Loops and Functions in R

Instructor: Alfredo Ascanio

E-mail: ascaniaa@miamioh.edu

Assistant instructors: Tereza Jezkova, Meelyn Pandit

1 GOALS

- Understand and built different types of loops (Monday)
  - Conditional statements
  - Structure of loops (mainly for-loop)
  - Uses
- Create your own functions (Tuesday)
  - Input and Output
  - Arguments/Parameters
  - Default values
  - Messages: warnings and errors

2 DATA

- Randomly generated data for initial examples
- Ecological data and maps for real-world applications: Species occurrences extracted from GBIF database
- **Install java JDK and Run script "Install_Packages.R" (both in the flash drives) beforehand to save time.**

3 RECOMMENDED PREVIOUS KNOWLEDGE

- Data types in R: numeric, character, factors, logical values, etc.
- Base R objects:
  - Vectors, Matrices, Lists, and Data Frames
  - Subsetting
- Using functions[1]
- Conditionals and Logical Operators[1]

[1] **WE WILL REVIEW SOME OF THESE CONTENTS**

4 WHY LOOPS AND FUNCTIONS?

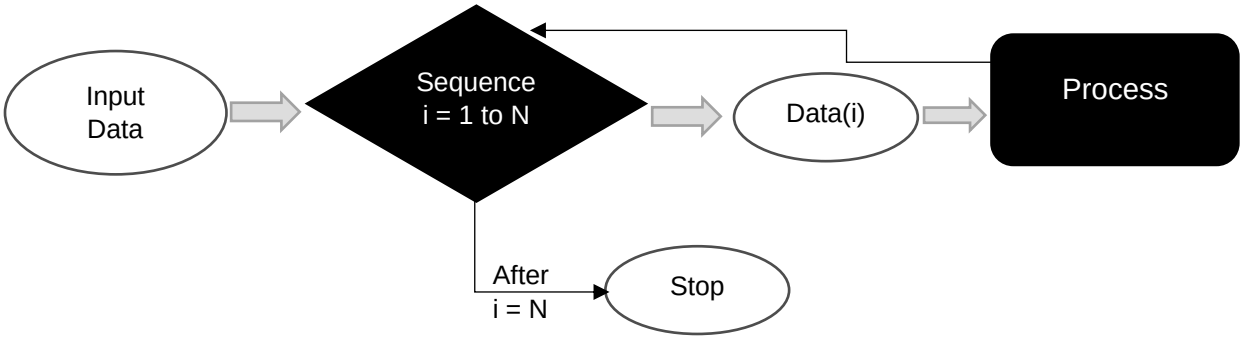
"If you need to repeat a process more than twice, program it"

While doing your work, have you found yourselves doing the same steps multiple times on different subsets of your data, or in multiple datasets with similar formats?

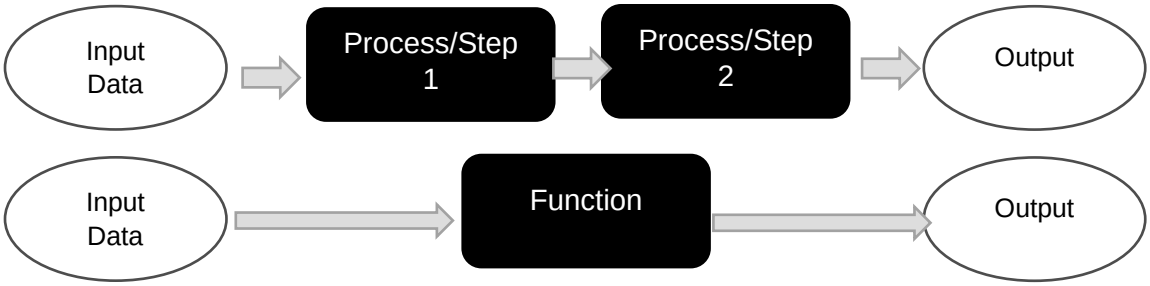
That's why understanding how to build loops and functions is important. Some of the advantages of coding your processes in this way are:

- Not having to make everything by hand
- It takes time at first, but it will save a lot of time later
- Shareability
- Transparency
- Ability to retrace your steps in the process and make corrections or updates

- **Loops** will help us repeat a process multiple times, e.g. on different datasets (or subsets of them) considering that their format remains the same. Loops are processes themselves.



- **Functions** will help us simplify those processes, that tend to be comprised by multiple and sometimes complex steps, into a couple lines of code.

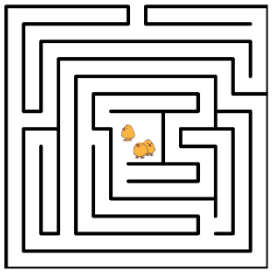


5 CONDITIONALS: IF-ELSE LOGIC

Decisions accounted for during a process

One important element within loops and functions are conditional statements. These statements, usually called "conditionals", can help us integrate decisions into the processes that we are using or building.

- Examples:
  - Games: **If** the coin turns head **then** I win, otherwise **(else)** I lose
  - Finding a path in a maze: **If** the path on the right is open, **then** I will go through it, otherwise **(else)** I will go left.
  - Daily decisions: **If** the food smells good, **then** I will eat it, otherwise **(else)** I will not.




The **If** statement represents an event that must happen for a process (**then**) to occur. Whenever the event does not happen (**else**), we have the option to add a secondary process to occur.

When writing a conditional always write your If-Else statement, try to remember the examples mentioned above. *Pen and paper are, and will be, your best friends* when making your own conditionals, loops, and functions.

5.1 HOW DO WE WRITE A CONDITIONAL?

1) First, let's create/open an R project in Rstudio. The flash-drive folder for this course already contains one, you can double click on it.
2) Put the CONDITIONAL.R script in your flash drive, on Project Folder.
3) Open the R script
4) **Let's make a game! – Coin Tossing Game**: This is a game with a random component, with binomial probability of success equal to 0.5. This means that we will have 50% chance of success or failure at each coin throw. **In the space below try to write the three steps for a coin toss game conditional where you can "win" or "lose"**

   1) **If –** _____
   2) **Then –** _____
   3) **Else –** _____

Now, *my solution* for the question above, would be as follows:

1) **If** the coin falls as I wanted
2) **Then** I win
3) **Else** I lose

I highlight that as my solution because *there can be more than one correct way to solve a problem*. What we must always consider is what information or data we have at the beginning of a process (**input**) and what is the expected and correct result of that process (**output**). The process in between may change between person to person, according to knowledge, coding experience, and preferences on how to handle the information.

5.2 How do we write a conditional in R?

Following with the same example of the **coin toss game**, we need to consider what R requires for it to understand our If-Else statements and the processes that must occur.

1) **If** the coin falls as I wanted – Must be a logical condition **(TRUE/FALSE)**
2) **Then** I win – Process to occur **if previous condition was TRUE**
3) **Else** I lose – Optional process to occur **if previous condition was FALSE**

We will first simulate a coin toss. As I said before, this is a random process that follows a binomial probability of success equal to 0.5.

```
> coin_toss <- rbinom(n = 1, size = 1, prob = 0.5)
# Simulates a coin toss, with 0.5 probability of success
# Results will be 0 (failure) or 1 (success) following a binomial probability
```

1) **If** the coin falls as I wanted – Must be a logical condition **(TRUE/FALSE)**
2) **Then** I win – Process to occur **if previous condition was TRUE**

```
> if(coin_toss == 1) { print("You win") }
# if condition within () results to TRUE --- then process within {} will run
```

3) **Else** I lose – Optional process to occur **if previous condition was FALSE**

```
> } else { print("You Lose")} # only if the condition within () results to FALSE
```

In summary, we have:

```
# Simulate a coin toss
> coin_toss <- rbinom(n = 1, size = 1, prob = 0.5)
# Set condition for success, define process to occur if conditions
# are TRUE or FALSE and corresponding outputs
if(coin_toss == 1) { print("You win")
        } else { print("You Lose")}
```

- **Exercise 1:** What if we want the output to be Head or Tails, instead of winning or losing? What if the coin is rigged, favoring heads over tails? Change the code to make it happen

```
# Simulate a coin toss
> coin_toss <- rbinom(n = 1, size = 1, prob = _____)
# Set condition for success, define process to occur if conditions
# are TRUE or FALSE and corresponding outputs
if(coin_toss == 1) { print("_____")
        } else { print("_____")}
```

5.3 Extensions to the If-Else Logic

1) The process within a conditional statement will run if, and only if, the condition results to TRUE (a logical value in R)

```
> if(TRUE) { print("Execute process 1!")

        } else { print("Execute process 2!")}
```

2) If the condition within the if() function results to FALSE, then the secondary and optional process will run instead.
3) We can add as many conditions as we needed

```
> if(condition_1 == TRUE) { print("Execute process 1!")

        } else if(condition_2 == TRUE) { print("Execute process 2!")

        } else {print("Final process if everything else fails"}
```

5.4 Logical operators and functions with logical outputs

Conditionals depend on how we create a logical vector of length one. We can see this as asking the computer a question, which answer can only be TRUE or FALSE. This can be achieved through the use of **logical operators**. These logical operators work with different types of data, e.g. comparing strings of characters, or factors, or dates, etc.

Aside from logical operators, there are several **functions with logical outputs**, that we can use to ask other types of questions.

Logical Operators

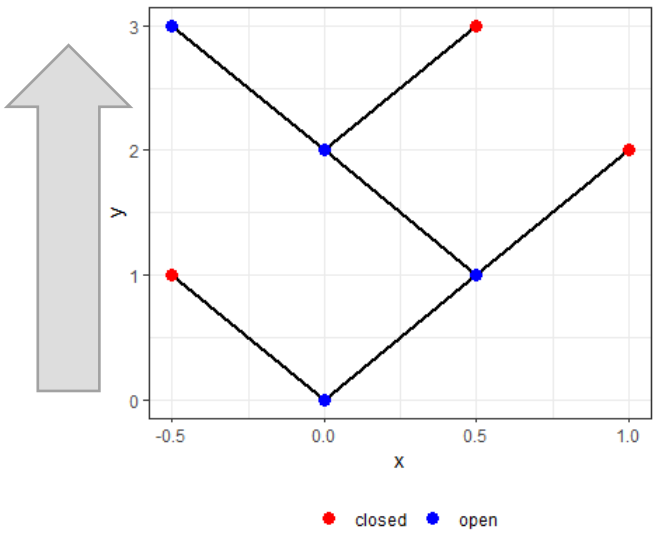
> x <- 5; y <- 8
> x > y *#is x greater than y? Results to FALSE*
> x < y *#is x less than y? Results to TRUE*
> x >= y *#is x equal to or greater than y? Results to FALSE*
> x <= y *#is x equal to or less than y? Results to TRUE*
> x == y *#is x equal to y? Results to FALSE*
> x != y *#is x different to y? Results to TRUE*
> X %in% y *#is the value of x contained in the values of y? Results FALSE*

Functions with logical outputs

> indetical(x, y) #if two objects are exactly the same returns one TRUE
> any() #Input is a logical vector, if any value is TRUE, returns one TRUE
> all() # input is a logical vector, if all values are TRUE, returns one TRUE
> is.() *#Family of functions*
    > is.numeric() # if vector or matrix is numeric, returns TRUE
    > is.integer() # if vector or matrix is integer, returns TRUE
    > is.vector() *# if object is vector, returns TRUE, etc…*
#There are many more that you will discover as you start needing them

5.5 EXERCISE 2: LET'S CHOOSE A PATH!

Imagine a maze in which each time you give a step, it bifurcates in two paths, one closed and one open. I want you to determine whether you should go left or right depending on where you are in the maze, and using a conditional in R.



I provided you with a function to create a random maze. It has two outputs:

- 1) the maze plot, and
- 2) a data frame that we can call our “maze object”

"Row"	"y"	"x"	"open"	"xinit"	"yinit"
"1"	0	0	1	0	0
"2"	1	-0.5	0	0	0
"3"	1	0.5	1	0	0
"4"	2	0	0	0.5	1
"5"	2	1	1	0.5	1
"6"	3	0.5	1	1	2
"7"	3	1.5	0	1	2

Run the function on your own computer, store and explore the maze object and the plot. This function has a random component, similar to our coin toss game, so the results we see may be different.

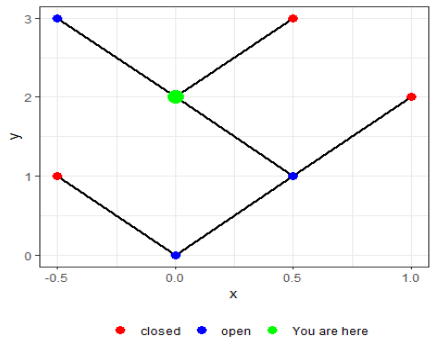
> **source(Maze_functions.R)** *# loads the functions for the maze*
> **Maze <- make_maze()** *# Produces a maze plot and data frame for 3 steps*

Imagine that you are located at any open path within the maze (except for the last one) and think about what question should you ask the computer to make the decision to move Left or Right?

Now run a second function where_are_you() on your object Maze, storing the output in a new object called where_at

> **where_at <- where_are_you(Maze)**
> *# locates you on one bifurcation within your maze*

This function has two outputs: a plot, so you can visualize where you are on your Maze, and a smaller data frame, showing only the row where you are (in green) and the next two rows or paths from which you have to choose (one closed and one open).



Row	y	x	open	xinit	yinit
3	1	0.5	2	0	0
4	2	0	0	0.5	1
5	2	1	1	0.5	1

1) **Think for a moment while exploring the smaller data frame "where_at**". In this data frame, what indicates whether a path is to the right or left, from the bifurcation you are in? As in the first examples, **write your conditional statements using your own words:**

If: _____

_____

Then: _____

Else: _____

2) **Once you wrote it in your own words, try to write the code in R. You'll find some space in the script**

Hints: Think of a way to identify whether the open path is "Left" or "Right" of the closed path. Which variables or columns in the "where_at" table gives you that information?

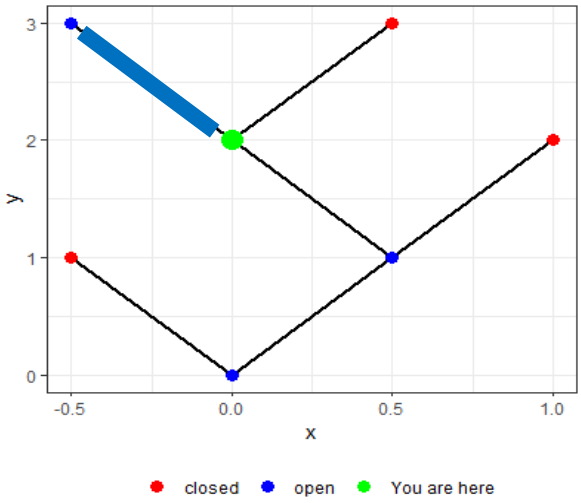
Again, there is more than one way to solve this. *My solution* to this problem would be as follows:

If the value of X where the path is open is lower than the value of X where the path is closed

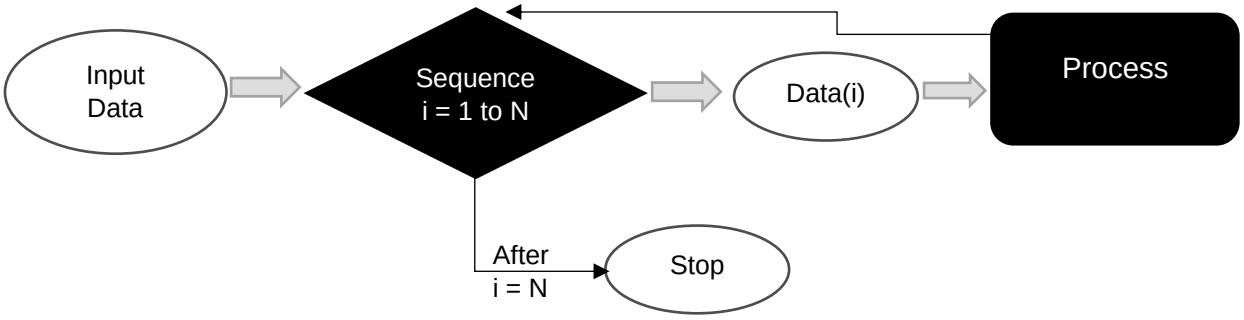
Then the solution should be "Right"

Else, the solution should be "Left"

```
> > if(where_at$x[where_at$open == 1] > where_at$x[where_at$open == 0]) {
        print("Right") } else {
              print("Left")
}
> "Left"
# loads the functions for the maze
# Produces a maze plot and data frame for 3 steps
```

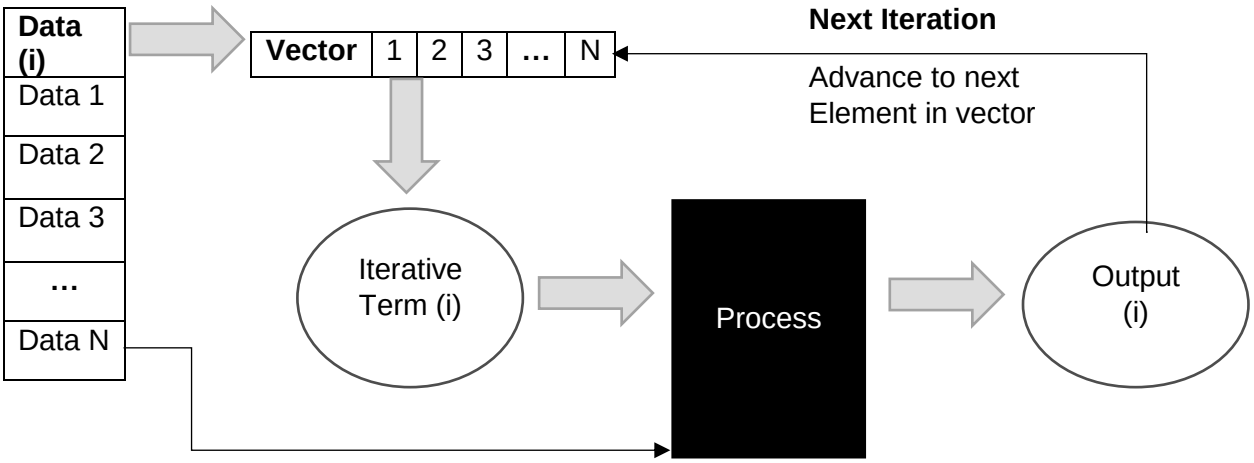


6 LOOPS: FOR-LOOP



A **For-loop** is one tool we can use to repeat a certain process a set number of times. The number of times (or cycles) the loop will execute its internal process are called **iterations**. Some characteristics of **for-loops in R** are:

- The number of iterations is predetermined since the beginning.
- You need to provide what we will call an **iterative term** and a **vector** as input
- Internally, the value of the **iterative term** will change at each new cycle of the loop, sequentially assigning it the values of the input **vector.**
- The **vector** could be of any data type



Structure of a for-loop

- for statement
- Iterative term
- Vector or sequence
- Process

```
> for(i in 1:10) {
        print(i)
}
# for() is the function that will initialize the loop
# i is the iterative term
# 1:10 is a numeric vector from 1 to 10
# print(i) is the process
```

<u>Quick Exercise in R</u>: Change the input vector and the name of the iterative term

a) You can add a mathematical operation to the process (if vector is numeric)

b) You can make the vector a character vector. See what happens.

6.1 IDENTIFYING THE PROPER ITERATIVE TERM

The proper use of the iterative terms and vectors will allow you to be more efficient each time you need to write a for-loop. It depends on two things: 1) the process you need to execute, and 2) the type of object you have.

- By vector (The **vector** could be of any data type)

Vector	I = 1	I = 2	I = 3	…	I = N

- By column (in matrix or data frame)

I = 1	I = 2	…	I = N

- By row (in matrix or data frame)

I = 1			
I = 2			
…			
I = N			

- By factor (in matrix or data frame)

Var1	Var2	Factor	Iteration
		A	I = 1
		A	I = 1
		B	I = 2

- There are many other types of objects that you will discover as you learn more.

6.2 PRACTICE WITH FOR-LOOPS

I will give you a couple R objects on which a process that needs to be repeated for certain elements. You need to solve these problems using a for-loop (there are simpler solutions to some of those, but the idea is that you learn how to construct the loops)

6.2.1 For-loop print elements from character vector – Guided

R provides us with a vector of lowercase or uppercase letters ("letters" and "LETTERS", respectively). Create a for loop that print each letter in either of those vectors.

```
# Option 1 – Using the same vector LETTERS as the sequence for the iterative term
> for(i in LETTERS) {
        print(i)
}

# Option 2 – Using the number of elements and their positions in the LETTERS vector
> for(i in 1:length(LETTERS)) {
        print(LETTERS[i])
}
```

6.2.2 For-loop math on a vector

Create a vector called math and perform the following process to each element:

1) Sum 2
2) Print the output

```
# Create a vector called math
> math <- c(5, 8, 10, 13, 14, 17, 21, 24)
```

With the same math vector, perform the following operation

1) Sum 2
2) Divide by 3
3) Raise to the power of the original number
4) Print output

6.2.3 For-loop by factor on a biological dataset

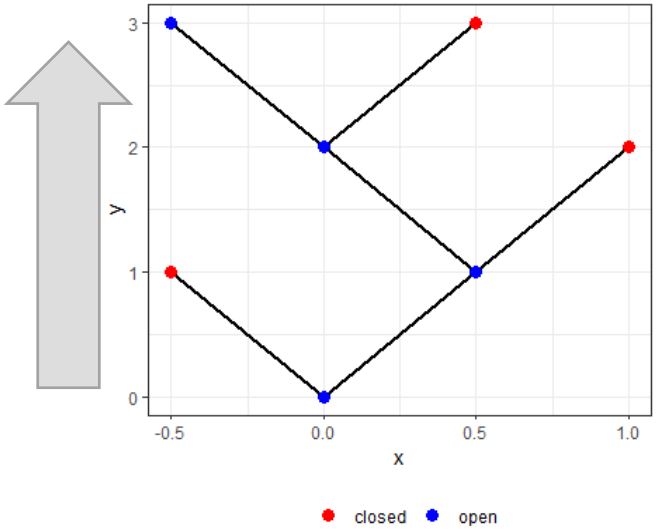
R provides us with a dataset called "iris", which contains information on the morphology of three flowers: *Iris setosa, I. virginica,* and *I. versicolor*. The dataset contains observations on 150 flowers, 50 per species. Create a loop to extract the mean values of Sepal and Petal length for each flower

```
# Assign Iris data to ''Data'' object
> Data <- iris
```

6.3 EXERCISE 3: LET'S FIND THE EXIT OF THE MAZE!

Using the same functions as in exercise 2, create a maze and using a for-loop find the solution of the maze

- 1) Identify an iterative term and sequence vector
  - Look for a column in the "Maze" object that represents a process, i.e. one decision in this case.
- 2) Define the process you need to make at each change of the iterative term
  - You can modify and use the same conditional you created for the maze before
- 3) Construct the loop
  - Use the same for-loop structure we have used in the previous examples and exercises



"Row"	"y"	"x"	"open"	"xinit"	"yinit"
"1"	0	0	1	0	0
"2"	1	-0.5	0	0	0
"3"	1	0.5	1	0	0
"4"	2	0	0	0.5	1
"5"	2	1	1	0.5	1
"6"	3	0.5	1	1	2
"7"	3	1.5	0	1	2

6.4 EXERCISE 4: LET'S JOIN DATA FROM DIFFERENT SOURCES!

In the future you may face the need to merge several data sets with similar format into one. This may be because information from different sampling years or sites were put into different worksheets; or there are several teams working on collecting the data; or the format may have change and you have data before and after that change. The data I provided to you in the course folder was obtained from GBIF, there are three species and you may choose whichever you like the most. I divided the original dataset into smaller subsets by source (museums, individual collections, etc.)

Using what you have learned and the functions I'll provide to you, construct a loop to merge tables from different sources. Remember:

- **Identify the iterative term and vector**
  - One process per file to read
  - Iterative term could either be a number per file or a file name
  - Thus, the vector can be a numeric sequence from 1 to the number of files
  - Or the vector can be a character vector from file names
  - This decision will depend on the rest of the process and what is more useful to you.

Functions to use:

```
# Create a NULL object
> full_data <- NULL

# Read csv file
> Data1 <- read.csv("file_path1.csv", stringsAsFactors = FALSE)
> Data2 <- read.csv("file_path1.csv", stringsAsFactors = FALSE)

# Join data frames by row
> full_data <- rbind(full_data, Data1, Data2)
# NULL objects can be joined with other objects
```

Exercise extension: If you have time, add some filters to each dataset before merging the data:

1) By year (keep occurrences after 1970)
2) By coordinate system (Only WGS84)

My solution to this problem looks as follows:

```
> files <- list.files("PATH_TO_FOLDER") # Make list of files
> full_data <- NULL # Make empty object
> for(i in 1:length(files)) {
        data <- read.csv(files[i], header = T, stringsAsFactors = F) # Read data
        # Filter data
        filtered_data <- data[data$year > 1970 & data$coord.sys == "WGS84", ]
        # Attach data
        full_data <- rbind(full_data, filtered_data)
} #  Close loop
```

7 WHILE AND REPEAT LOOPS

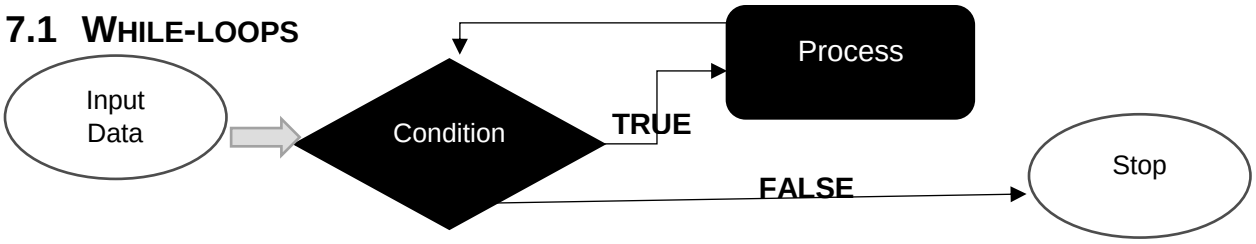
We won't go too deep into the While and Repeat loops, because they are used less often than for-loops for daily problems you may face. However, they can be pretty useful when doing mathematical and statistical analysis and modeling.

The main differences between the for-loop and these two are:

- There is no set number of iterations
- There must be a condition for the loops to execute (While-loop) or break/stop (Repeat-loop)

One of the reasons these loops appear less often is that we tend to know how many iterations we need for a certain process, specially while handling and cleaning data. The other reason is that, as the number of iterations is not set since the beginning, we may create infinite loops, i.e., loops that do not stop at any point and keep always repeating a process. Thus, we need to make sure that the condition we set for initializing or breaking the loop will be met eventually.

7.1 WHILE-LOOPS



In the while-loops, we need a conditional statement at the beginning. As its name implies, while the condition is met (TRUE) the process will keep running, otherwise it will be stopped.

Example:

Imagine we have a variable X = 1. While X remains less than 10, we will sum 1 to it.

```
# Create X variable
> x <- 1

# While-loop
> while(x < 10) {
        x = x + 1
        print(x)
}
```

Quick Exercise:

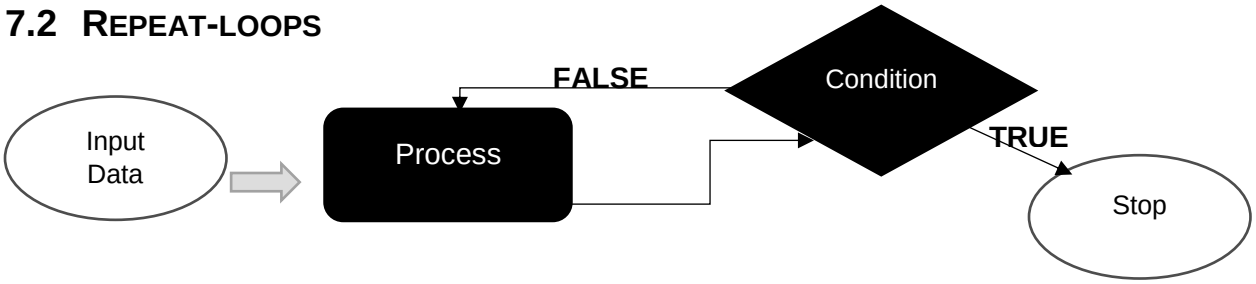
Let's apply the coin toss analogy here. We start with a variable X = 1. Each time we flip a coin successfully (coin_toss == 1), we will sum a unit to X. This process will run while X remains less than or equal to 10.

```
# Remember the coin_toss
> coin_toss <- rbinom(n = 1, size = 1, prob = 0.5)

# Create X variable
> x <- 1

# While-loop
> while(x _____) { # Add a initial condition
        _____ # Add a coin_toss
        _____ # Add a conditional
        _____ # Add a print of x
}
# In this case, ther is no need to add an ''else'' statement, as nothing should happen
to X if we fail the coin_toss
```

7.2 REPEAT-LOOPS



In the repeat-loops, we need a conditional statement at some point during or at the end of the process. As its name implies, this loop will repeat its internal process until the condition is met (TRUE), whenever that happens, the loop will stop.

Example:

Imagine we have a variable X = 1. Sum 1 to X until X equals 10

```
# Create X variable
> x <- 1

# Repeat-loop
> repeat {
        x = x + 1
        print(x)
        if(x == 10) {break()}
}
# We use the function break() after a conditional to stop the loop
```

Quick Exercise:

Transform the previous coin_toss while-loop into a repeat loop, i.e. repeat the loop until X is less than or equal to 10

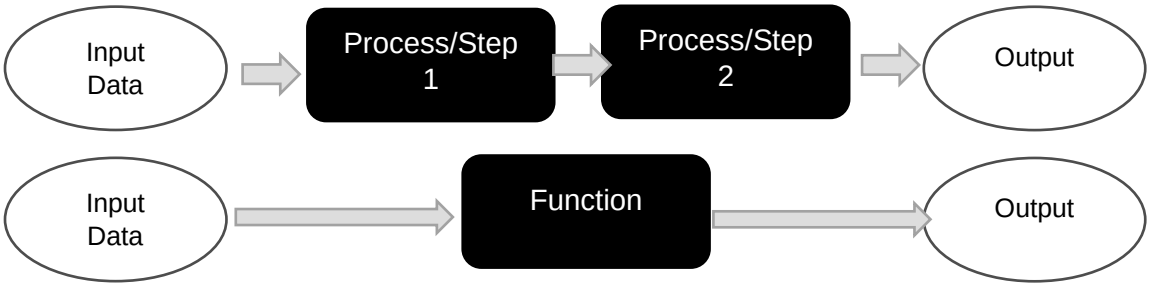
```
# Remember the coin_toss
> coin_toss <- rbinom(n = 1, size = 1, prob = 0.5)

# Create X variable
> x <- 1

# While-loop
> repeat {
        _____________ # Add a coin_toss
        _____________ # Add a conditional
        _____________ # Add a print of x
        _____________ # Add breaking condition of x

}
# In this case, ther is no need to add an ''else'' statement, as nothing should happen
to X if we fail the coin_toss
```

8 FUNCTIONS



Functions are a way to summarize multi-step processes into one or few steps. We can view R functions as small programs. All functions or programs have 3 main components:

- **Input:** The information or data we start with
- **Process:** The different steps of cleaning, analyses, mathematical operations, etc., that we want to occur over our data. Loops are processes themselves, so you can put a loop within a function.
- **Output:** The expected result of the process. Can be a new table, summary statistics, analysis results, plots, and more.

The **input** and **output** can be anything, from a single number, to a table, map, or genome. If we want to build our own functions, we need to make sure that we understand what our 3 main components are for any process we want to transform into a function.

R has +16.000 packages available. **A package** (or library in other languages) is a set of functions that are used for specific reasons. Some are developed to perform particular analysis, such as statistical modeling, study survival data, making gene alignments, etc. Others have a broader impact and are used for general analysis purposes, plotting, loading information into R, etc.

Even with that many information, packages, and functions available, there is still a lot to improve, develop and discover. In the future you may encounter an analysis or protocol that you would want to perform, to realize that it has not been fully implemented in any packages. You may also find a useful function that has some imperfections and you would like to improve them. For these reasons and more, learning how to read and write functions is a great tool for career development.

8.1 STRUCTURE OF FUNCTIONS IN R

A function in R requires the three components we stated before.

- **function() statement:** the function() statement contains our functions **arguments**. These are the names of objects or variables that will play a role in the internal process, and that must be provided by the user.
- **Internal process**: within a pair of curly brackets after the function statement "{}", we need to list all the steps and sub-processes the data will go through.
    - **Output**: Just at the end of the internal process, we must make sure to **return** a value or object, that will be the result of our function

8.2 EXAMPLE OF A FUNCTION

8.2.1 sample()

The sample function is part of the base R packages that will be always loaded when you open R or Rstudio. It allows us to make a random sample (with or without replacement) from a given vector. We can use this function to extract random points from a vector or dataset.

If we request the help for this function [run ?sample, or help(sample)], R will show usage, summarizing the arguments we can use in a function; arguments, explaining in a more detailed manner what each argument means for the function; details, which provides more information about the internal process of the function; and in some instances reproducible examples at the end.

You will notice that the function sample has at least 4 arguments:

sample(x, size, replace = FALSE, prob = NULL)

Each one of these terms will be used in some way or another within the function. Whenever we have an argument followed by an equal sign and a value, such as "replace = FALSE", we call them default values of arguments. When we use a function, all the arguments will be used, and the user must provide as input all of those that are not followed by a default value. In this sense, we can modify the default values if needed, but it is optional to do so.

If you want to see the internal process of a function, you need to run the function without parenthesis, or surrounded by the View() function:

sample

View(sample)

There, you will see the different arguments that define the function and how they are used in the internal process. As a disclaimer, not all functions are written in R. This language has some compatibility with others, such as C, C++, Java, and FORTRAN. Whenever you can't see the full code of a function, it may have been written in one of those.

Once you get more use to the gist of coding, you can copy a function and modify it in any way for your personal use. That's one of the advantages of R as an open source software and programming language.

8.3 LET'S WRITE OUR FIRST FUNCTION!

Our task for the moment is to make two numbers multiply themselves using a function. R has its own multiplication operator, an asterisk (*), however, the goal of this exercise is to get you used to the structure of a function in R.

- **Input:** arguments X and Y
- **Process:** Multiply X*Y. Assign the result to a new variable Z
- **Output:** return Z

```
# Create function
# We put a name to the function in a similar way as when we assign values to objects
> Multiplication <- function(X, Y) {  # name, function statements, and arguments
        # Internal process
        Z = X*Y
        return(Z)
}

> Multiplication(X = 3, Y = 8) # Results in 24
> Multiplication(2, 9) # Results in 18
> Multiplication() # Results in Error, as there were no default values
```

Now, we can put some default values to our function. Think of a couple random numbers and add them to your Multiplication function in this way:

```
# Create function
# We put a name to the function in a similar way as when we assign values to objects
> Multiplication <- function(X = 1, Y = 7) {  # name, function statements, and arguments
        # Internal process
        Z = X*Y
        return(Z)
}

> Multiplication() # Results in 7
> Multiplication(X = 3, Y = 8) # Results in 24
> Multiplication(9) # Results in 63
> Multiplication(Y = 9) # Results in 9
```

Think about those results for a minute. What do you think is happening?

Some rules of functions

- Functions respect the order of the arguments.
- You may put the name of the argument you want to use with the value/object you need it to be.
- If you put the input values in order, you may omit the writing of the arguments' names.
- However, whenever you want to change the order of those arguments, or only refer to one of them that's not the first argument, you must write its name and value.
- All other unwritten arguments with default values will be kept as they are in the function definition.

Quick Exercise: Create a function called "power_of", that will raise one number to the power of the second number (X^y). Set default values in a way that, whenever the second number is not provided, the result of power_of(AnyNumber) will be 1.

8.4 EXERCISE 5: BUILD A FUNCTION TO SOLVE THE MAZE!

Using what you have learn until now, build a function around your loop for solving any maze.

- **Input:** maze data frame
- **Process:** Your previous loop
  - **Output:** Can be the print that the loop already does
  - **Output (optional)**: Change the loop to store each solution into a vector. Return that vector.

9 FINAL EXERCISE: MERGING LOOPS AND FUNCTIONS

In this final exercise we will use what you have learned on loops and functions to make maps of occurrences for the species *Python bivittatus* in the U.S. The data was extracted from the GBIF database. This dataset contains information on reported observations of the species in Florida. You must:

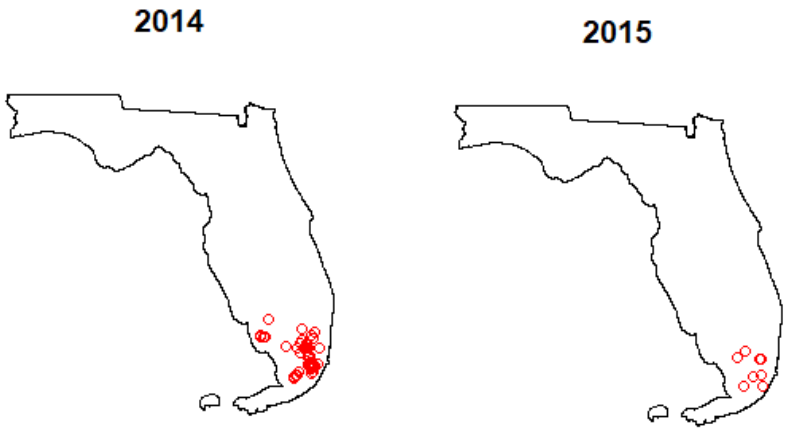
1) Create a loop to make a map of occurrences of *P. bivittatus* in Florida by year, from 1991 to 2020.
2) You will make a function called map_years() around that loop, that should work on any species extracted from GBIF.

- **Input:** Occurrence data frame and Shapefile of Florida
- **Process:** Cleaning, transforming, and plotting the data
- **Output:** The expected result of the process. Can be a new table, summary statistics, analysis results, plots, and more.

The functions you will need for your loop and function internal process are in the next page. You will need to construct a loop and a function over those functions. Remember, for the loop try to identify what should be the iterative term, the vector, and the internal process. For the function, identify what the input arguments should be. The plot and points function work as outputs. **Once your function run, we can try testing it with a new shapefile and/or occurrences records.**

If we have enough time, I will teach you to store the output of images in PNG format and into a folder in your computer. If there is not enough time, look for them in forums such as StackOverflow and R-bloggers, I'm sure you will find the answers.

Examples of simple maps created through a loop from this dataset



Final Exercise functions

```
#### Load packages ####
library(rgdal)
library(sp)
library(tidyverse)

#### Read Data ####
occurrences <- read.delim("Python_bivittatus/occurrence.txt")
Florida <- readOGR("Florida_State_Waters_and_Land_Boundary-shp",
"Florida_State_Waters_and_Land_Boundary")

# Extracts species name, year, and coordinates
occurrences <- occurrences[, c(230, 103, 133, 134)]

head(occurrences)

# Retain full records, without NAs #
occurrences <- occurrences[complete.cases(occurrences), ]

# Frequency table of occurrences per year #
table(occurrences$year)

# Transform into spatial points object with year as attribute #
occurrences <- SpatialPointsDataFrame(coords = occurrences[, 4:3], data =
occurrences[, 1:2])

# Plotting all the data first #
plot(Florida)
points(occurrences)

# Plotting just one year #
plot(Florida, main = 1991)
points(occurrences[occurrences$year == 1991, ], col = “red”)
```