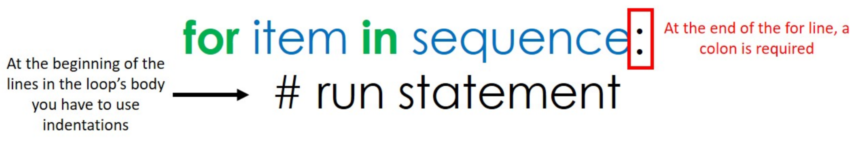
6 Iterative Loops

Loops are essential in almost every programming language. Loops are among the most basic and powerful programming concepts. A loop in a computer program is an instruction that repeats until a specified condition is reached. In a loop structure, the loop asks a question. If the answer requires an action, it is executed. The same question is asked again and again until no further action is required. Each time the question is asked is called an iteration. Just about every programming language includes the concept of a loop. One of the main types of loops is the **for** loop.

6.1 for Loop

for Loop repeats a block of code a set number of times. The reason they are called **for** loops is that you can tell your program how many times you want it to repeat the code. **for** loops help you reduce repetition in your code because they let you execute the same operation multiple times. **for** loops use a variable to count how many times the code has been repeated, called a counter. You control how often the loop repeats by setting where the counter starts and ends. You also set how much the counter goes up each time the code repeats. In most scenarios, you'll want the counter to increase by 1 each time the loop repeats. The syntax for **for** loop can be shown as following:

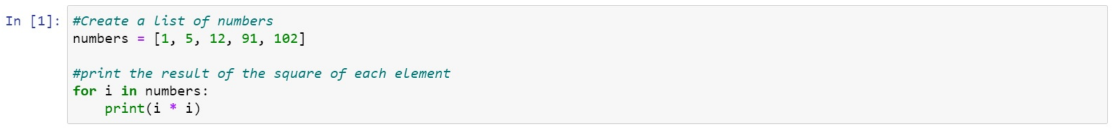


```
for item in sequence:
    # run statement
```

At the end of the for line, a colon is required
At the beginning of the lines in the loop's body you have to use indentations

Here, the `sequence` can be either a list of strings or a range of numbers. The `item`, on the other hand, is the variable that iterates over the elements of the list, and every time it is iterated, the line (or multiple lines) of code is executed. It is very important here to mention that you need to be careful with the syntax. At the end of the **for** line, a colon is required. Also, Python relies on indentation (whitespace at the beginning of a line) to define the scope of the code. This means that lines that contain expressions to execute must be intended.

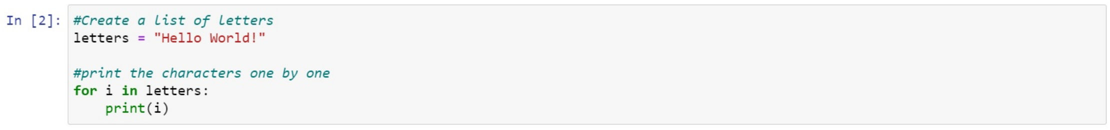
For example, Fig(29) shows how to compute the squared values of each number in a given list using **for** loop. "**for**" and "**in**" are Python keywords and `numbers` is the name of our list. Also, "`i`" is a temporary variable and its only role is to store the given element of the list that we will work within the given iteration of the loop. Even if this variable is called `i` most of the time (in online tutorials or books for instance), it's good to know that the naming is totally arbitrary. The function inside the loop asks the loop to print the result of the squared value each time an iteration is achieved.



```
In [1]: #Create a list of numbers
        numbers = [1, 5, 12, 91, 102]

        #print the result of the square of each element
        for i in numbers:
            print(i * i)
```

Figure 29: **for** loop over a list of numbers.

As mentioned before, you can use other sequences than lists too. For example, a string, as shown in Fig(30). Remember, strings are basically handled as sequences of characters, thus the **for** loop will work with them pretty much as it did with lists.



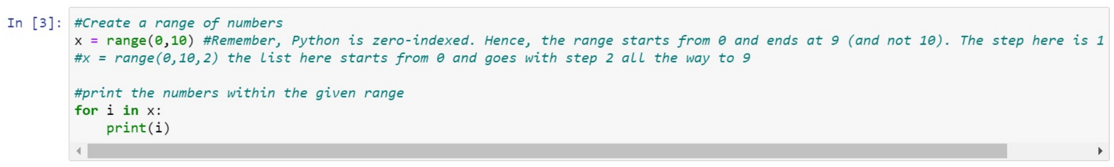
```
In [2]: #Create a list of letters
        letters = "Hello World!"

        #print the characters one by one
        for i in letters:
            print(i)
```

Figure 30: **for** loop over a list of characters.

An important function that should be mentioned in this context is the `range()` function. `range()` is a built-in function in Python and is used almost exclusively within **for** loops. It mainly generates a list of numbers. As shown in Fig(31), it accepts three arguments:

- **The first element:** This will be the first element of your range.
- **The last element:** You might assume that this will be the last element of your range... but it isn't. Again, as mentioned previously, Python is zero-indexed, which means that it starts counting from zero. Therefore, if the last element is 10, as in the example shown, then the range will go from 0 to 9.
- **The step:** This is the difference between each element in the range. So if it's 2, you will only print every second element.

Note: the first element and the step attributes are optional. If you don't specify them, then the first element will be 0 and the step will be 1 by default.



```
In [3]: #Create a range of numbers
        x = range(0,10) #Remember, Python is zero-indexed. Hence, the range starts from 0 and ends at 9 (and not 10). The step here is 1
        #x = range(0,10,2) the list here starts from 0 and goes with step 2 all the way to 9

        #print the numbers within the given range
        for i in x:
            print(i)
```

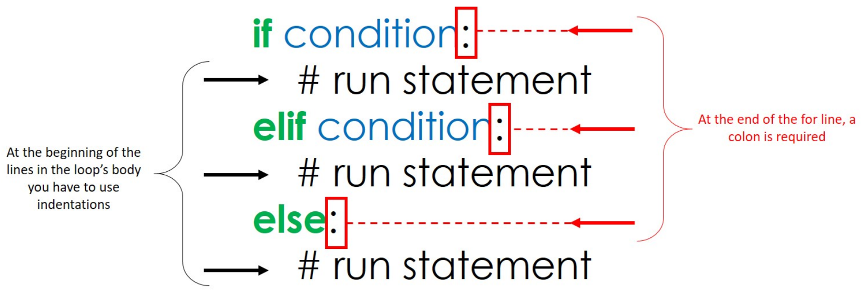
Figure 31: **for** loop over a range of numbers.

7 Conditional Statements

Conditionals (that is, conditional statements, conditional expressions, and conditional constructs) are programming language commands for handling decisions. Specifically, conditionals perform different computations or actions depending on whether a programmer-defined boolean condition evaluates to true or false. **if** statement is a key part of the decision-making process in programming.

7.1 if Statement

The **if** statement is one of the powerful conditional statements and is common across many programming languages. It is responsible for modifying the flow of execution of a program. It is always used with a condition. The condition is evaluated first to either true or false before executing any statement inside the body of **if**. The syntax for **if** statement can be broadly constructed as follows:

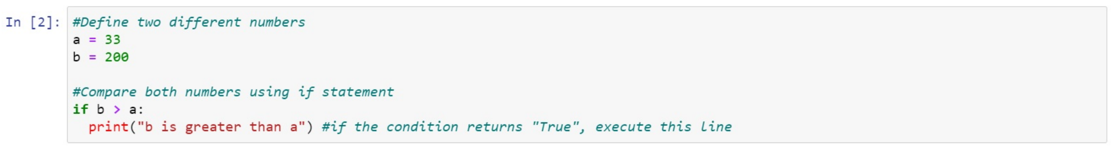


if condition:
# run statement
elif condition:
# run statement
else:
# run statement
At the beginning of the lines in the loop's body you have to use indentations
At the end of the for line, a colon is required

Similar to **for** loop, the **if** statement requires a colon at the end of the **if**, **elif** and **else** first lines. Moreover, as mentioned before, Python relies on indentation to define the scope in the code, and that's also applied to **if** statement structure. If no indentation is applied, you will get an error.

It is noticed from the structure that there are three keywords that are used in the **if** statement: **if**, **elif** and **else**. Both, **if** and **elif** are followed by a condition to be evaluated. These conditions are mainly formulated using one of the operators that Python supports, listed in Table(1). Some codes do not require for **elif** and else to be in the structure of **if** statement. Sometimes, **if** is enough to achieve the required task. For example, Fig(32) shows that we can evaluate the comparison between `a` and `b` just by using the **if** keyword. In this example, we use two variables, `a` and `b`, which are used as part of the **if** statement to test whether `b` is greater than `a`. As `a` is 33, and `b` is 200, we know that 200 is greater than 33, and so we print to screen that "`b is greater than a`".

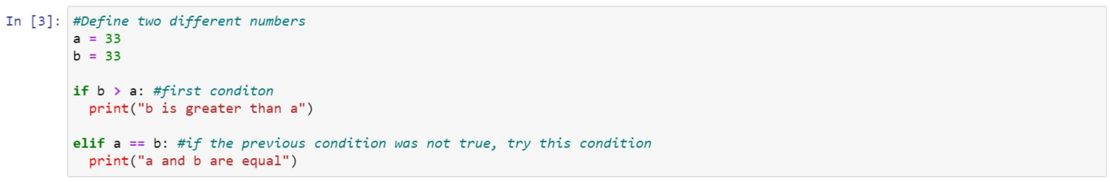
The **elif** keyword is Python's way of saying "if the previous conditions were not true, then try this condition". As shown in the example in Fig(33), `a` is equal



```
In [2]: #Define two different numbers
        a = 33
        b = 200

        #Compare both numbers using if statement
        if b > a:
          print("b is greater than a") #if the condition returns "True", execute this line
```

Figure 32: The use of the **if** keyword in **if** statement.

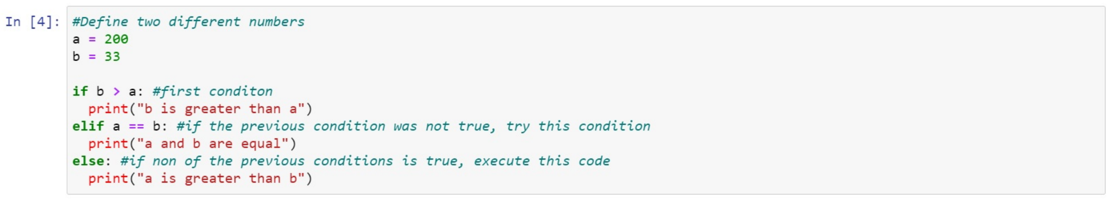
to `b`, so the first condition is not true, but the **elif** condition is true, so we print to screen that “`a and b are equal`”.



```
In [3]: #Define two different numbers
        a = 33
        b = 33

        if b > a: #first conditon
          print("b is greater than a")

        elif a == b: #if the previous condition was not true, try this condition
          print("a and b are equal")
```

Figure 33: The use of the **if** and **elif** keywords in **if** statement.

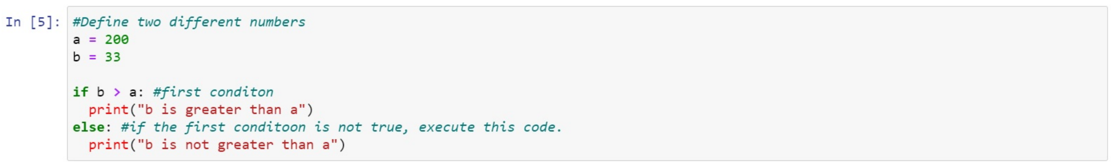
The **else** keyword, on the other hand, catches anything that isn’t caught by the preceding conditions. In the example in Fig(34), `a` is greater than `b`, so the first condition is not true, also the **elif** condition is not true, so we go to the **else** condition and print to the screen that “`a is greater than b`”.



```
In [4]: #Define two different numbers
        a = 200
        b = 33

        if b > a: #first conditon
          print("b is greater than a")
        elif a == b: #if the previous condition was not true, try this condition
          print("a and b are equal")
        else: #if non of the previous conditions is true, execute this code
          print("a is greater than b")
```

Figure 34: The use of the **if**, **elif** and **else** keywords in **if** statement.

You can also have an **else** without the **elif** within the **if** structure, as shown in the example in Fig(35).



```
In [5]: #Define two different numbers
        a = 200
        b = 33

        if b > a: #first conditon
          print("b is greater than a")
        else: #if the first conditoon is not true, execute this code.
          print("b is not greater than a")
```

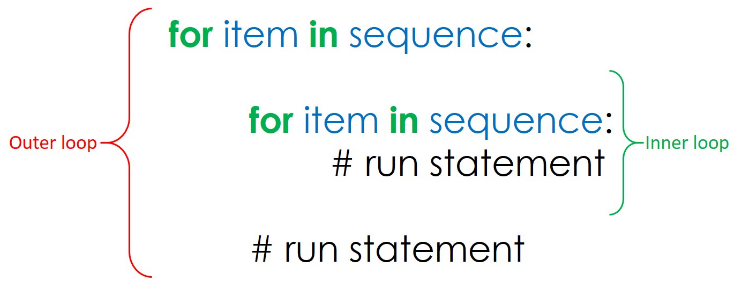
Figure 35: The use of the **if** and **else** keywords in **if** statement.

8 Nesting

Nesting occurs when one programming construct is included within another. Nesting allows for powerful, yet simple programming. It reduces the amount of code needed while making it simple for a programmer to debug and edit. In the following, we will see how loops and conditions can be nested illustrated by examples.

8.1 Nested for

Nesting can applied on iterations, i.e., multiple layers of loops one within another. The main structure of such a nest can be viewed below:



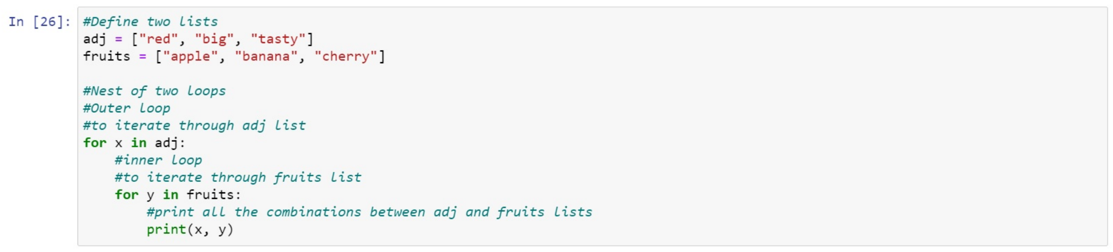
It can be seen that, in the case of two layers of iterations, there are two nested loops to go through: the inner and the outer loops. For each iteration of an outer loop, the inner loop executes all its iterations before the outer loop can continue to its next iteration.

We should note here that the outer loop can contain more than one inner loop. There is no limitation on the chaining of loops.

In the nested loop, the number of iterations will be equal to the number of iterations in the outer loop multiplied by the iterations in the inner loop.

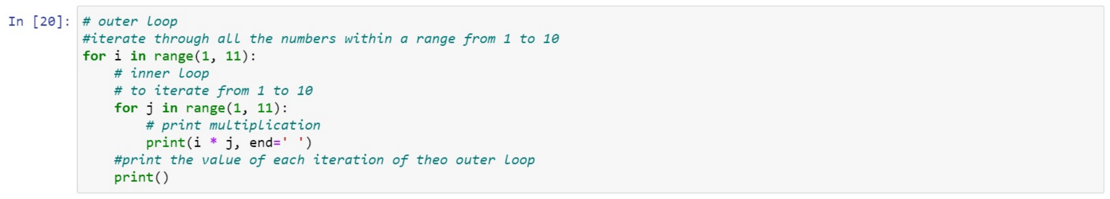
As shown in Fig(36), nested loops can be used to go over string lists. The outer **for** loop goes through each element of “`adj`” list and for each iteration, the code asks the inner **for** loop to go over each element of “`fruit`” list and prints the combinations that result from each iteration.

Nested **for** loops can be also used over a range of numbers, as shown in Fig(37). It has been mentioned before that the `range()` command in Python goes all the way through the specified range minus the last number. The outer **for** loop shown in the code iterates from 1 to 10 (not 11!) and asks the inner loop to iterate through another range from 1 to 10 and prints the result of the product of each of their elements in each iteration.



```
In [26]: #Define two lists
         adj = ["red", "big", "tasty"]
         fruits = ["apple", "banana", "cherry"]

         #Nest of two loops
         #Outer loop
         #to iterate through adj list
         for x in adj:
             #inner loop
             #to iterate through fruits list
             for y in fruits:
                 #print all the combinations between adj and fruits lists
                 print(x, y)
```

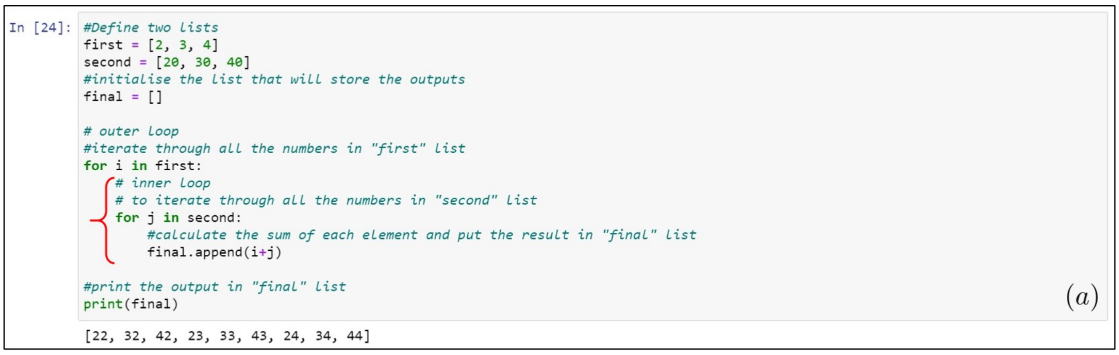
Figure 36: Two nested **for** loops over string lists.



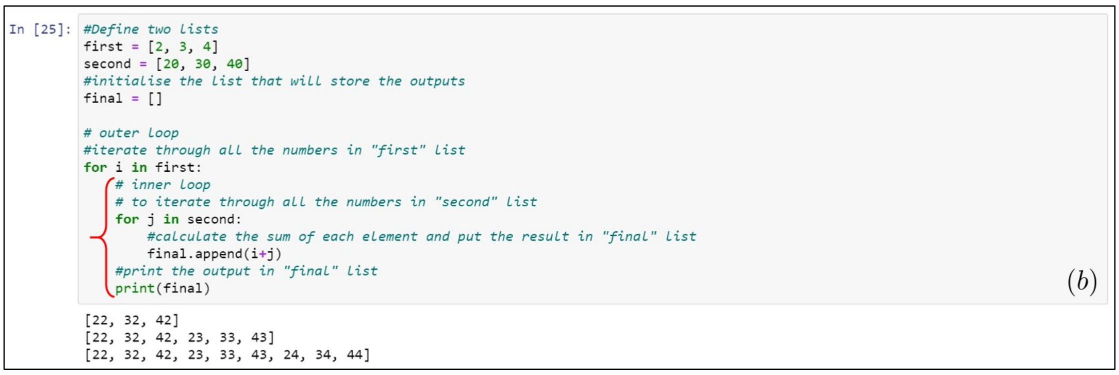
```
In [20]: # outer loop
         #iterate through all the numbers within a range from 1 to 10
         for i in range(1, 11):
             # inner loop
             # to iterate from 1 to 10
             for j in range(1, 11):
                 # print multiplication
                 print(i * j, end=' ')
             #print the value of each iteration of theo outer loop
             print()
```

Figure 37: Two nested **for** loops over two ranges of numbers.

The code shown in Fig(38) illustrates how the indentation affects the shape of the output. It shows different outputs for different nested **for** loop structures. For example in (a), the command `print()` resides outside the nested loops, therefore the result only shows the final output. However, in (b), and since the `print()` command sits inside the body of the inner loop, it shows the result of each of its iterations.



```
In [24]: #Define two lists
         first = [2, 3, 4]
         second = [20, 30, 40]
         #initialise the list that will store the outputs
         final = []

         # outer loop
         #iterate through all the numbers in "first" list
         for i in first:
             # inner loop
             # to iterate through all the numbers in "second" list
             for j in second:
                 #calculate the sum of each element and put the result in "final" list
                 final.append(i+j)

         #print the output in "final" list
         print(final)

         [22, 32, 42, 23, 33, 43, 24, 34, 44]
```



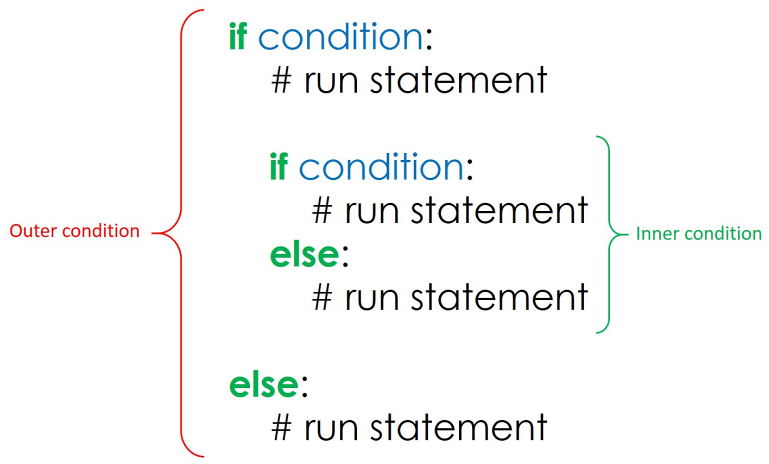
```
In [25]: #Define two lists
         first = [2, 3, 4]
         second = [20, 30, 40]
         #initialise the list that will store the outputs
         final = []

         # outer loop
         #iterate through all the numbers in "first" list
         for i in first:
             # inner loop
             # to iterate through all the numbers in "second" list
             for j in second:
                 #calculate the sum of each element and put the result in "final" list
                 final.append(i+j)
             #print the output in "final" list
             print(final)

         [22, 32, 42]
         [22, 32, 42, 23, 33, 43]
         [22, 32, 42, 23, 33, 43, 24, 34, 44]
```

Figure 38: Two nested **for** loops over to lists of numbers and stores the output in a different list.

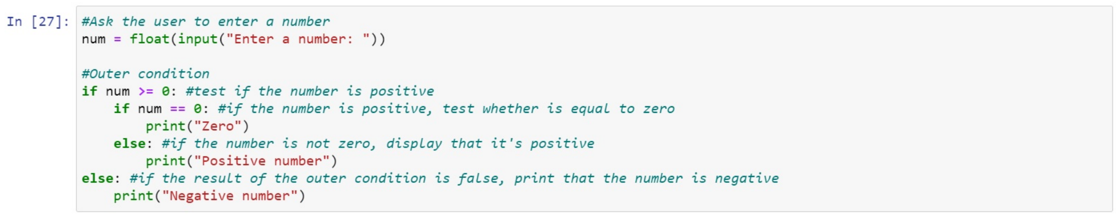
8.2 Nested if

Similar to iterations, conditions can also be nested, i.e., you can have **if** statements inside **if** statements. There may be a situation when you want to check for another condition after a condition resolves to be true. Any number of these conditions can be nested inside one another. Indentation is the only way to figure out the level of nesting. The general structure of nested conditions is shown below.

The code shown in Fig(39), asks the user to input a number (positive or negative). The outer **if** condition tests whether the number is positive, otherwise the **else** command prints on the screen that the number is negative. The inner condition, however, provided the result of the outer condition is true, tests whether the number is zero.



```
if condition:
    # run statement

    if condition:
        # run statement
    else:
        # run statement

else:
    # run statement
```



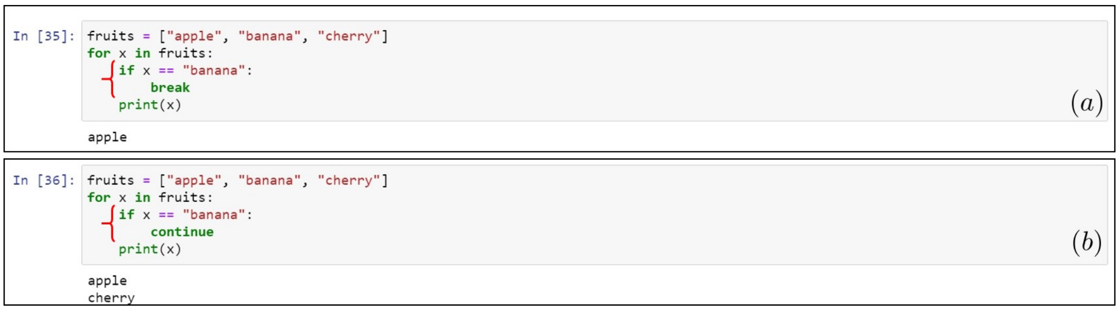
```
In [27]: #Ask the user to enter a number
         num = float(input("Enter a number: "))

         #Outer condition
         if num >= 0: #test if the number is positive
             if num == 0: #if the number is positive, test whether is equal to zero
                 print("Zero")
             else: #if the number is not zero, display that it's positive
                 print("Positive number")
         else: #if the result of the outer condition is false, print that the number is negative
             print("Negative number")
```

Figure 39: Two nested **if** conditions that test whether a given number is positive or negative.

8.3 Nested for and if

We can also have a hybrid combination of nested **for** loops and **if** conditions in the same structure. Again: when you use an **if** statement within a **for** loop, be extremely careful with the indentations because if you misplace them, you can get errors or incorrect results!

It's also worth mentioning here two common commands when using hybrid nesting: `break` and `continue`. In Fig(40), we have two programs, in (a), the code goes through all the elements of the "`fruits`" list via for loop. In the inner **if** condition, it asks if the element is equal to "`banana`". If it's true, the code must `break`. Otherwise, it should print the result of the iteration. With the `break` statement, we stop the loop before it has looped through all the items. That's why the answer for this chunk of code is only `apple`. On the other hand, in (b), the code goes through the same "`fruits`" list via the for loop, but here when the **if** statement asks whether the iterative variable equals to "`banana`", it demands the code to `continue`. With the `continue` statement, we stop the current iteration of the loop and continue with the next. That's why the result of this code is `apple` and `cherry`.



```
In [35]: fruits = ["apple", "banana", "cherry"]
         for x in fruits:
             if x == "banana":
                 break
             print(x)

         apple
```

(a)

```
In [36]: fruits = ["apple", "banana", "cherry"]
         for x in fruits:
             if x == "banana":
                 continue
             print(x)

         apple
         cherry
```

(b)

Figure 40: The use of `break` and `continue` in hybrid nesting.

8.4 Problem: Bisection Method

To make a practical example of the use of **for** and **if** statements, we consider the bisection method for finding zeros of a function [7]. In particular, we will consider the function:

$$f(x) = x^3 + 1 = 0 \tag{6}$$

for which the values of x which make this true must be found computationally. Fig(41) shows the function with the intersection point that represents its root (the red circle) over the interval $x \in [-4, 4]$ where $x_l = -4$ and $x_r = 4$.

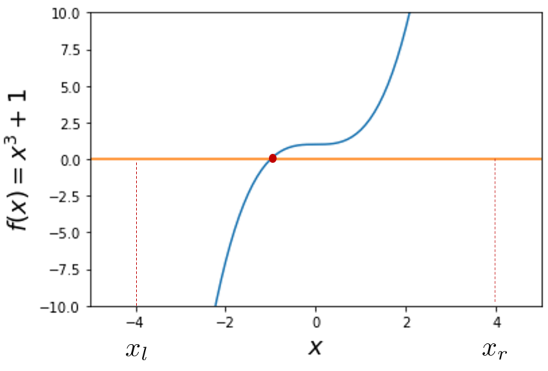
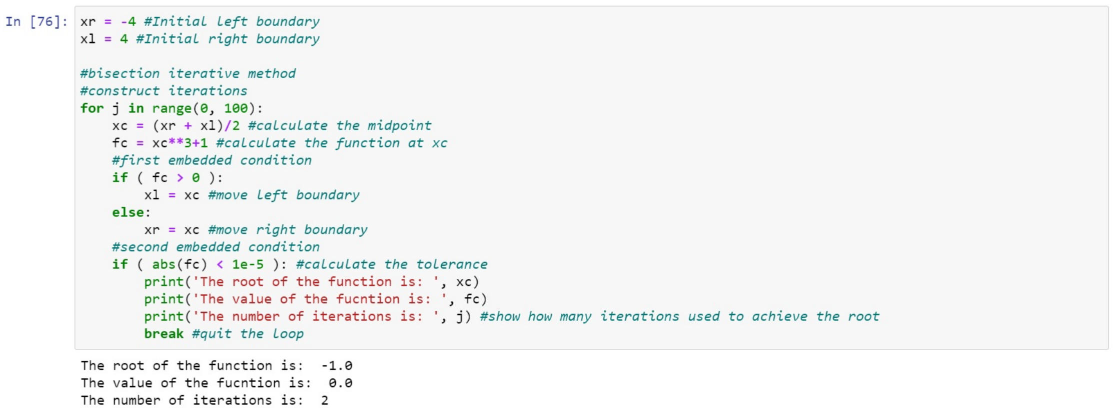


Figure 41: The function $f(x) = x^3 + 1$ with the corresponding zero (the red circle) over the interval $x \in [-4, 4]$ where $x_l = -4$ and $x_r = 4$.

The bisection method simply cuts the given interval in half and determines if the root is on the left or right side of the cut. Once this is established, a new interval is chosen with the midpoint now becoming the left or right end of the new domain, depending of course on the location of the root. This method is repeated until the interval has become so small and the function considered has come to within some tolerance of zero. The following algorithm uses an outside **for** loop to continue cutting the interval in half while the embedded **if** statement determines the new interval of interest. A second **if** statement is used to ensure that once a certain tolerance has been achieved, i.e., the absolute value of the function $f(x) = x^3 + 1$ is less than 10^{-5}, then the iteration process is stopped.



```
In [76]: xr = -4 #Initial left boundary
         xl = 4 #Initial right boundary

         #bisection iterative method
         #construct iterations
         for j in range(0, 100):
             xc = (xr + xl)/2 #calculate the midpoint
             fc = xc**3+1 #calculate the function at xc
             #first embedded condition
             if ( fc > 0 ):
                 xl = xc #move left boundary
             else:
                 xr = xc #move right boundary
             #second embedded condition
             if ( abs(fc) < 1e-5 ): #calculate the tolerance
                 print('The root of the function is: ', xc)
                 print('The value of the fucntion is: ', fc)
                 print('The number of iterations is: ', j) #show how many iterations used to achieve the root
                 break #quit the loop

         The root of the function is:  -1.0
         The value of the fucntion is:  0.0
         The number of iterations is:  2
```

Figure 42: The Bisection Algorithm to fins the root of the function $f(x) = x^3 + 1$.

Note that the `break` command ejects you from the current loop. This effectively stops the iteration procedure for cutting the intervals in half. It can be seen from Fig(42), that only two iterations were necessary to reach the value of the root of the function $f(x)$. Surely, the number of iterations mainly depends on the initialisation of the assumed values of the left and right boundaries.

9 Functions

Functions are a set of instructions bundled together to achieve a specific outcome. Once a function is written, it can be called and reused multiple times within the code. This makes Functions a good alternative to having repeating blocks of code in a program which increases the re-usability of code.

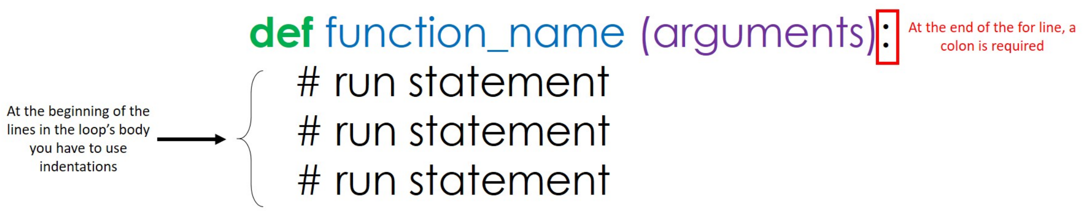
The main reason why we write functions is because they allow us to understand the program as sub-steps. Each sub-step can have its own function. When any program seems too hard, just break the overall program into sub-steps.

A common thing about functions is that they are famous with several names. Different programming languages name them differently, for example, functions, methods, sub-routines, procedures, etc.

The steps to writing a clear and easy-to-understand function are:

- Understand the purpose of the function.
- Define the parameters that the function needs to accomplish its purpose.
- Decide on the set of steps that the program will use to accomplish this goal.
- Code, test, and optimize; go back to any previous step as necessary.

In Python, functions have certain syntax as shown below.

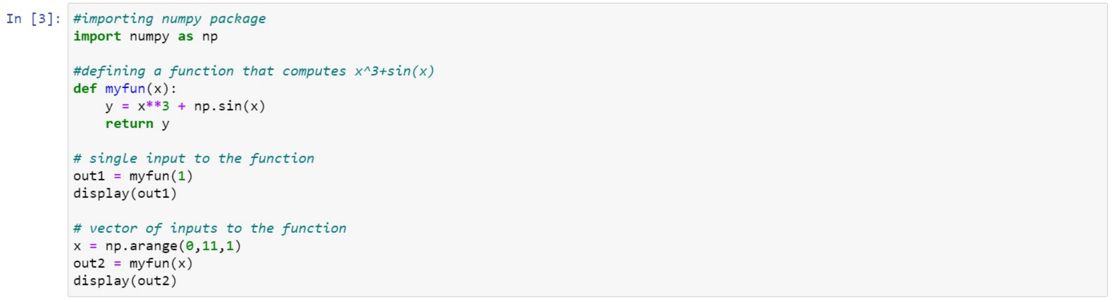


```
def function_name (arguments):
    # run statement
    # run statement
    # run statement
```

At the end of the for line, a colon is required
At the beginning of the lines in the loop's body you have to use indentations

First, the keyword **def** marks the start of the function header and is used to define and create a function. A `function name` follows to uniquely identify the function. This name is essential when calling the function later in the code. Next, `parameters` (arguments) need to be defined within parentheses through which we pass values to a function. Remember, just like in loops and conditional statements, to end this line with a colon. The body of the function contains all the `statements` (instructions) that describe what the function does. Statements must have the same indentation level.

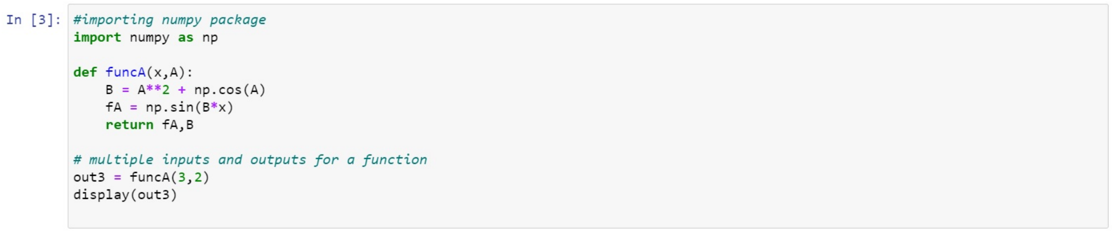
Let's clarify this with an example. Fig(43) shows a code that defines a function `myfun` which computes $y = x^3 + \sin x$. The first line of the function has three main components: the keyword **def**, its name `myfun`, and `x` as an argument. The body of the function has two statements to run, the first is the mathematical expression we need to compute, i.e., $y = x^3 + \sin x$, and the second is to let the function return the value of y. Once we have defined a function, we can simply call it by typing the function name with the appropriate parameters. The code shows two ways to pass arguments through this function; either you pass only one value as in `out1`, or you define a range of values and ask the function to compute the expression of y at each one of these values, as in `out2`. In Python, the function definition should always be present before the function call. Otherwise, we will get an error.



```
In [3]: #importing numpy package
        import numpy as np

        #defining a function that computes x^3+sin(x)
        def myfun(x):
            y = x**3 + np.sin(x)
            return y

        # single input to the function
        out1 = myfun(1)
        display(out1)

        # vector of inputs to the function
        x = np.arange(0,11,1)
        out2 = myfun(x)
        display(out2)
```

Figure 43: A function that computes $y = x^3 + \sin x$.

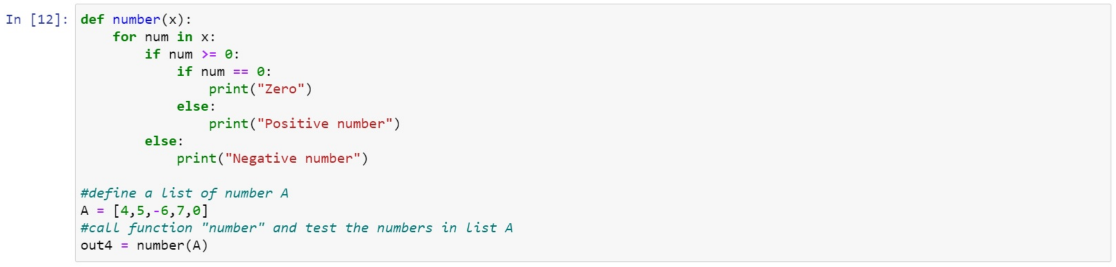
Another example is shown in Fig(44). This code illustrates a function `funcA` that computes $f_A = \sin(Bx)$ where $B = A^2 + \cos(A)$ by passing more than one argument: two arguments to be specific; x and A. It is shown from the body of the function that two statements should be defined to achieve this computation. The first statement calculates B. The second statement uses the value of B obtained before to compute the expression of f_A. The code ends by returning the values of both f_A and B. Once the structure of the function is finalised, it can be used and called later on in the code to compute the mathematical expressions for given arguments. In this code, it is shown that the function is called for the arguments $x = 3$ and $A = 2$.



```
In [3]: #importing numpy package
        import numpy as np

        def funcA(x,A):
            B = A**2 + np.cos(A)
            fA = np.sin(B*x)
            return fA,B

        # multiple inputs and outputs for a function
        out3 = funcA(3,2)
        display(out3)
```

Figure 44: A function that computes $f_A = \sin(Bx)$ where $B = A^2 + \cos(A)$.

Loops and conditions can be nested within functions as statements. The structure of both remains the same as explained before, but always pay attention to the indentation. Fig(45) shows a function `number` that goes through all the numbers of a given list `x` and tests whether each number is a positive, negative, or zero. To achieve this task, the argument of the function `x` should be a list of numbers. Next, a loop over all the elements of a given list `x` should be used and three conditions need to be implemented to test the three options on each element. Once the function is constructed, we can call it to apply it on list `A` and print out the type of all of its elements.



```
In [12]: def number(x):
             for num in x:
                 if num >= 0:
                     if num == 0:
                         print("Zero")
                     else:
                         print("Positive number")
                 else:
                     print("Negative number")

         #define a list of number A
         A = [4,5,-6,7,0]
         #call function "number" and test the numbers in list A
         out4 = number(A)
```

Figure 45: A function that evaluates each number of a list whether it's positive, negative, or zero.

9.1 Problem: Simple Harmonic Motion

Many kinds of motion repeat themselves over and over: the vibration of a quartz crystal in a watch, the swinging pendulum of a grandfather clock, the sound vibrations produced by a clarinet or an organ pipe, and the back-and-forth motion of the pistons in a car engine. This kind of motion is called a periodic motion or oscillation [6]. A body that undergoes periodic motion always has a stable equilibrium position. When it is moved away from this position and released, a force or torque comes into play to put it back toward equilibrium. But by the time it gets there, it has picked up some kinetic energy, so it overshoots, stopping somewhere on the other side, and is again pulled back toward equilibrium. Fig(46) shows one of the simplest systems that can have periodic motion. A body with mass m rests on a frictionless horizontal guide system, such as a linear air track, so it can move along the x-axis only. The body is attached to a spring of negligible mass that can be either stretched or compressed. The left end of the spring is held fixed, and the right end is attached to the body. The spring force is the only horizontal force acting on the body; the vertical normal and gravitational forces always add to zero.

The simplest kind of oscillation occurs when the restoring force F is directly proportional to the displacement from equilibrium x. This happens if the spring

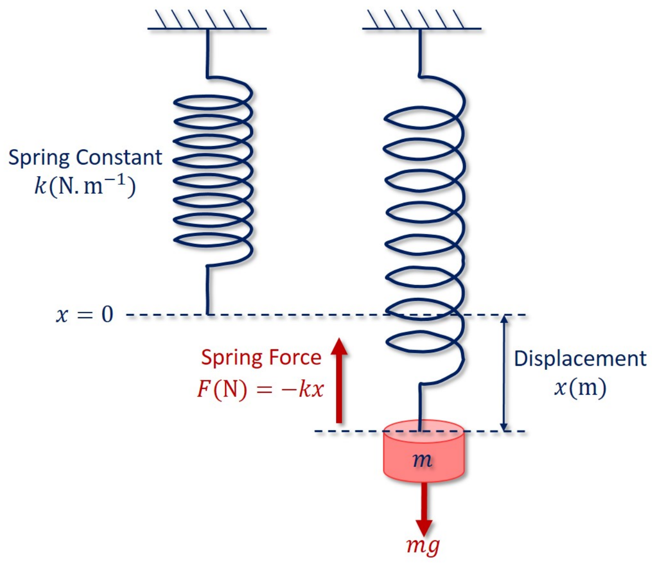


Figure 46: Model for periodic motion. When the body is displaced from its equilibrium position at $x = 0$, the spring exerts a restoring force back toward the equilibrium position [6].

is an ideal one and obeys *Hooke's law*. The constant of proportionality between F and x is the spring constant k and these quantities are related as:

$$F = -kx \tag{7}$$

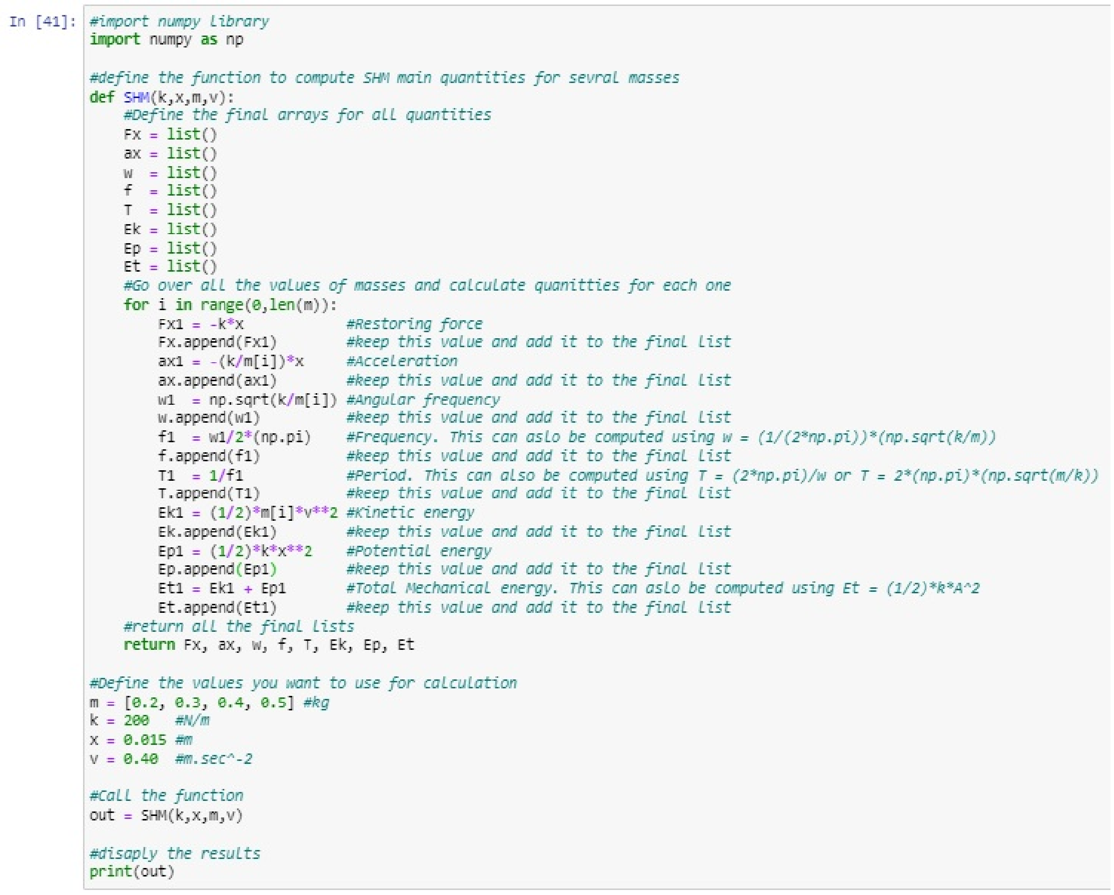
This equation gives the correct magnitude and sign of the force, whether x is positive, negative, or zero. The spring constant k is always positive and has units of (N/m). The oscillation modeled by this equation is called *simple harmonic motion (SHM)*. This motion can be further understood by defining several other physical quantities as shown in Table(2).

Our aim now is to write a Python function that computes all of the physical quantities listed in Table(2) by inserting only the spring constant k, displacement x, velocity v, and mass m as inputs. Fig(47) shows the code that does this computation for several mass values and outputs the results as lists. This is informative as it tells us what happens to these quantities as the attached mass changes its value.

Notice the use of the `append()` method. The `append()` method appends an element to the end of the list. This method allows keeping the results of each iteration and assigning them to the pre-assigned lists. Hence, the lists change their sizes in each iteration. If this method is not used, Python only shows the results of the last calculation throwing away the previous ones.

Quantity	Equation	Definitions
Restoring Force	$F = -kx$	
Acceleration	$a_x = -\frac{k}{m}x$	m is the mass
Angular Frequency	$\omega = \sqrt{\frac{k}{m}}$	
Frequency	$f = \frac{\omega}{2\pi} = \frac{1}{2\pi}\sqrt{\frac{k}{m}}$	
Period	$T = \frac{1}{2f} = \frac{2\pi}{\omega} = 2\pi\sqrt{\frac{m}{k}}$	
Kinetic energy	$E_k = \frac{1}{2}mv^2$	v is the velocity
Potential energy	$E_p = \frac{1}{2}kx^2$	
Total mechanical energy	$E_t = E_k + E_p = \frac{1}{2}kA^2$	A is the amplitude

Table 2: Main physical quantities of simple harmonic motion.



```
In [41]: #import numpy library
         import numpy as np

         #define the function to compute SHM main quantities for sevral masses
         def SHM(k,x,m,v):
             #Define the final arrays for all quantities
             Fx = list()
             ax = list()
             w  = list()
             f  = list()
             T  = list()
             Ek = list()
             Ep = list()
             Et = list()
             #Go over all the values of masses and calculate quanitties for each one
             for i in range(0,len(m)):
                 Fx1 = -k*x            #Restoring force
                 Fx.append(Fx1)        #keep this value and add it to the final List
                 ax1 = -(k/m[i])*x     #Acceleration
                 ax.append(ax1)        #keep this value and add it to the final List
                 w1  = np.sqrt(k/m[i]) #Angular frequency
                 w.append(w1)          #keep this value and add it to the final List
                 f1  = w1/2*(np.pi)    #Frequency. This can aslo be computed using w = (1/(2*np.pi))*(np.sqrt(k/m))
                 f.append(f1)          #keep this value and add it to the final List
                 T1  = 1/f1            #Period. This can also be computed using T = (2*np.pi)/w or T = 2*(np.pi)*(np.sqrt(m/k))
                 T.append(T1)          #keep this value and add it to the final List
                 Ek1 = (1/2)*m[i]*v**2 #Kinetic energy
                 Ek.append(Ek1)        #keep this value and add it to the final List
                 Ep1 = (1/2)*k*x**2    #Potential energy
                 Ep.append(Ep1)        #keep this value and add it to the final List
                 Et1 = Ek1 + Ep1       #Total Mechanical energy. This can aslo be computed using Et = (1/2)*k*A^2
                 Et.append(Et1)        #keep this value and add it to the final List
             #return all the final Lists
             return Fx, ax, w, f, T, Ek, Ep, Et

         #Define the values you want to use for calculation
         m = [0.2, 0.3, 0.4, 0.5] #kg
         k = 200   #N/m
         x = 0.015 #m
         v = 0.40  #m.sec^-2

         #Call the function
         out = SHM(k,x,m,v)

         #disaply the results
         print(out)
```

Figure 47: Python function that calculates several physical quantities of the simple harmonic motion: restoring force, acceleration, angular frequency, frequency, period, kinetic energy, potential energy, and the total mechanical energy, given only the mass, displacement, spring constant, and the velocity.