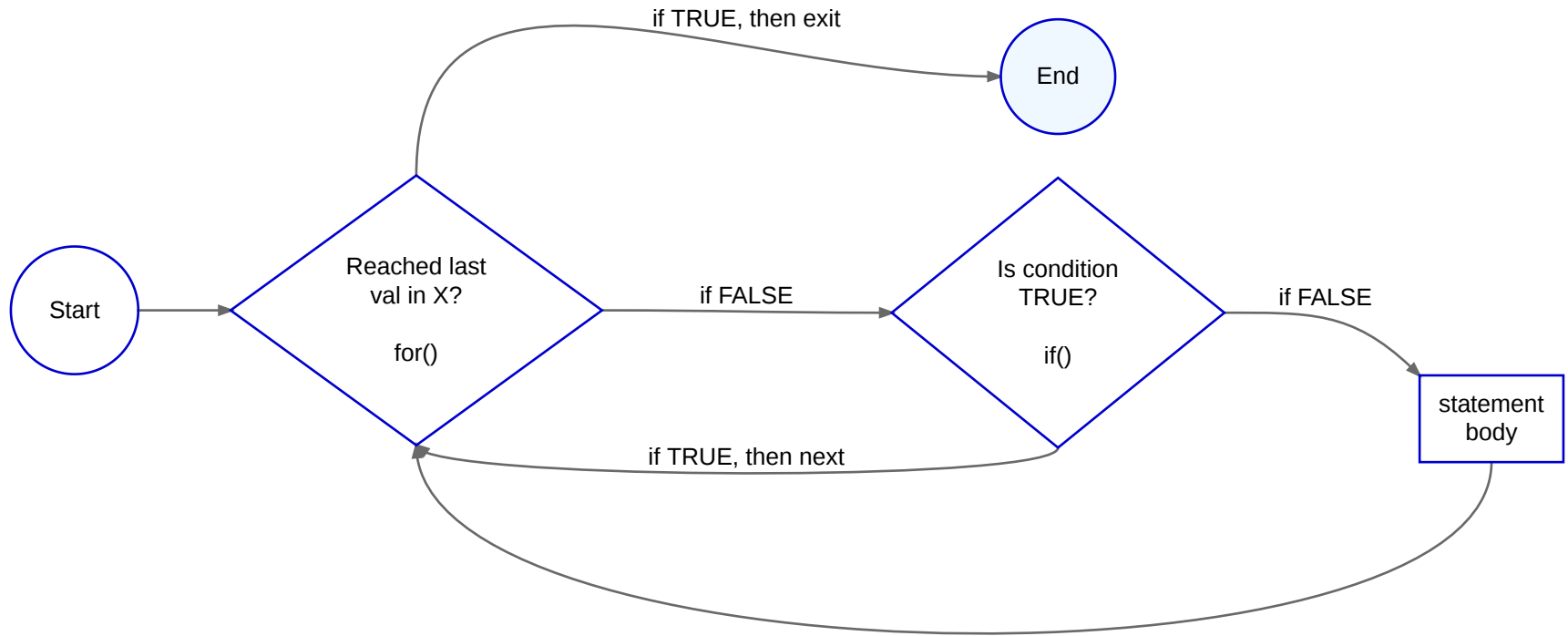
# Modifying iterations: `break` statement

```
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<sub>2</sub> 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 == "nonchilled")
# [1] 42
```

# Control flow roadmap

`if` and `if else` statements



`for` loop



`break` and `next` statements



`repeat` loop



`while` loop

# Modifying iterations: `repeat` loop

This could be equivalently written using a `repeat` loop and `break`:

```
count <- 0
i <- 0
repeat {
  i <- i + 1
  if(CO2$Treatment[i] == "nonchilled") next # skip this loop
  count <- count + 1
  print(CO2$conc[i])
  if(i == nrow(CO2)) break # stop looping
}
print(count)
```

# Control flow roadmap

`if` and `if else` statements



`for` loop



`break` and `next` statements



`repeat` loop



`while` loop



# Modifying iterations: `while` loop

This could also be written using a `while` loop:

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

# 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<sub>2</sub>** 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  
  }  
}
```

# Edit a plot using `for()` and `if()`

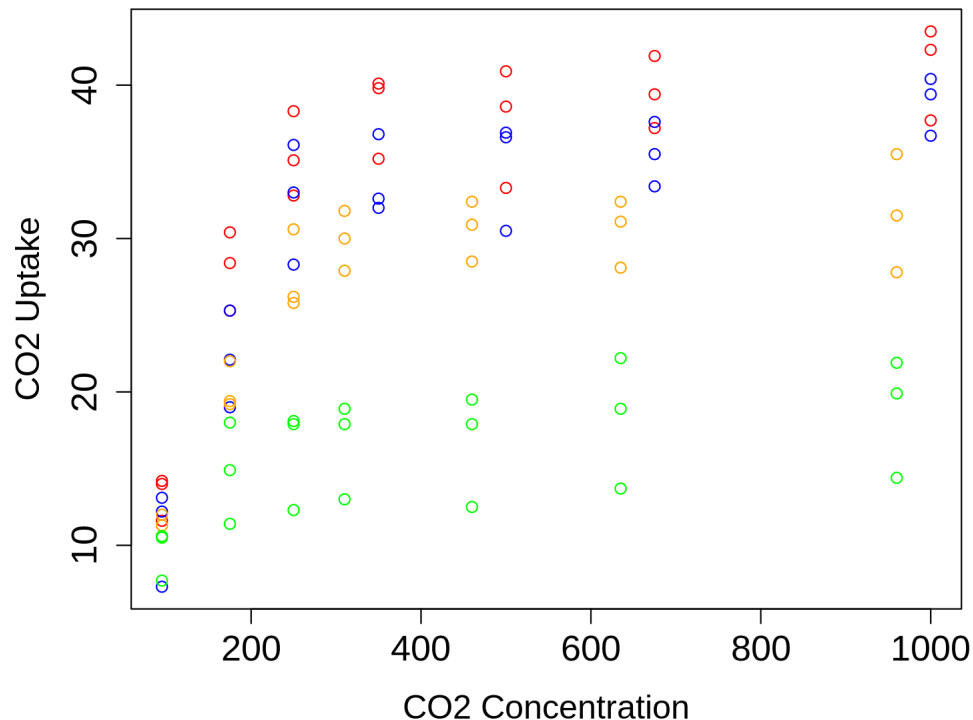
Let us plot **uptake** versus **concentration** with points of different colors according to their type (*Québec* or *Mississippi*) and **treatment** (*chilled* or *nonchilled*):

```
plot(x = C02$conc,
     y = C02$uptake,
     type = "n", # points
     cex.lab = 1.4, cex.axis = 1.5,
     cex.main = 1.5, cex.sub = 1.5,
     xlab = "C02 concentration",
     ylab = "C02 uptake")

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")
  }
}
```

# Edit a plot using `for()` and `if()`

Plotting **uptake** versus **concentration** with points of different colors according to their type using `for()` loop and `if()`.



# Challenge 4



Generate a plot showing **concentration** versus **uptake** where each plant is shown using a point with a different colour.

**Bonus points** for doing it with nested loops!

*Steps:*

1. Create an empty plot;
2. Create a list of plants (hint: `?unique`);
3. Fill the plot using `for()` and `if()` statements.

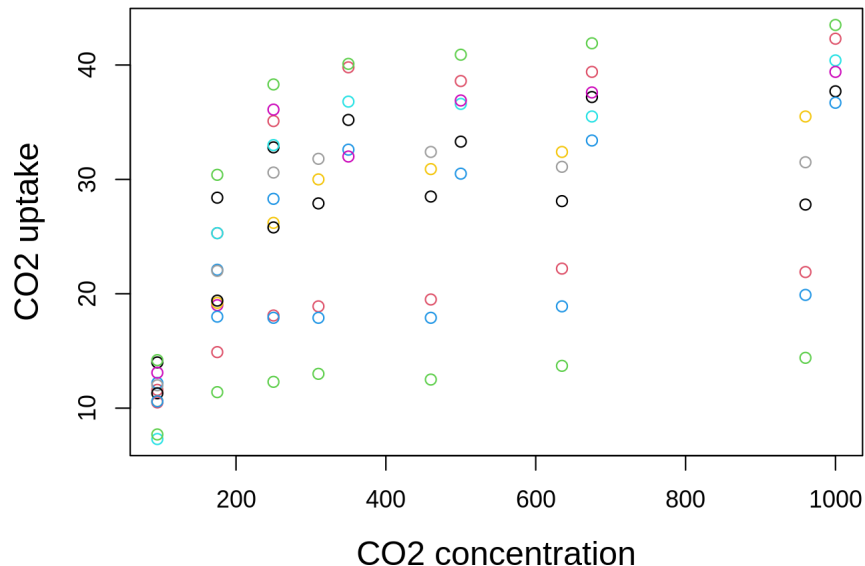
# Challenge 4: Solution



```
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)  
    }  
  }  
}
```



# Writing functions



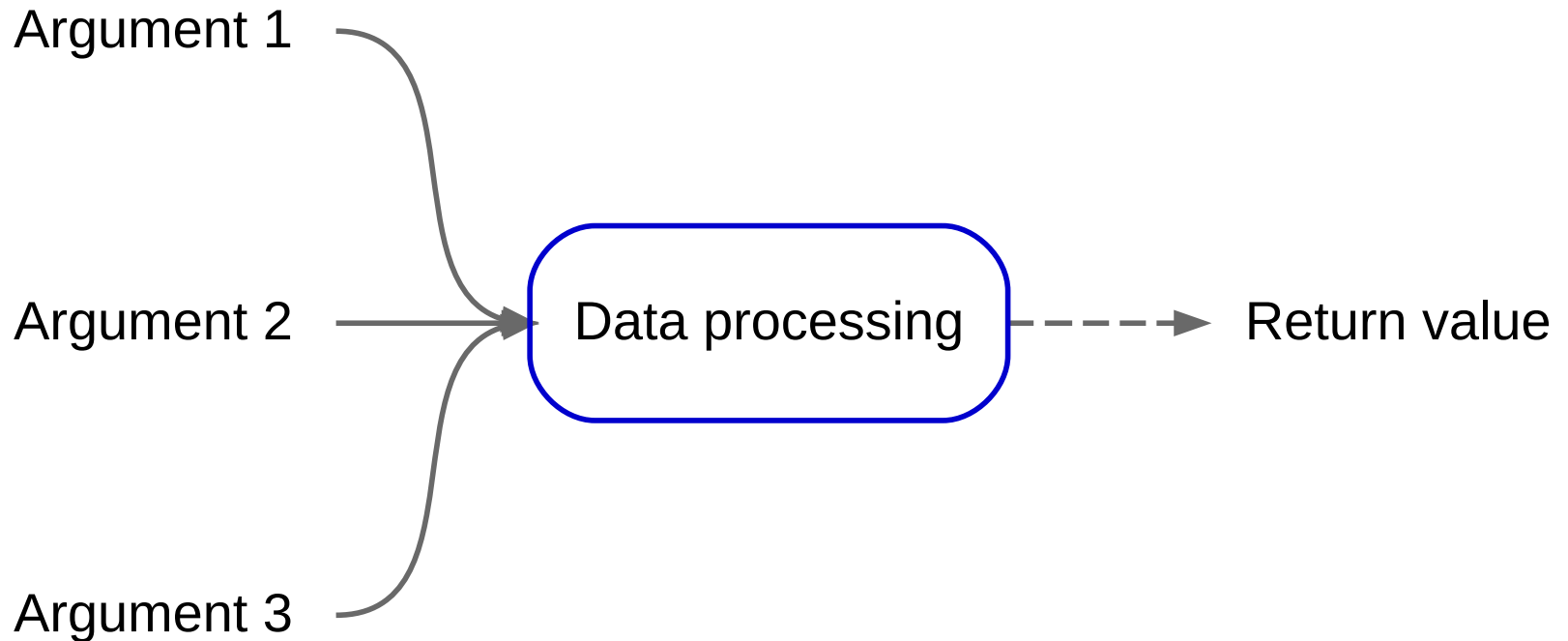
# Why write functions?

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

They 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_name <- function(argument1, argument2, ...) {  
  expression... # What we want the function to do  
  return(value) # Optional  
}
```

# Arguments of a function

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

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

A function can have between 0 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 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:

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

print_animal(Scruffy)
# [1] "woof"

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

# Challenge 5: 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 values in a function

Arguments can also be optional and be provided with a **default value**.

This is useful when using a function with the same settings, but still provides the flexibility to change its values, if needed.

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

```
operations(1, 2, 3) # is equivalent to
```

```
# [1] 9
```

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

```
# [1] 9
```

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

```
# [1] 6
```

# 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", ...)  
    }  
  }  
}
```

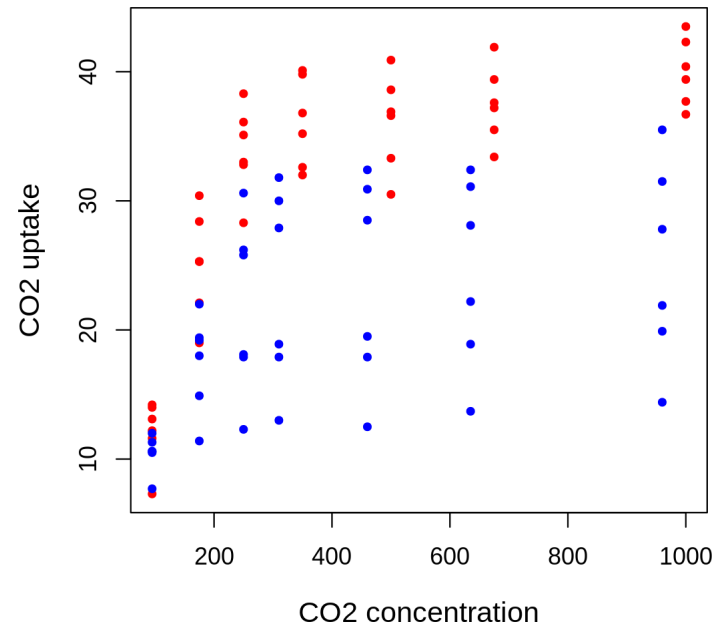
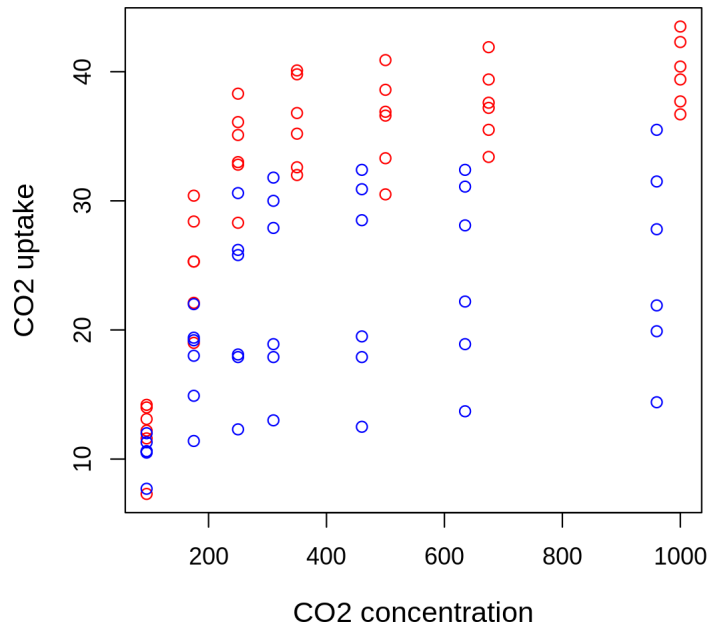
```
plot.CO2(CO2,  
         cex.lab=1.2,  
         xlab="CO2 concentration",  
         ylab="CO2 uptake")
```

```
plot.CO2(CO2,
```



# 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()`.



# Argument

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

```
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
```

# 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 6



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 6: 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:

- ➔ Variables defined **inside** a function are not accessible outside of it!
- ➔ Variables defined **outside** a function are accessible inside. But it is NEVER a good idea, as your function will not function if the outside variable is erased.

# Accessibility of variables

```
var1 <- 3      # var1 is defined outside our function
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] -4 -3 -2 -1 0 1 2 3
# [9] 4 5

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

**Mandatory tip.** Use arguments then!

Also, inside a function, arguments names will take over other variable names.

```
var1 <- 3      # var1 is defined outside our function

vartest <- function(var1) {
  print(var1) # print var1
}

vartest(8)     # Inside our function var1 is now our argument
# [1] 8

              # and takes its value

var1          # var1 still has the same value
# [1] 3
```

# 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 problem

```
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!

# Good programming practices

# Why should I care about programming practices?

- To make your life easier;
- To achieve greater readability and makes sharing and reusing your code a lot less painful;
- To reduce the time you will spend to understand your 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;
- Use consistently the same assignation operator.
  - `<-` is often preferred. `=` is sometimes OK, but do not switch all the time between the two;
- Use brackets when using flow control statements:
  - Inside brackets, indent by *at least* two spaces;
  - 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.

# 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;
- Use consistently the same assignation operator.
  - `<-` is often preferred. `=` is sometimes OK, but do not switch all the time between the two;
- Use brackets when using flow control statements:
  - Inside brackets, indent by *at least* two spaces;
  - 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("Between 1 and 2")  
}
```

**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. 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. 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 twice 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 Mississippi 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 = "Mississippi", bias = -20)  
newCO2 <- recalibrate(newCO2, "Quebec", bias = 50)
```

# 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)  
}
```

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

```
# Recalibrates the C02 dataset by modifying the C02 uptake concentration
# by a fixed amount depending on the region of sampling.

# Arguments
# C02: the C02 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(C02, type, bias) {
  for(i in 1:nrow(C02)) {
    if(C02$Type[i] == type) {
      C02$uptake[i] <- C02$uptake[i] + bias
    }
  }
  return(C02)
}
```

# Group exercise

Using what you learned, write an `if()` statement that tests whether a numeric variable `x` is zero. 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. `0.54`, `0.12`, and `0`;
2. `0.20`, `0.54`, and `1`;
3. `0.12`, `0.20`, and `1`.

# 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!**

