Loops and Functions (Python)

**Instructor:** Alfredo Ascanio

**E-mail:** [ascaniaa@miamioh.edu](mailto:ascaniaa@miamioh.edu)

**Assistant instructors:** Luis M. Montilla, Hank Stevens

**Goals**

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

**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.**

**Recommended Previous Knowledge**

* Data types in **R or Python**: numeric, character, factors, logical/boolean values, etc.
* Base **R or Python** objects and subsetting:
  + Vectors, Matrices, Lists, and Data Frames
* Using functions1
* Conditionals and Logical Operators1

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

**Why loops and functions?**

*“If you need to repeat a process more than twice, program it”*

While doing your work, have you found yourself 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
* Transparency
* Ability to retrace your steps in the process and make corrections or updates
* Shareability
* **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.

Sequence  
i = 1 to N

After  
i = N

Process

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

Process/Step 1

Process/Step 2

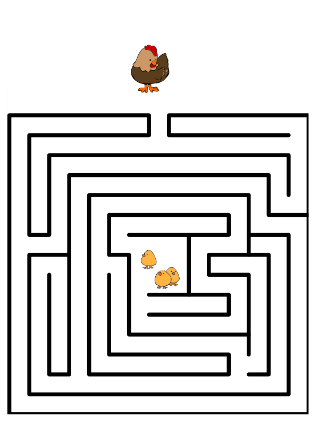
Function

**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.

**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”**
5. **If – \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**
6. **Then – \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**
7. **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.

**How do we write a conditional in Python?**

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.

import numpy as np  
n, p, size = 1, .5, 1 # number of trials, probability of each trial  
coin\_toss = np.random.binomial(n, p, 1)

# 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 results to True --- then process indexed below will run*

1. **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  
import numpy as np  
n, p, size = 1, .5, 1 # number of trials, probability of each trial  
coin\_toss = np.random.binomial(n, p, 1)  
# 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  
n, p, size = 1, \_\_\_, 1 # number of coin tosses, probability of success,  
coin\_toss = np.random.binomial(n, p, 1)  
# Set condition for success, define process to occur if conditions  
# are True or False and corresponding outputs  
if coin\_toss == 1:  
 print("\_\_\_\_")  
else:  
 print("\_\_\_\_")

**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 Boolean value in Python)

if True:  
 print("Execute process 1!")  
else:  
 print("Execute process 2!")

1. If the condition after the *if* statement results to False, then the secondary and optional process will run instead.
2. We can add as many conditions as we needed

if condition\_1 == True:  
 print("Execute process 1!")  
elif condition\_2 == True:  
 print("Execute process 2!")  
else:  
 print("Execute process 3!")

**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 ####*

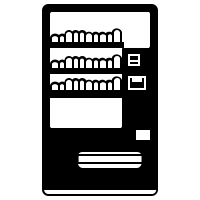
x, y = 5, 8  
print(x > y) # is x greater than y? Results to FALSE  
print(x < y) # is x less than y? Results to TRUE  
print(x >= y) # is x equal to or greater than y? Results to FALSE  
print(x <= y) # is x equal to or less than y? Results to TRUE  
print(x == y) # is x equal to y? Results to FALSE  
print(x != y) # is x different to y? Results to TRUE  
y = [5, 9]  
print(x in y) # is the value of x contained in the values of y? Results FALSE

*#### Functions with logical outputs ####*

x = [False, False, False];  
y = [False, False, True]  
print(x == y) # if two objects are exactly the same returns one TRUE  
  
print(any(x)) # Input is a logical vector, if any value is TRUE, returns one TRUE  
print(any(y))  
  
print(all(y)) # input is a logical vector, if all values are TRUE, returns one TRUE

*#There are many more that you will discover as you start needing them*

**Exercise 2: Vending Machine!**



***Drinks***

***Puzzling***

There is a famous company of vending machines called *Puzzling Drinks*. These vending machines sell 3 drinking products: iced latte, iced cappuccino, and iced mocha. When a client goes to purchase something from the machines, it selects a product at random and gives them a puzzle piece. After they have collected the three puzzle pieces they get a drink for free. The owner of the company wants to expand its range of products so those that are lactose intolerant can buy their drinks. Thus, the machines will now sell 6 drinks (the 3 original, and 3 lactose-free).

I’ll gave you a function that will produce a client, with a given probability of being lactose intolerant. You need to create a conditional statement that checks whether the person is lactose intolerant or not. The process to occur will be to print the person status, and then print a random drink **from the appropriate selection of drinks**.

**Loops: For-Loop**

Sequence  
i = 1 to N

Process

After  
i = N

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

Process

**Next Iteration**

Advance to next   
Element in vector

|  |
| --- |
| **Data (i)** |
| Data 1 |
| Data 2 |
| Data 3 |
| **…** |
| Data N |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Vector** | 1 | 2 | 3 | **…** | N |

**Structure of a for-loop**

for i in range(1, 10):  
 print(i)

*# for is the statement that will initialize the loop*

*# i is the iterative term*

*# range(1, 10) is a list of numbers from 1 to 9*

*# print(i) is the process*

* + for statement
  + Iterative term
  + Vector or sequence
  + Process

**Quick Exercise: Change the input sequence and the name of the iterative term**

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

b) You can make the list a list of characters. See what happens.

**Identifying the proper iterative term**

The proper use of the iterative terms and sequences 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 list (The **list** 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 variable category/factor level (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.

**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)

**For-loop print elements from character vector – Guided**

The python library *string* provides us with a vector of lowercase or uppercase letters (“ascii\_lowercase” and “ascii\_uppercase”, 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

from string import ascii\_lowercase  
char\_vector = ascii\_lowercase # You can change this vector if you want  
# Option 1  
for char in char\_vector:  
 print(char)

*# Option 2 – Using the number of elements and their positions in the LETTERS vector*

for i in range(0, len(char\_vector)):  
 print(char\_vector[i])

**For-loop math on a vector**

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

1. Add 2
2. Print the output

*# Create a vector called math*

math = [5, 8, 10, 13, 14, 17, 21, 24]

With the same math vector, perform the following operation

1. Add 2
2. Divide by 3
3. Raise to the power of the original number
4. Print output

**For-loop by factor on a biological dataset**

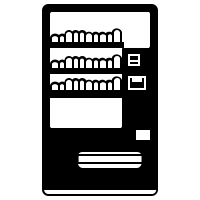
The python library *pandas* provides us with a way to read csv files. We will read the dataset called “iris\_csv”, which contains information on the morphology of three species of flowers: *Iris setosa, I. virginica,* and *I. versicolor*. The dataset contains observations on 150 flowers, 50 per species. Create a loop to extract the mean values of Sepal and Petal length for each flower

*# Assign Iris data to “Data” object*

import pandas as pd

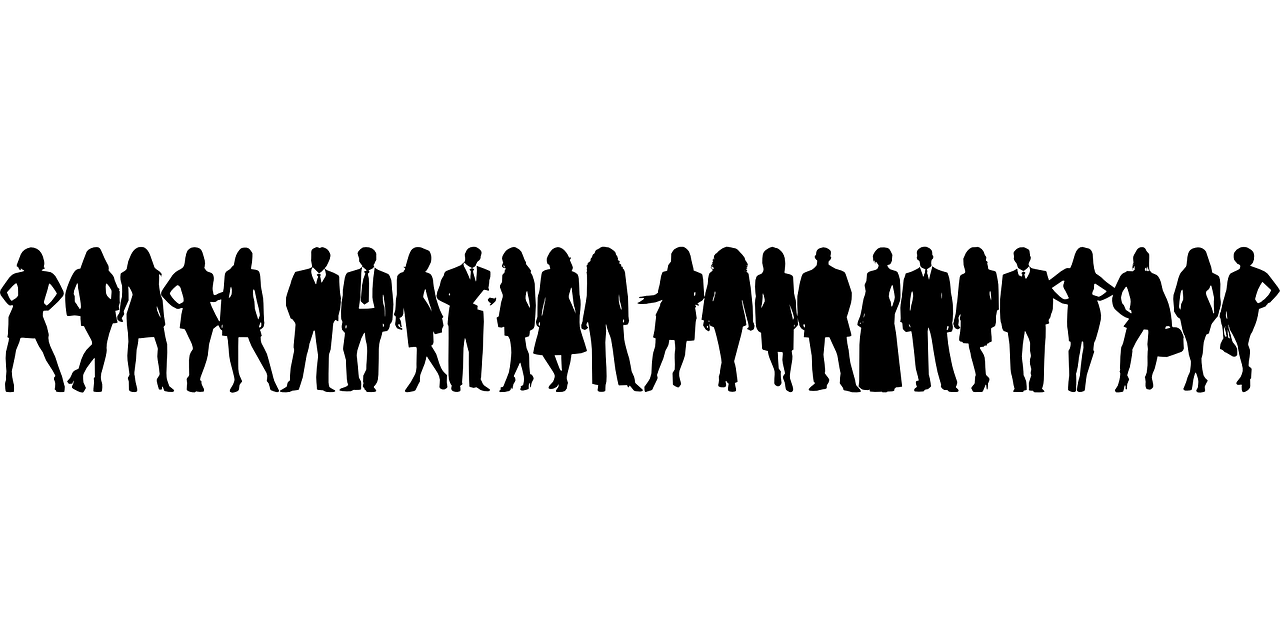
Data = pd.read\_csv("iris\_csv.csv")

**Exercise 3: Let's sell drinks to multiple clients!**



***Drinks***

***Puzzling***



Is the Vending Machine you fixed ready to attend more than one person in line to buy puzzling drinks? Now you have 100 clients in line trying to buy drinks in a vending machine. Let’s create a for-loop. For each people in line, the for-loop should print their status followed by the drink that was served to them.

**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 sequence 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 an empty data frame*

full\_data = pd.DataFrame()

*# Read csv file*

Data1 = pd.read\_csv("Splitted\_Tmanatus/Diveboard - Scuba diving citizen science.csv")  
Data2 = pd.read\_csv("Splitted\_Tmanatus/Fort Fisher.csv")

*# Join data frames by row*

**> full\_data <- rbind(full\_data, Data1, Data2)**

# you can append NULL objects with others, may be useful for your loop  
full\_data = full\_data.append(Data1, ignore\_index = True)  
  
# Joining two datasets by row (if they have the exact same columns' structure)  
full\_data = pd.concat([Data1, Data2])

*# List all names of files in a folder*

from os import listdir  
# listdir() allows you to list all the files in a folder  
files = listdir("Splitted\_Tmanatus")

**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:

from os import listdir  
# listdir() allows you to list all the files in a folder  
files = listdir("Splitted\_Tmanatus")  
  
# You can paste the folder name to your files  
for i in range(0, len(files)):  
 files[i] = "Splitted\_Tmanatus\\" + files[i]  
files  
  
#### Create your Loop ####  
full\_data = pd.DataFrame()  
for i in range(0, len(files)):  
 Data = pd.read\_csv(files[i]

Data = Data[Data["Year"] > 1970]  
 full\_data = full\_data.append(Data, ignore\_index = True)

**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.

**While-loops**

Process

**True**

Condition

**False**

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 add 1 to it.

# Create X variable  
x = 1  
# While-loop  
while x < 10: # Condition within (), instead of sequence  
 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.

# Create X variable  
x = 1  
# while-loop  
while x < 10:  
 # Add coin\_toss  
 if \_\_\_\_\_: # Add a conditional over the result of coin toss  
 \_\_\_\_\_# In this case, there is no need to add an "else" statement, as nothing  
 \_\_\_\_\_# should happen to x if we fail the coin\_toss  
 \_\_\_\_\_# Add a print of x

*# In this case, there is no need to add an “else” statement, as nothing should happen to X if we fail the coin\_toss*

**Repeat-loops**

Condition

**False**

**True**

Process

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 in python can be created as a modification of while-loop  
while True:  
 x = x + 1  
 print(x)  
 if x >= 10:  
 break

*# We use the statement 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  
n, p, size = 1, .5, 1 # number of trials, probability of each trial  
coin\_toss = np.random.binomial(n, p, 1)  
# Create X variable  
x = 1  
# while-loop  
while True:  
 coin\_toss = np.random.binomial(n, p, 1) # Add coin\_toss  
 if coin\_toss == 1: # Add a conditional over the result of coin toss  
 x = x + 1  
 \_\_\_\_ # In this case, there is no need to add an "else" statement, as nothing  
 \_\_\_\_ # should happen to x if we fail the coin\_toss  
 \_\_\_\_ # Add a print of x  
 \_\_\_\_ # Add breaking condition

*# In this case, there is no need to add an “else” statement, as nothing should happen to X if we fail the coin\_toss*

**Functions**

Process/Step 1

Process/Step 2

Function

Functions are a way to summarize multi-step processes into one or few steps. We can view Python 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.

Python has +137.000 libraries available. **A library** (or **package** in R) 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 python, etc.

Even with that many information, libraries, 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 libraries. 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.

**Structure of Functions in Python**

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

* + ***def* statement:** the *def* statement contains our function **name** and **arguments**. These arguments are the names of objects or variables that will play a role in the internal process, and that must be provided by the user. This line must end with a colon “**:**”
  + **Internal process**: 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

**Example of a function**

**sample()**

The sample function is part of the *random* Python library. It allows us to make a random sample (without replacement) from a given list of elements. We can use this function to extract random points from a vector or dataset.

If we request the help for this function [run help(sample)], Python will print in console information explaining its usage, summarizing what the arguments refer to, and expected output.

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

**sample(population, k, \*, counts = None)**

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 “counts = None”, 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 import the library *inspect* and use the function getsource()

**import inspect**

**inspect.getsource(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 need to be written in Python. 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 used 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 python as an open source software and programming language.

**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*

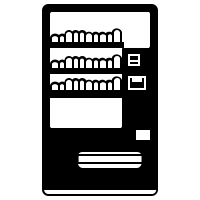
**Think about those results for a minute. What do you think is happening?**

**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’s definition.

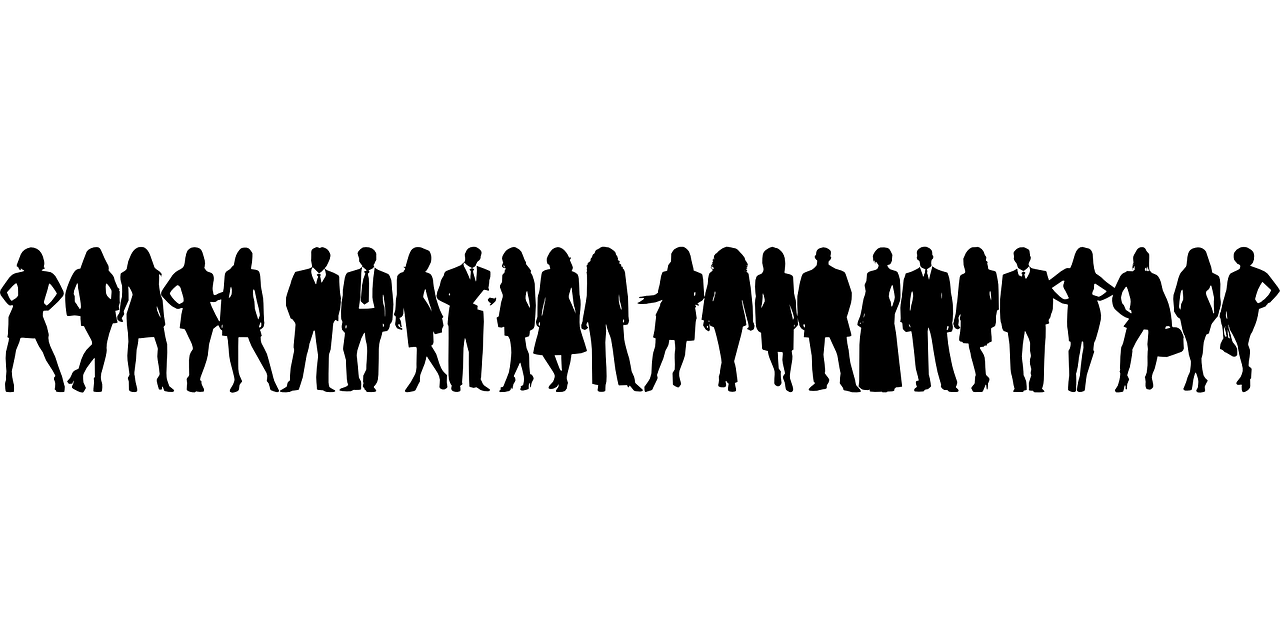
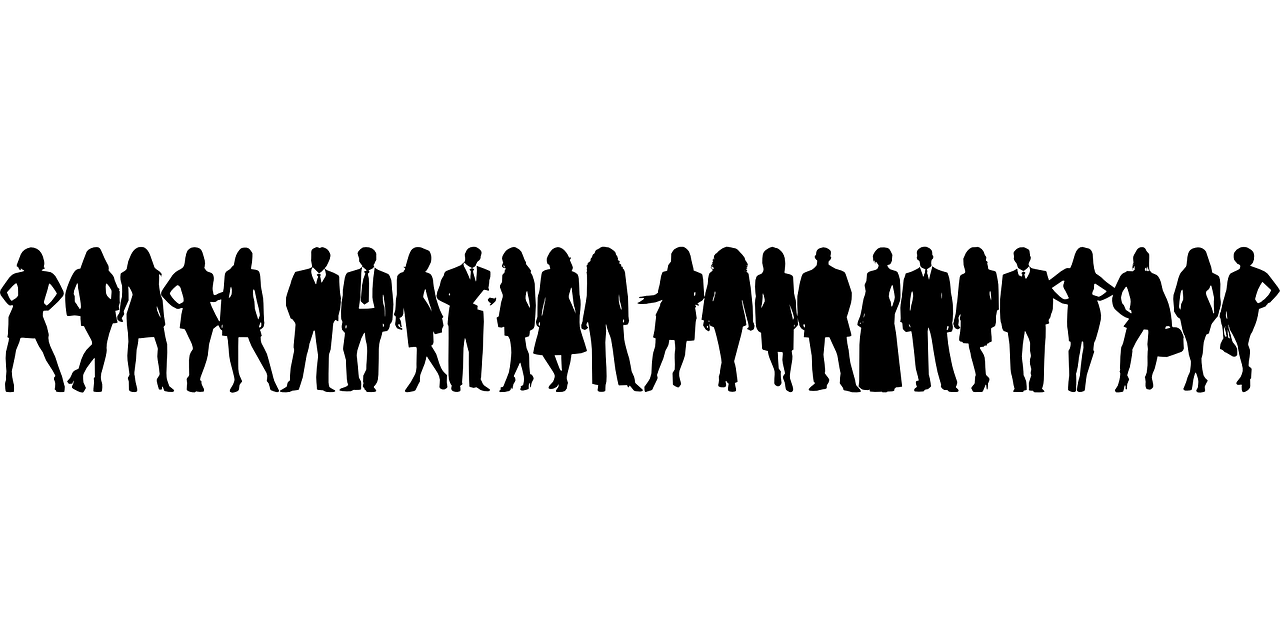
**Quick Exercise:** Create a function called “power\_of”, that will raise one number to the power of the second number (**Xy**). Set default values in a way that, whenever the second number is not provided, the result of power\_of(AnyNumber) will be 1.

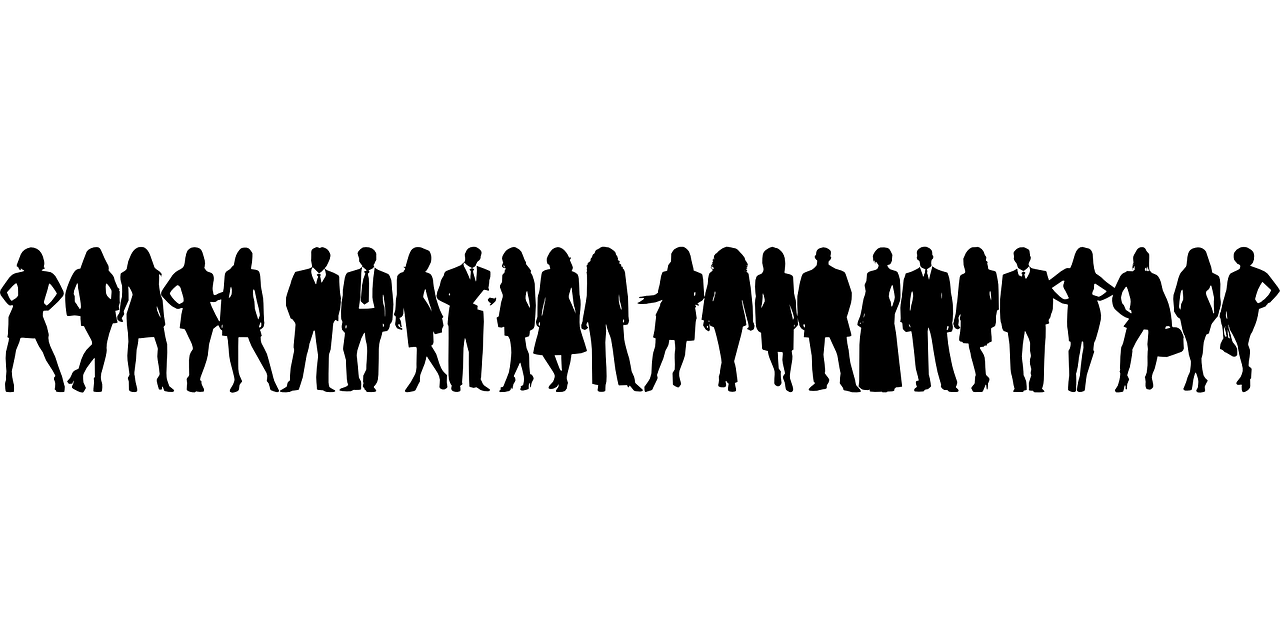
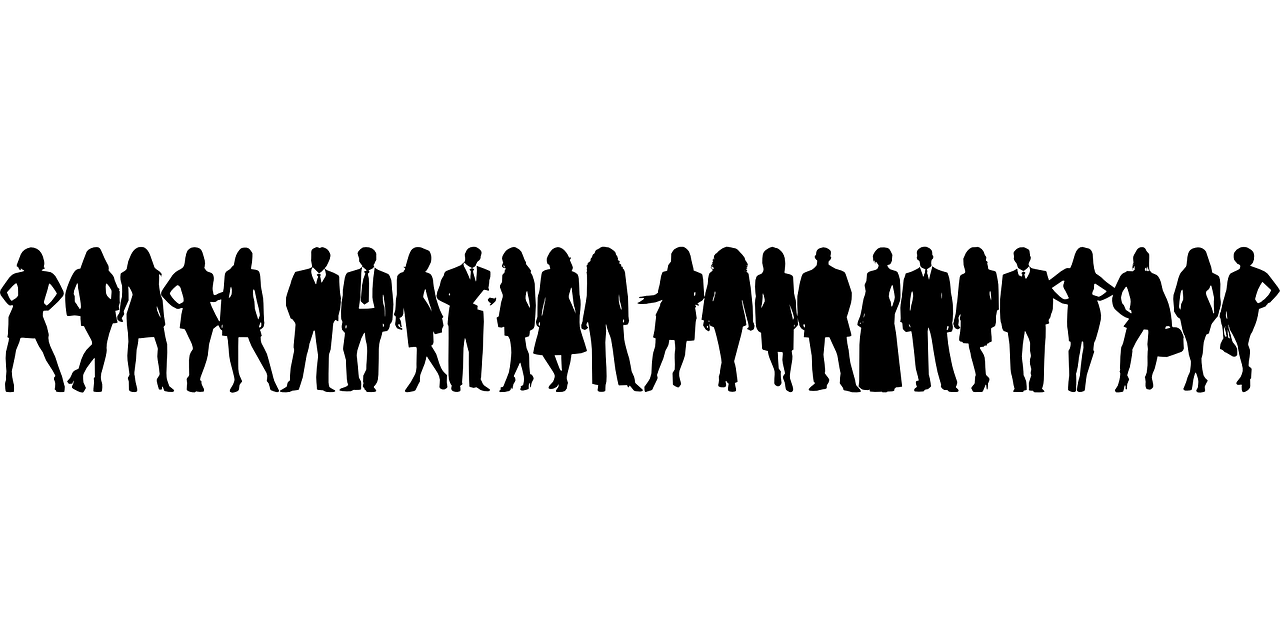
**Exercise 5: Let's make a function that sells drinks to any number of clients!**



***Drinks***

***Puzzling***





Is the Vending Machine you fixed ready to sell drinks to more than one specific size of group in line? Now you have 1000 clients in line trying to buy drinks in a vending machine. Let’s create a function which argument is any line of people from the function *clients­\_line()*. Your function should contain your previous for-loop and conditional. For each people in line, the function should print their status followed by the drink that was served to them.