2. Functions in python

Function syntax has already been introduced in the previous chapters, we have already worked with some of functions from python's standard library like `print()`, `list()`, `tuple()`, `len()`, `id()` and some other functions. The python library has several functions that can be used for common programming tasks.

Functions in mathematics are of the form `f(x) = y`. Here function 'f' when given the input '`x`' performs operations on 'x' and gives 'y' as the output. An example of function a definition in mathematics is

```
f(x) = 3x + 4
```

From this function definition we can compute the values of `f(x)` for the different values of 'x' as below.

```
f(3) = 13
```

```
f(5) = 19
```

```
f(0) = 4
```

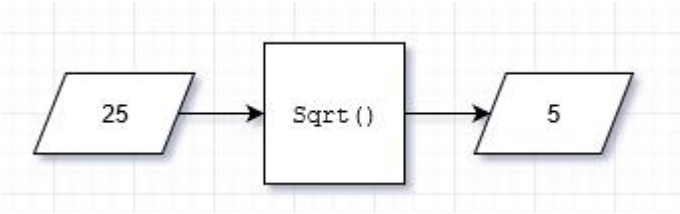
Functions in python is a block of code that is given a name(like the name of above function is 'f'). This block of code can be called any where in the program, when called control is shifted to the body of the called function. The above function 'f' can be written in python as follows.

```
In [1]: def f(x):return 3 * x + 4

In [2]: f(3)
Out[2]: 13

In [3]: f(5)
Out[3]: 19

In [4]: f(0)
Out[4]: 4
```

One example of a function in pythons standard library is the square root function, this function is named `sqrt`. The square root functions takes numeric as parameter and returns a float value, like square root of "25" is '5'.

Functions should be thought of as tools that perform a certain operation, so it's important to know "what a function does" than "how it does it". Thinking of functions this way allows the programmer to see functions as a black box that performs the desired operation. In the case of the square root function it allows the programmer to focus on the task at hand rather than focusing on the algorithm to calculate the square

root of a number, like for example to make a program that return the square root of a number provided by the user, the `sqrt()` function can be used to avoid the complex logic of computing the square root of a number. The code example below returns the square root of a number provided by the user by using the `sqrt()` function instead of implementing the logic for square root calculation.




```
1  from math import sqrt
2
3  # ask user for input
4  num = eval(input("Enter number: "))
5
6  # Compute the square root with sqrt()
7  # sqrt() is responsible for square root calculation,
8  # allowing the programmer to focus on whats important
9  root = sqrt(num);
10
11 # print the result
12 print("Square root of", num, "=", root)
```

Output for the above program is below.

```
Enter number: 25
Square root of 25 = 5.0
```

Here `sqrt(num)` is function invocation or a function call(a call to the `sqrt()` function). A function provides a service to it's the the caller(invoker of the service). The above code that we have to compute square root of a number is the "calling code" or the "client code" to the `sqrt()` function which acts like a service that can be invoked

```
root = sqrt(num)
```

The above statement invokes `sqrt()` passing it the value of num. The expression `sqrt(num)` the square root of the value of the variable "num".

Unlike the small collection of functions(like `type, str, int, range, id`) that are always available in python programs, the `sqrt()` function is available in the "math" module and can not be used unless the math module is imported, hence the purpose of the "import" statement(the first line of code)

```
from math import sqrt
```

The above code imports the `sqrt()` function from the "math" module. A module is a collection of code that can be imported and used in other python programs. The "math" module is very rich in the functions it provides to perform math operations, it provides functions for trigonometric, logarithmic and several other math operations.

A function has a name followed by a set of parentheses which contain the information that the function needs to perform it's task.

```
Sqrt( num ) # sqrt needs a parameter to return the square root of
```

`num` is the value that the `sqrt()` function operates on. The values that are passed to a function are called the parameters or arguments of the function. Functions after they perform the computations it was defined to perform can return a value back to the caller(client invoking it) like in the case of the `sqrt()` function which returns the square root of the parameter passed as a float type to its invoker. This value that is returned to the client can be used by the client just like any other value.

```
In [3]: print(sqrt(81))
9.0
```

The `print()` function above can use the value returned by the `sqrt()` function.

```
In [4]: y = sqrt(144)

In [5]: y
Out[5]: 12.0
```

Like in the above assignment statement, the value returned by a function can be assigned to a name. When the client code attempts to pass an argument to the function it's invoking that has a type that is different then what is expected by the function, then the interpreter raises an error.

```
In [6]: sqrt('12')
Traceback (most recent call last):

  File "<ipython-input-6-b61ce7c9721f>", line 1, in <module>
    sqrt('12')

TypeError: must be real number, not str
```

A function can be called any number of times anywhere in the program. Remember to he the client of a function, the function is like a labeled black box, the knows client knows what the function bing called does and is not concerned with how the task is performed. All functions can be treated like black boxes. A service that a function provides can be used without being concerned of the internal details.

The only way a behaviour of a function can be altered is through the functions parameters(the arguments that are passed to a function can be different). When immutable types(int, float, string, tuple) are passed as arguments to a function, a copy of the objects are passed instead of a reference to the original objects, therefore immutable objects when passed to a function as an argument can not be changed by the called function. In the case that the arguments passed are mutable objects a reference to those objects are passed instead of a new copy of those objects, therefore mutable objects when passed to a function as an argument can be changed by the invoked function.

As we already seen before functions can take more than one parameter like the `range()` function that can take one, two or three parameters.

From the view of the client/caller, a function has three parts. Lets discuss eac of them in brief.

Name

A function's name is the handle that is used to identify the code to be executed when it is invoked. The same rules that apply to variable names apply to function names. Function names, like variable names are also identifiers.

Parameters

Functions are called with zero or more parameters. The types of each argument passed in the invocation of a function must be the same as the types of parameters in the function definition. If more are or less parameters are passed than what is expected the interpreter raises an error. Some examples of this these two kinds of errors are illustrated in the shell execution below.

```
In [1]: from math import sqrt

In [2]: sqrt("123")
Traceback (most recent call last):

  File "<ipython-input-2-9c20d16c4ba7>", line 1, in <module>
    sqrt("123")

TypeError: must be real number, not str


In [3]:

In [3]: sqrt(625,6,3)
Traceback (most recent call last):

  File "<ipython-input-3-5521980cd00a>", line 1, in <module>
    sqrt(625,6,3)

TypeError: sqrt() takes exactly one argument (3 given)


In [4]:

In [4]: sqrt()
Traceback (most recent call last):

  File "<ipython-input-4-a69d70ec25ee>", line 1, in <module>
    sqrt()

TypeError: sqrt() takes exactly one argument (0 given)
```

Return type

A function always returns a value to it's caller. The client code should be compatible with the type of the object returned by called function. The return type and the parameter types are not related. Even in the case that the function does not specifically return any value, a special "none" value is return to the client.

There are functions that take no parameter like the `random()` function. The `random()` function returns a random floating point value and it belongs to the "random" module. Again if the arguments passed does not match with the number of parameters and the type of each parameter in the function definition then an error is raised by the interpreter. You can see in the below code that if a parameter is passed to the `random()` function an error is raised.

```
In [1]:  from random import random

In [2]: random() # Takes no paramters
Out[2]: 0.26771707004169076

In [3]: random(10, 20) # error raised when passed arguments
Traceback (most recent call last):

  File "<ipython-input-3-a4f2f2b8f180>", line 1, in <module>
    random(10, 20) # error raised when passed arguments

TypeError: random() takes no arguments (2 given)
```

One of the ways that functions can be categorized is as,

- **Functions without side effects:** These types of functions just perform the required task and return the result to the client caller and that is all it does, it makes no other changes like printing to the display, changing the variables of the caller, etc. This type of functions always return a result(not none) to the invoking client
- **Functions with side effects:** These types of variables perform the required task which may also require to change it's environment like printing to the display or changing the value of the caller, etc. This type of functions may sometimes return a special value "none" to the invoking client.

2.1 Standard mathematical functions

The "math" module provides almost all of the functionality of a scientific a scientific calculator. Some of he function of the "math" module a re listed below.

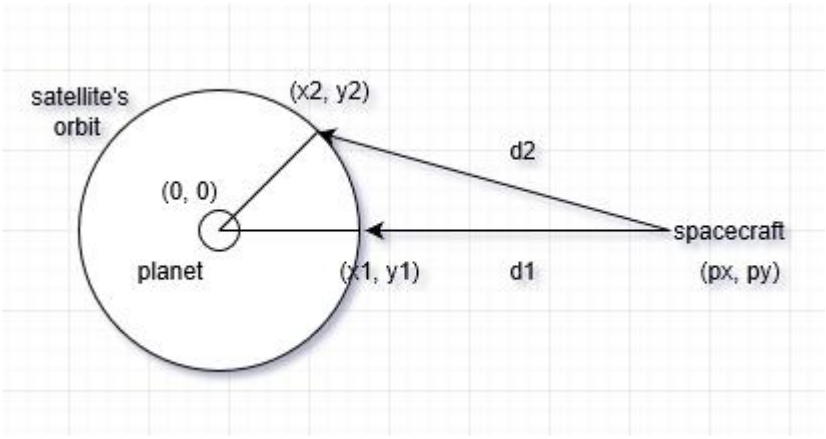
Function	description
`sqrt`	Computes the square root of a number.
`exp`	Computes e raised a power
`log`	Computes the natural logarithm of a number.
`log10`	Computes the common logarithm of a number.
`cos`	Computes the cosine of a value specified in radians: cos(x) = cos x; other trigonometric functions include sine, tangent, arc cosine, arc sine, arc tangent, hyperbolic cosine, hyperbolic sine, and hyperbolic tangent

`pow`	Raises one number to a power of another.
`degrees`	Converts a value in radians to degrees.
`radians`	Converts a value in degrees to radians.
`fabs`	Computes the absolute value of a number.

The parameters passed to the function by the calling client code is called the actual parameters(arguments) and the parameters that are specified in the function definition are the formal parameters. When a function is invoked the first actual parameter is assigned to the first formal parameter, then the second actual parameter is assigned to the second formal parameter, so the assignment of actual parameters to formal parameters happens from left to right. The order of passing the assignment matters, for example calling the `pow()` function as `pow(10,2)` returns 100 but calling it as `pow(2, 10)` returns 1024.

Lets try to solve a trigonometry problem with the functions provided by the math module. The problem is as follows.

Suppose that there is a spacecraft at some distance from a plane. A satellite orbits this planet, the problem is to find how much farther a way the satellite would be away from the spacecraft if the the satellite moves 10 degrees along it's orbital path. The spacecraft is on the plane of the satellites orbit.



Let the center of the planet be the origin(0, 0) of the coordinate system, this is also the center of the satellites orbit. In this problem the spacecraft is located at the point (p_x, p_y), the satellite is initially at point (x_1, y_1) then moves to point (x_2 , y_2). Here the the points (x_1, y_1) and (p_x, p_y) are given and the problem is to find the the difference between d_1 and d_2. This problem can be solved in two steps as follows.

Step 1

Calculate the point (x_1, y_1). For any given point (x_1, y_1) a rotation of theta degrees gives a new point (x_2, y_2), where

$$\begin{aligned} x_2 &= x_1 \cos\theta - y_1 \sin\theta \\ y_2 &= x_1 \sin\theta + y_1 \cos\theta \end{aligned}$$

Step 2

The distance d_1 between the two points (p_x, p_y) and (x_1, y_1) is given by,

$$d_1 = \sqrt{(x_1 - p_x)^2 + (y_1 - p_y)^2}$$

The same way, distance d_2 between the two points (p_x, p_y) and (x_2, y_2) is given by,

$$d_2 = \sqrt{(x_2 - p_x)^2 + (y_2 - p_y)^2}$$

The python program below solves the orbital distance problem for the spacecraft at location (100, 0) and user provided starting location for the satellite. You can see that the standard math problem was used in the program to com at the solution.

```
#=====================================================================
# Python program for the orbital distance problem
#=====================================================================
from math import sqrt, cos, sin, pi

# The orbiting point is (x,y)
# A fixed point is always (100, 0),
# (p_x, p_y). Change them as necessary.
p_x = 100
p_y = 0

# number of radians in 10 degrees
radians = 10 * pi/180

# The cosine and sine of 10 degrees
cos10 = cos(radians)
sin10 = sin(radians)

# ask the satellites starting point from the user
x1, y1 = eval(input("Enter the satellites initial coordinates (x1,y1):"))

# calculate the initial distance d1
d1 = sqrt((p_x - x1)**2 + (p_y - y1)**2)

# calculate the new location (x2, y2) of the satellite after movement
x2 = x1*cos10 - y1*sin10

y2 = x1*sin10 + y1*cos10

# Compute the new distance d2
d2 = sqrt((p_x - x2)**2 + (p_y - y2)**2)

# Print the difference in the distances
print("Difference in the distances: %.5f" % (d2 - d1))

```

The output when the above program is executed for the starting location (10, 10) is,

```
Enter the satellites initial coordinates (x1,y1):10, 10
Difference in the distances: 2.06192
```

2.2 Time functions

Pythons “time” module provides a lot of functions that involve time. In this section we will take a look at two of the functions that are provided by this module the `clock()` and `sleep()` functions.

The clock function allows measuring time in seconds and it can be used anywhere in a program allowing the measure of time at any part of a program. The clock function works differently based on the operating system that it is being executed on. On unix like operating systems like linux and mac OS, the clock() function returns the number number of seconds elapsed from the start of the programs execution. On Microsoft windows the clock() function returns the number of seconds elapsed from the previous call to the clock() function. The return the number of seconds elapsed as a floating point number. The difference between the returned values of two clock function calls can be used to find the time period between the two calls.

The below program returns the amount of time it took the user to respond.

```
1  #=====================================================================
2  # This program prints the time it took the user to respond in seconds
3  #=====================================================================
4  from time import clock # import the clock method
5
6  print("User respond! : ")
7  startTime = clock()
8  input()
9  endTime = clock()
10 print("It took you %f seconds to respond " % (endTime - startTime))
11
12
```

An output a very slow user might get is,

```
User respond! :

Hello
It took you 18.599143 seconds to respond
```

Here is a another program that displays the amount of time it took to add the first ten million natural numbers.

```
1  #=====================================================================
2  # Time taken to add the first million natural numbers
3  #=====================================================================
4  from time import clock # import the clock method
5
6  sum = 0 # initialize sum as 0
7
8  print("started adding")
9  startTime = clock()
10
```

```
for i in range(10000000):
    sum += i

endTime = clock()
print("It took %f seconds " % (endTime - startTime))

```

The output of this program is given below.

```
started adding
It took 1.363418 seconds
```

The sleep function can be used to suspend a program for a period of time in seconds. The number of seconds to suspend the program for is passed as an argument to the sleep function. The below program counts down to zero from three with one second intervals.

```
#=====================================================================
# Three second count down timer!
#=====================================================================
from time import sleep    # Import the sleep method

timer = int(3)            # Initialize the timer to a predefined value

print("Count down starts... %i" %timer)

while timer > 0:
    sleep(1)
    timer -= 1
    print(timer)

sleep(1)

```

The proves to be very useful when in comes to controlling the speed in graphical animations. This program when ran produces the following output.

```
count down starts... 3
2
1
0
```

2.3 Random functions

The package random provides some functions that deal with randomness. Applications that require the behaviour of randomness need these kind of functions. Almost all random number generating algorithms produce “pseudo random” numbers which means they are not truly random. pseudo random number generators when repeated long enough produce results that start repeating the exact same sequence of numbers. This not a serious problem as the length of this sequence is long enough to be considered safe for almost all practical applications.

Pythons “random” module consists of a number of functions that enable the use of pseudo random numbers in programs. Few of the commonly used functions of this module are described below.

Function	Description
`random()`	Returns a pseudo random floating-point number x in the range `0 <= x < 1`
`randrange()`	Returns a pseudo random integer value within a specified range.
`seed()`	Sets the random number seed.

The `seed()` sets the value from which the sequence of the pseudo random numbers are generated. Every time the functions `random()` and `randrange()` are called they return the next value in the sequence of pseudo random values. The program below, prints ten random numbers in the range of one to ten.

```
1  from random import randrange, seed
2
3  seed(10)
4
5  for i in range(10):
6      print(i+1,")",randrange(1, 10))
7
```

This program produces the output shown below.

```
1 ) 1
2 ) 7
3 ) 8
4 ) 1
5 ) 4
6 ) 8
7 ) 8
8 ) 5
9 ) 3
10 ) 1
```

The function seed() sets the “seed number” that is used to generate the pseudo random number sequence and as there is a dependency if the seed value is changed then then sequence of pseudo random random numbers will also change. This dependency is in one way a problem because if the above program is run again it produces the exact same sequence of random numbers. This is illustrated in the example program shown below.

```
1  from random import randrange, seed
2
3  seed(10) # The seed is 10
4
5  for i in range(3):
6      print(i + 1, ")", randrange(1, 10))
7  print("----------")
```

```
8
9  seed(14) # Change the seed to something else
10
11 for i in range(3):
12     print(i + 1, ")", randrange(1, 10))
13 print("----------")
14
15 seed(3)
16
17 for i in range(3):
18     print(i + 1, ")", randrange(1, 10))
19 print("----------")
20
```

From this program you can see that the same sequence of pseudo random numbers are generated for the same value of the seed. Having constant value of the seed isn't a safe way to generate random numbers, but it can be useful while testing a program. When the value of the seed is not specifically mentioned, then the system time determines the seed. This allows the pseudo random numbers generated to have a more random behaviour. The out put of the above program are shown below.

```
1 ) 1
2 ) 7
3 ) 8
--------
1 ) 4
2 ) 4
3 ) 7
--------
1 ) 1
2 ) 7
3 ) 8
--------
```

Now lets write a program that simulates the rolling of a six sided die. A die has six sides labeled one to six with one of the sides that can land on top when the die is rolled. We want to user to provide information on the number of dice to be thrown. The random number generated should lie between zero and six, so the randrange(1, 6) will be used for this, and it will be called for each die thrown.Lastly this entire setup should be placed in a loop that continue indefinitely until the user provides the number zero as input.

```
1  from random import randrange
2
3  # skipping the seed setup
4
5  sides = 7
6
7  while 1:
8      diceCount = int(input("Enter the number of dice to throw :"))
9      for i in range(diceCount):
10         dieValue = randrange(1,sides)
11         print("%i " % (dieValue), end = " " )
12
```

An output for the above program is given below.

```
Enter the number of dice to throw :2
1  3
Enter the number of dice to throw :5
5  5  5  6  4
Enter the number of dice to throw :10
2  5  6  1  6  5  1  4  4  6
Enter the number of dice to throw :0
```

2.4 Why write your own functions

All the programs that we have seen till now where very small and come no where near the size of practical applications. When the programs size increase the complexity of the program increases as well and it starts to get difficult to manage this complexity. Programs like the ones we have gone through are “monolithic code”, the program execution flows from start to finish, the these types of programs are like one big block of code,when the size of a monolithic program grows it’s complexity starts to get out of hand. The solution to this is to divide the code into several almost independent smaller pieces with very specific roles. This division of a programs complexity into smaller pieces can be done using functions. A program can be made up of multiple functions, each one having the task of addressing a part of the problem and when used together solves the problem as a whole.

One big and complex monolithic code is avoided for several reasons.

- **Difficulty writing correctly:** When writing any statement, the details of the entire code must be considered. This becomes unmanageable for programmers beyond a certain point of complexity.
- **Difficulty debugging:** The larger a code block is, the harder it is to find the source of an error or an exception in that block of code. The effect of an erroneous statement might not be know until later in the program a correct statement uses the result of the erroneous statement.
- **Difficulty in modification:** Before a code sequence can be modified to achieve a different result, this code sequence needs to be understood, which becomes a very heavy task as the size of code sequence increases.

Our code can be made more manageable by dividing our code into parts and rewriting these parts of our code as functions, calling them in the program when and as needed. Using this divide and conquer strategy, a complicated code block can now be be broken down into simpler code blocks each being handled by a function. The programmer can now accomplish the same original task by delegating tasks to functions, this way a results in more manageable code blocks that are easy to read, write, debug and modify. Besides these advantages of structuring the code, functions provide several other advantages like reusing same code block for the same kind of task(reusability). Functions that provides similar functionality can be bundled together into reusable parts.

Although it’s time saving to use library functions, it is sometimes necessary to write our own functions for custom behaviour. Fortunately python allows programmers to create their own custom functions and use them.

If the purpose of the custom functions are general enough and written in a proper manner then this function can even be used in other programs. In the following sections we'll take the first steps in writing functions in python.

2.5 Functions basics

Every python function has

1. **Function definition:** A functions definition describes every detail about that function including how many parameters and what type of parameters it takes, what the function does and also what the function returns.

2. **Function invocation(call):** After a function is defined, it can be called from anywhere in the program by passing the appropriate parameters to the function. A function can have only one definition but it can be invoked any number of times.

A function definition has three parts, they are listed below.

- **Name:** The name of a function is what is used to identify the block of code that this function contains. Function names are also identifiers, so they follow all the rules of python identifiers. One of the best ways to name a function is to give it the name of the task it is designed to do.

- **Parameters:** The parameters that the function accepts from callers is specified in the function definition. They appear with a set of parentheses(( )) in a comma(,) separated list. The list of parameters can be left empty if the functions definition does not specify parameters to passed, these types of function do not require information from the caller to perform the task it was designed to carry out.

- **Body:** The block of indented statements that follow after the functions header is the functions body. This is what gets executed when the function call is made. If the function is required to produce a result and return it to the client, the `return` statement can be used to return the result back to the caller.

The syntax of a function definition is given below.

```
def <function_name>(<parameter list>):

    """<docstring>"""

    <statement(s)>
```

The above function definition syntax has the following the components.

- `def` is the keyword that signifies the start of the header of a function

- `<function_name>` is the unique name that identifies the the body of the function

- `<parameter list>` is used to pass arguments to the function. They are optional.

- “:” signifies the end of the function header
- <”””docstring”””> is the documentation string that can be used to describe the purpose and functionality of the function
- <statement(s)> are the valid python statements that make up the body of the function. All the statements in the function body have the same indentation usually four spaces(“ ”) area used.
- A return statement is optional and it is used to return the result back to the caller f any

The example code below illustrates a simple function definition and calling this defined function.

```
1  def greet(name):
2      """This function prints a hello and doesn't return anything"""
3      print("Hello " + name)
4
5  greet("khalid")
6
```

In the above code example a function “greet” is defined that takes a single string(name) as a parameter and prints a specified string onto the prompt. Then the defined function is immediately called after it’s definition, this executes the body of the greet function with the passed arguments.

2.5.1 Docstring

The first line after the function header optionally has a string called the “docstring” that is sort for “document string”. The “docstring” is used for explaining what the function does.

If you have already programmed in other languages you would be aware how programmers tend to forget what their code does, so it’s always a best practice to document your code. Even if you may not use the documentation to refresh your memory on what the function does, it can help others who will be working with your code.

The docstring of a function can be accessed with “.__doc__” after a function name. So if the following code is entered into the python shell it returns,

```
In [3]: print (greet.__doc__)
This function prints a hello and doesnt return anything
```

2.5.2 The return statement

The return statement for functions is like how the break statement is for loops in a way that they both takes the control out of the block they were encountered at. The return statement exits the function, passes the control back to the place it was called from the calling code additionally the return statement contains followed by an expression list that is evaluated and returned back to the caller.

The syntax of the return statement is given below.

```
return <expression_list>
```

Here the expression in the `return` statement is evaluated and returned back to the client. In the case that there is no expression provided or the `return` statement itself is skipped then a special object `none` is returned back to the calling client.

For example try the following code in the python shell. Print the value that is return by the greet function.

```
In [4]: print(greet("Earth"))
Hello Earth
None
```

In the above call to print, which is passed a call to the greet function as the parameter. First the greet method is called which prints to the prompt then the greet method returns a “none” object back to the print function, the print function then prints the “none” object

Lets take a look at an example that illustrates the `return` statement. The program below defines and uses a function that returns the absolute value of an number that’s passed to it. You can see how the return evaluates the expression given to it and returns the result as well as the control back to the caller. Remember that an expression is any combination of values, variables, operators, function calls. Expressions always return an object. If there is no return statement in a function the execution of the function ends when the last statement of the functions code block has been executed and the “none” object as well as the control is returned back to the caller.

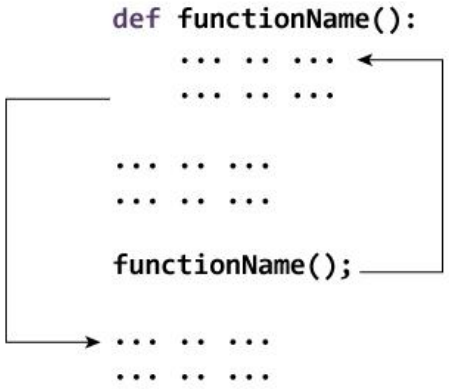
```
1  #===============================================================
2  # Evaluate absolute value of a number
3  #===============================================================
4  def absolute_value(num):
5      """This function returns the absolute
6      value of the number passed as an argument"""
7
8      if num >= 0:
9          return num
10     else:
11         return (-num)
12
13 userInput = eval(input("Enter a number: ")) # evaluate users input
14
15 print("It's absolute value is " + str(absolute_value(userInput)))
16
```

The output for the above program for the value “-56” is shown below.

```
Enter a number: -56
It's absolute value is 56
```

2.5.3 Flow of control at function call and return

The below figure illustrates the control flow when a function call is made and when the the execution of the functions code block completes.



When a function call is made the control shifts to the body of the called function, the only two ways that control can exit from a function is by the return statement or by executing the last statement in the body of the function.

2.5.4 The scope and lifetime of variables

Scope of a variable in a program is the part of the program that can recognize and use the object referred to by that variable. Parameters of a function are considered to be local to that function(local variables) and so they are in the scope of that functions code block but outside the scope of the rest of the program. A functions local variables can’t be used by the parts of the program outside that functions code block hence they have local scope.

Lifetime of a variable is the period through which the variable exists in the memory. The lifetime of variables inside functions(i.e local variables) is as long as the function executes. After the function completes it’s execution all it’s variables are deleted and their associated objects are garbage collected, and because of this functions don’t have any memory between two function calls, they don’t know what the value of local variable was in the previous call.

The below example code illustrates the scope of a variable inside a function

```
#=========================================================================
# This program illustrates the scope of a variable inside a function
#=========================================================================
def printLocalx():
    x = 10     # Local variable, scope lines 2 - 4
    print("This is printLocalx() function's value of x: ", x)
    print("This is printLocalx() function's value of y: ", y)
    # Local variable x is deleted once the control flow reaches this point

x = 20         # Global variable, scope lines 1 - 8
y = 40         # Global variable, scope lines 1 - 8
printLocalx()
print("Value outside the function:", x)

```

The above program produces the below output when executed. Observe that even though the the global variable `x` is in the scope of `printLocalx()` function, the function prefers the value of it's local variable when there is a match.

```
This is printLocalx() function's value of x : 10
This is printLocalx() function's value of y : 40
Value outside the function: 20
```

Even though function `printLocalx()` changes the value of the the variable x, it does not affect the global variable `x`. This is because variable `x` inside the `printLocalx()` function is different from the one outside of it, so remember that they are two different variables with different scopes.

In Python the `global` keyword allows modification variables that are outside the current scope. The can be used to create global variables and modify it in the local context.

Things to remember when using the global keyword.

1. A variable is by default a local to a function when it is declared inside the function
2. A variable is by default a global variable when its declared outside a function. It makes no difference using the global keyword on variables defined outside functions.
3. The global keyword can be used inside functions to read and write global variables inside functions.

Lets take a look at some examples that illustrate the use and working of the global keyword.

```
x = 20        # x is a global variable

def printx():
    print(x) # Since there is no local variable x, the global variable x
             # can be accessed

printx()

```

This example code produces the below output.

```
20
```

Here, notice that as the function `printx()` doesn't have a local variable 'x', it can read the value of the global variable 'x', but the interpreter raises an error when we try to write to the same variable.

```
1  x = 20       # x is a global variable
2
3  def printx():
4      x = x + 4 # Try using the value of a global variable.
5                # Local variable x is referenced before
6                # it's assignment
7      print(x)
8
9  printx()
10
```

When the above program is ran it raises the error below,because the value of a global variable is being modified in the function.without using the global keyword it is only possible to read the value of a global variable inside a function

```
  File "C:/Users/khalid/Documents/ML/Bucca/global2.py", line 4, in printx
    x = x + 4

UnboundLocalError: local variable 'x' referenced before assignment
```

The `global` keyword can be used to overcome this kind of a problem where the modifications to global variables from within a function is required. The example below shows how a global keyword can be used to modify a global variable from within a function. To modify a global variable from inside a function this variable has to first to be declared as `global` inside the function it needs to be used in. This is illustrated in the example below.

```
1  x = 20       # x is a global variable
2
3  def printx():
4      global x  # Now the global variable x can be modified
5      x = x + 4 # Try using the value of the global variable.
6                # you'll see there is no error raised this time
7      print("This is the value of x inside function printx(): %i" %x)
8
9  printx()
10 print("This is the value of x outside function printx(): %i" %x)
11
```

The output of the above program is given below. notice that the value of the variable 'x' is the same inside and outside the `printx()` function. This is because variable 'x' inside the `printx()` function now refers to the same object as the variable 'x' outside this function.

```
This is the value of x inside printx: 24
This is the value of x outside printx: 24
```

More on the global and local scopes will be discussed in the following sections.

2.5.5 Arguments

Functions may take any number of parameters(No upper or lower bound) to perform the task that it was designed for. Before calling a user defined function, this function must be defined and it's function definition must specify the parameters it needs. When a function is called by a client code, the client caller must provide the same number of arguments as specified in the function definition. if there is any mismatch in the number of arguments passed then what was expected then the python interpreter raises an error. The examples below executed in the python shell show the errors raised when there are issues with the arguments passed to a function call

```
In [1]: def square(x): return x * x  # Takes 1 integer and returns it's square

In [2]: square(9, 8 , 0)             # Number of arguments don't match
Traceback (most recent call last):

  File "<ipython-input-2-50d9f47ac714>", line 1, in <module>
    square(9, 8 , 0)             # Number of arguments don't match

TypeError: square() takes 1 positional argument but 3 were given


In [3]:

In [3]: square(9)                    # A valid call
Out[3]: 81

In [4]: square("Can I be passed?")   # Valid call, error raised from function body
Traceback (most recent call last):

  File "<ipython-input-4-6f0de8c10498>", line 1, in <module>
    square("Can I be passed?")   # Valid call, error raised from function body


  File "<ipython-input-1-b50f9e4cba73>", line 1, in square
    def square(x): return x * x  # Takes 1 integer and returns it's square

TypeError: can't multiply sequence by non-int of type 'str'
```

Observe that the number of arguments to pass are compulsory but the types of the arguments are not. Although passing an argument of a different type than what is expected would most likely end up raising an error.

Python also allows it's users to define functions that take a variable number of parameters. Functions that take a variable number of parameters can be defined with the following types of arguments. We'll discuss each of them in the following section.

1. Default arguments

2. Keyword arguments
3. Arbitrary arguments

Default and non-default arguments

Python allows to assign default values for function parameters A default value for a parameter can be given with the "=" operator. When a parameter is assigned a default value, its argument can be skipped in the function call in which case the default value provided in the function definition gets assigned instead. In the example below we create our own range function that in turn calls python's standard range function. This example illustrates python's default arguments.

```
1  def My_range_func(stop, start = 0, step = 1):
2      """
3      This function in turn calls
4      the standard range function
5      and returns it's result as
6      a list
7      """
8
9      li = list(range(start, stop, step))
10     return li
11
```

Now when this program is executed, try making calls to this function with different parameters from the python shell. Observe that the function call can be made with one, two or three parameters, this is because the second and third arguments are default arguments are default arguments and when their values are not provided they assume their default values. Here it is necessary to pass a value for parameter `stop` and failing to do so would raise an error. For illustration there are some function calls to the `My_range_func()` given below. These function calls where made from the python shell.

```
In [1]: runfile('C:/Users/khalid/Documents/ML/Bucca/myrangefunc.py', wdir='C
Bucca')

In [2]: My_range_func(20, -3, 2)
Out[2]: [-3, -1, 1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

In [3]: My_range_func(12, 4)
Out[3]: [4, 5, 6, 7, 8, 9, 10, 11]

In [4]: My_range_func(9)
Out[4]: [0, 1, 2, 3, 4, 5, 6, 7, 8]
```

Here the parameter `start` does not have a default value and is mandatory to provide in a call. Default values `0` and `1` are provided for parameters `start` and `step` and so they're value can be skipped in a function call but if provided they overwrite the default values. Any number of parameters in a function can have a default value but it must be made sure that all default values are pushed to the right of the parameter list. This is needed as without it there would be difficulty differentiating between default and compulsory arguments in a function call. To ensure this the python interpreter

raises an error when it encounters a function header that doesnt have all default parameters pushed to the right. An example is given below.

```
1 def My_range_func( start = 0, stop, step = 1): # This raises an error
2
```

```
SyntaxError: non-default argument follows default argument
```

Keyword and positional arguments

When a function is called by passing arguments to it, the arguments are assigned to the parameters based on it's position in the argument list which is from left to right. For instance in the above function `My_range_func()` when `(20, -3, 2)` is passed as the argument list, the assignment occurs in the following order.

1. `20` gets assigned to variable `stop`
2. `-3` gets assigned to variable `start`
3. `2` gets assigned to variable `step`

These arguments are positional arguments as the position of the arguments matter in the function call. Python allows re-ordering of the arguments that are passed if the arguments are passed as keyword arguments. When function call is made with keyword arguments the arguments can be passed in any order. The general syntax of argument list when passing arguments as keyword arguments is given below.

```
<function_name>(<ParamterA> = <value>, <ParameterB> = <value>,...)
```

The following are calls to the previously defined `My_range_func()` function and they are all valid.

```
In [2]: My_range_func(start = 3, step = 1, stop = 8)
Out[2]: [3, 4, 5, 6, 7]

In [3]: My_range_func(step = 3, stop = 10)
Out[3]: [0, 3, 6, 9]

In [4]: My_range_func(step = 3, stop = 10, start = 5)
Out[4]: [5, 8]
```

A function call can be made with a mixture of positional and keyword arguments but in this case the keyword arguments must be pushed to the right side of the argument list which means there should not be any positional arguments that follow keyword arguments in the argument list of a function call. The example below show a valid and invalid instance of making function calls with combination of both positional and keyword arguments.

```
In [5]: My_range_func(10, step = 3, start = 5)
Out[5]: [5, 8]

In [6]: My_range_func(step = 3, 20, start = 5)
  File "<ipython-input-6-ccd9242036e2>", line 1
    My_range_func(step = 3, 20, start = 5)

SyntaxError: positional argument follows keyword argument
```

Arbitrary arguments

There are many cases where the number of arguments that will be passed during a function call is unknown. An example of this case is a function that simply prints every argument it receives,but the number of arguments it is passed is arbitrary. An arbitrary parameter can be specified in the function definition by adding a '*' before the parameter which says that this parameter is a tuple, this tuple will contain the arbitrary number of arguments that will be passed. Lets take a look at an example that illustrates arbitrary arguments in python.

```
1  def sayHiTo(*people):
2     """This function says hi
3     to everyone in the people tuple."""
4
5     # names is a tuple that contains all the arguments
6     for person in people:
7         print("Hi " + person + "!")
8
9  sayHiTo("khalid", "guido von rossum")
10
```

The above program defines a function that takes an arbitrary number of arguments that is available to the function through the "people" tuple. "people" tuple is a tuple that consists of all the arbitrary arguments that are passed to the function when it's called. "Hi <person name>!" is printed for each element of the "people" tuple. The output of the above program is given below.

```
Hi khalid!
Hi guido von rossum!
```

2.5.6 Recursive functions

We already know that functions in python can call other functions, which means there also can be a case where a function calls itself. These kind of functions are called as "recursive functions" as they are defined in terms of themselves. A very common example that is used to explain recursive functions is the factorial program(program that returns the factorial of a given number). The factorial of a number is the product of all integers from one to itself

For example factorial of five represented as "`5!`" is equal to "`1 * 2 * 3 * 4 * 5`". Lets take a look at an example that defines a recursive function to return the factorial of a given number.

```
1  #=========================================================
2  # Python program to calculates the factorial of a number
3  #=========================================================
4
5  def factorial(x):
6      """This is a recursive function
7      that calculates the factorial of
8      an integer"""
```

```
9
10      if x == 1: # recursive calls terminates when this condition is met
11          return 1
12      else:
13          return (x * factorial(x-1)) # The function calls itself
14
15  while 1:
16      num = eval(input("Enter an number: "))
17      if num == -1:
18          break
19      print("The factorial of", num, "is", factorial(num))
20
```

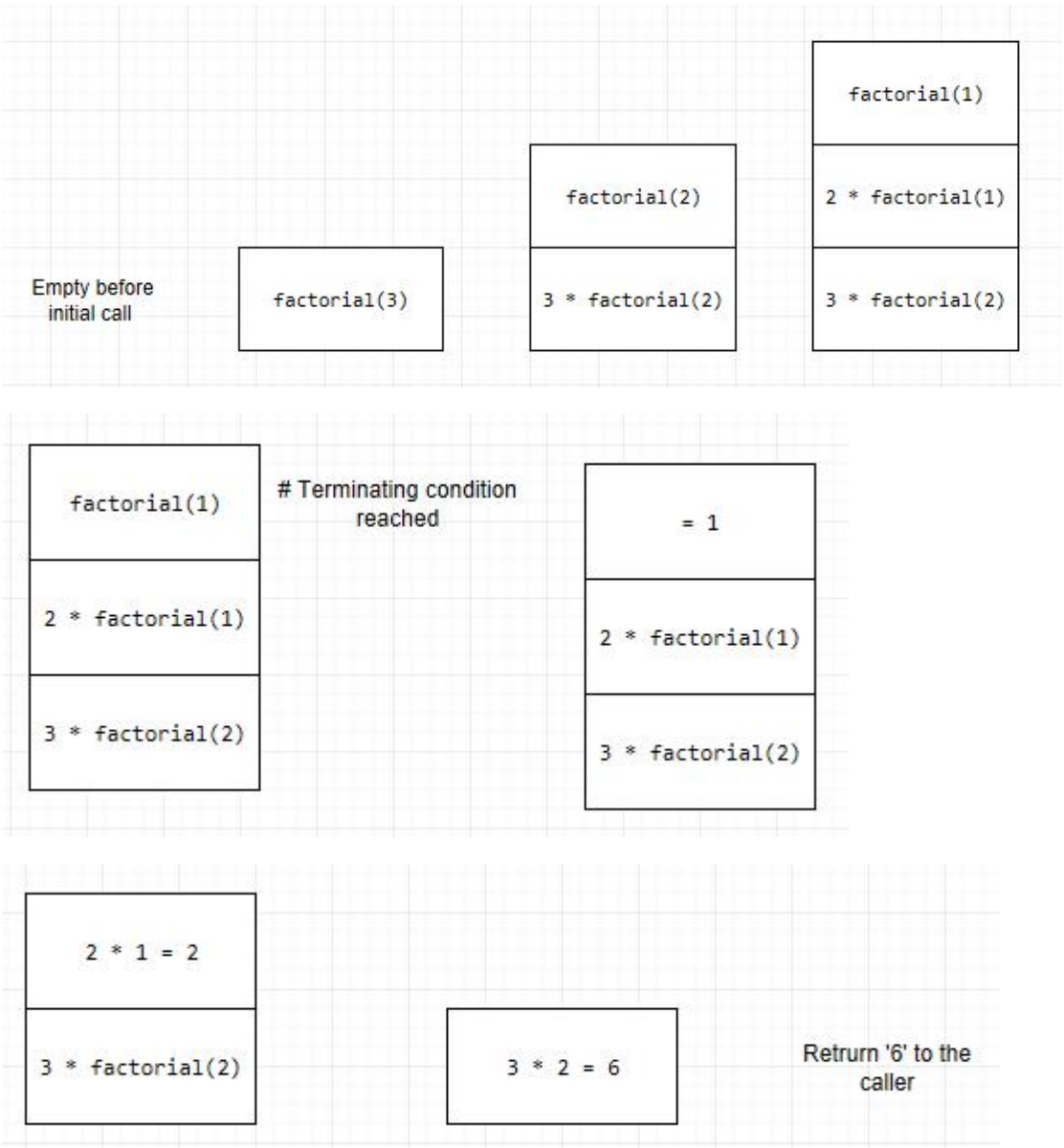
In this example the factorial() function is recursive as it makes a call to itself in the function body. Whenever the factorial function is called, it recursively calls itself decrementing the number passed as an argument each time until the terminating condition is met that is "`x == 1`" The results are continually evaluated from the last call to the first and the result of each function call is returned to it's client caller until the initial call is reached. The a possible output of the above program is given below.

```
Enter an number: 1
The factorial of 1 is 1

Enter an number: 2
The factorial of 2 is 2

Enter an number: 3
The factorial of 3 is 6

Enter an number: 4
The factorial of 4 is 24

Enter an number: 5
The factorial of 5 is 120

Enter an number: -1
```

The following illustration shows the method of evaluation when the `factorial()` function is called with `5` passed as the argument.

```
factorial(4)                # initial call with 4
4 * factorial(3)            # second call with 3
4 * 3 * factorial(2)        # third call with 2
4 * 3 * 2 * factorial(1)    # fourth call with 1
4 * 3 * 2 * 1               # fourth call returns 1
4 * 3 * 2                   # third call returns 2 * 1 = 2
4 * 6                       # second call returns 3 * 2 = 6
24                          # return from initial call 4 * 6 = 24
                            # 24 is the value returned by factorial(4)
```

Recursive function calls can be easily understood in terms of pushing and popping elements from a stack, where each call to a recursive function is pushed to the top of the stack. After the last insertion to the stack has been made the elements of the stack are popped one by one till all the elements have been popped. Before popping an element at the top of the stack the function is evaluated and the if there is a any value returned by that function it is returned to it's caller in the next top of the stack. This order of evaluation continues till there are no more elements in the stack and the if any value is returned by the last element of the stack it is returned to the initial caller of the recursive function. Lets try to visualize this with the function call `factorial(3)`.



Advantages of Recursion:

- The use of recursive functions make the code more elegant and readable.

- Recursive functions can be used to break a complex problem down to simpler problems of a repetitive kind.
- Generating a sequence is easier using recursion than using nested iteration.

Disadvantages of Recursion:

- Sometimes the logic of recursive code can be hard to follow or understand.
- Recursive calls take up more memory than it's iterative version.
- Harder to debug
-

2.5.7 Lambda functions

In this section we'll take a look at what lambda functions are in python and some examples using lambda functions.

Python allows creating functions without a name called lambda functions. Like normal functions are defined using the `def` keyword lambda functions are defined using the `lambda` keyword. Lambda functions are also called as "anonymous functions". The syntax of lambda functions in python are given below.

```
Lambda <arguments>: <Expression>
```

Lambda functions are allowed to have any number of arguments but can have only one expression as it's body. When a lambda function is called the expression is evaluated and returned. Lambda functions are used whenever function objects are needed. Let's take a look at an example that defines and uses a lambda function.

```
1  # Program that defines and uses a lambda function that
2  # returns the cube of number
3
4  cube = lambda a: a * a * a # Defining the lambda function
5
6  num = eval(input("Enter a number: "))
7  print(cube(num))          # Using the lambda function
8
```

In the above code `lambda a: a * a * a` where "a" is the argument and "a * a * a" is the expression that is evaluated and returned. This lambda function has no name that's why its called an anonymous function. The identifier cube returns the function object that is assigned to it, so it can be called a normal function.

The statement

```
cube = lambda a: a * a * a
```

Has the same meaning as the below function definition

```
def cube(a):
```

```
    return a * a * a
```

Lambda functions are usually used when a nameless function is needed for a short time. They are generally used as an argument to functions that take other functions as parameters(higher order functions). Lets go through two examples that show the practical use of lambda functions in python.

Using lambda functions in the `filter()` function

The `filter()` function in python takes a function and a list for parameters. The provided function is called for all the elements of the provided list and a new list is returned with all the elements that returned true when passed to the provided function.

The example below shows the calls the "filter" function to filters out all the zero's from the provided list and returns a new list object with only non zero elements in the provided list.

```
1  #============================================================
2  # Program to filter out zero's from a list
3  #============================================================
4  my_list = [10, 0, 7, 76, 0, 0, 39, 87]
5
6  li = list(filter(lambda a: a != 0 , my_list))
7
8  print(li)
9
```

The above program prints the output given below when executed.

```
[10, 7, 76, 39, 87]
```

Using lambda functions in the `map()` function

pythons map() function takes in a function object and a list object as it's arguments and returns the a list object containing all the values returned by the provided function when it's called on every element in the provided list.

The example code below makes a call to the map() function to return the square of all the elements in the provided list.

```
1  #============================================================
2  # Program to square all the elements of a given list
3  #============================================================
4  my_list = [1, 2, 4, 8, 16, 32, 64, 128]
5
6  li = list(map(lambda x: x * x, my_list))
7
8  print(li)
9
```

The output of this program is shown below.

```
[1, 4, 16, 64, 256, 1024, 4096, 16384]
```

2.5.8 Global, local and non local variables

Lifetime and scope of variables has already been introduces in the previous chapters, now let's take a look at global, local and non local variables in detail.

Global variables

The parts of a python program that exists at the surface level i.e outside of any internal code blocks like "if block", "while block", "function definitions", etc are said to be the global scope and any variables that are created at that level are called global variables. Variables declared in the global scope can be accessed from any part of the program as long as they are alive. In this example, 'x' is declared in the global scope so it's a global variable and can be accessed from any part of the program this ability of global variable x is shown by by defining a function that uses the value of 'x' without it having an 'x' as parameter nor as a variable declared inside the function.

```
1  #----------------------------------------------------Global scope
2  x = 10      # Global variable
3  y = 20      # Global variable
4  z = 30      # Global variable
5  def foo(): # foo's definition, part of the global scope
6  #----------------------------------------------------Global scope
7      #-----------------foo's local scope starts
8      a = 0 # foo's local variable
9      print(a, x, y, z, b) # foo can access global variables
10     #-----------------foo's local scope ends
11
12 #----------------------------------------------------Global scope
13 b = 40      # Global variable
14 foo()       # making a call to function foo
15
16 #----------------------------------------------------Global scope
17
```

This program prints the output below. Note that the function foo can read the global variables.

```
0 10 20 30 40
```

Take a look at an interesting case where the python interpreter raises an error because it misunderstands what is actually meant.

```
1  #----------------------------------------------------------Global scope
2  x = 10      # Global variable
3  def foo():  # foo's definition, part of the global scope
4  #----------------------------------------------------------Global scope
5      #-----------------------------------foo's local scope starts
6      x = x + 40          # Trying to change the value of x ---------!
7      a = 0               # foo's local variable
8      print(a, x)         # foo can access global variables
9      #-----------------------------------foo's local scope ends
10 #----------------------------------------------------------Global scope
11 foo()       # Calling foo
```

```
12 #-------------------------------------------------------Global scope
13
```

When the above code is executed the interpreter raises an error because, in the normal case when an assignment statement is encountered by the interpreter in a local scope, the name on the right hand side of the assignment is considered to be local variable of that local scope. So when the interpreter encounters the statement "`x = x + 40`", it checks the left hand side, creates a new name 'x' in the local scope of the function "foo" then checks the expression on the right hand side of the assignment statement tries to access the non existent object associated to the local variable 'x'. it raises an error because 'x' is referenced in an expression before it's assigned a value. When the above code is executed, it gives the following output.

```
  File "C:/Users/khalid/Documents/ML/Bucca/global scope 2.py", line 7, in foo
    x = x + 40          # Trying to change the value of x ---------!

UnboundLocalError: local variable 'x' referenced before assignment
```

To solve this problem of not being able to modify a global variable, python provides a global variable which will be discussed in the sections that follow.

Local Variables

A variable that is declared inside a local scope is a local variable, a variable local to the scope that it was created in. A local variable exists only in the scope that it was defined in and can not be accessed outside of that scope. Local variable are created just like global variables the only difference being that local variables are declared inside functions or are arguments of a function call and global variables are declared at the global scope of a python program. The example below creates a local variable inside function "foo".

```
1 def foo():  # foo's definition
2     #-----------------------------------foo's local scope starts
3     a = "I'm a local variable in function foo!" # foo's local variable
4     print(a)
5     #-----------------------------------foo's local scope ends
6
7 foo()
8
```

The output prints the output given below.

```
Im a local variable in function foo!
```

In the example below the variable 'a' that is local to the function "foo" is accessed outside of the scope of the variable which results in an error.

```
1 def foo():  # foo's definition
2     #-----------------------------------foo's local scope starts
3     a = 0               # foo's local variable
4     print(a)
```

```
    #-----------------------------------foo's local scope ends

#-------------------------------------------------Global scope

print(a) #-------------------------> a is not defined in this scope!

#-------------------------------------------------Global scope

```

The following error is raised when the above program is executed as the variable 'a' is defined in the local scope of function "foo" and is available for use only there, it can't be accessed by the upper levels. This program raises the error shown below.

```
  File "C:/Users/khalid/Documents/ML/Bucca/local scope1.py", line 9, in <module>
    print(a) #-------------------------> a is not defined in this scope!

NameError: name 'a' is not defined
```

Global and local variables with the same name

When a variable is declared in the local scope of a function that has the same name as a variable in the global scope, which variable gets accessed? In such cases the python prefers the local variable over the global variable, so when a variable name in the local scope is accessed that has the same name as a variable in the global scope the object associated to that name in the local scope is returned. The code example below shows an instance of this case.

```
#-----------------------------------------------------Global scope
a = 10          # Global variable
def foo():       # foo's definition, part of the global scope
#-----------------------------------------------------Global scope
    #-----------------------------------foo's local scope starts
    a = 0       # foo's local variable
    return a     # The local variable a is accessed (a = 0)
    #-----------------------------------foo's local scope ends

#-----------------------------------------------------Global scope
print("The value of 'a' in foo is ",foo())
print("The value of 'a' in global scope is ", a)
#-----------------------------------------------------Global scope

```

In the above code, the same name 'a' is used both in the global scope as well as in the local scope of function "foo". When the values of both variables are printed we get different values as the local variables preferred over global if there is a name match

```
The value of 'a' in foo is  0
The value of 'a' in global scope is  10
```

Lets look back at the previous example where we try to change the value of a global variable inside a function. This causes an error to be raised by the python interpreter, and it's already been mentioned that the solution python provides for this is the "global" keyword

In Python the `global` keyword allows modification variables that are outside the current scope. The can be used to create global variables and modify it in the local context.

Here are some things to remember while using the `global` keyword.

1. A variable is by default a local to a function when it is declared inside the function
2. A variable is by default a global variable when its declared outside a function. It makes no difference using the global keyword on variables defined outside functions.
3. The global keyword can be used inside functions to read and write global variables inside functions.

Lets take a look at some examples that illustrate the use and working of the global keyword.

```
x = 20       # x is a global variable

def printx():
    print(x)  # Since there is no local variable x, the global variable x
              # can be accessed

printx()

```

This example code produces the below output.

```
20
```

Here, notice that as the function `printx()` doesn't have a local variable 'x', it can read the value of the global variable 'x', but the interpreter raises an error when we try to write to the same variable.

```
x = 20 # x is a global variable

def printx():
    x = x + 4  # try using the value of the global variable
    print(x)

printx()

```

When the above program is ran it raises the error below,because the value of a global variable is being modified in the function.without using the global keyword it is only possible to read the value of a global variable inside a function

```
  File "C:/Users/khalid/Documents/ML/Bucca/global2.py", line 4, in printx
    x = x + 4

UnboundLocalError: local variable 'x' referenced before assignment
```

The `global` keyword can be used to overcome this kind of a problem where the modifications to global variables from within a function is required. The example below shows how a global keyword can be used to modify a global variable from within a function. To modify a global variable from inside a function this variable has to first to be declared as `global` inside the function it needs to be used in. This is illustrated in the example below.

```
1  x = 20        # x is a global variable
2
3  def printx():
4      global x   # Now the global variable x can be modified
5      x = x + 4  # No error raised
6      print("This is the value of x inside printx: %i" %x)
7
8  printx()
9  print("This is the value of x outside printx: %i" %x)
10
```

The output of the above program is given below. notice that the value of the variable 'x' is the same inside and outside the `printx()` function. This is because variable 'x' inside the `printx()` function now refers to the same object as the variable 'x' outside this function.

```
This is the value of x inside printx: 24
This is the value of x outside printx: 24
```

The `global` keyword can also be used in nested functions, let's take a look at an example of using the `global` keyword in a nested function.

```
1  a = 10
2  print("a in the global scope before calling foo: ", a)
3
4  def foo():
5      a = 40
6      print("a in local scope before calling nested_foo: ", a)
7
8      def nested_foo():
9          global a
10         a = 20
11
12     nested_foo()
13     print("a in local scope after calling nested_foo: ", a)
14
15 foo()
16 print("a in the global scope before calling foo: ", a)
17
```

When the global keyword is used on the variable 'a' in the `nested_foo()` function, any change that this function makes to the variable 'a' in it's scope affects the variable at the global scope. The output of the above program is given below.

```
a in the global scope before calling foo:  10
a in local scope before calling nested_foo:  40
a in local scope after calling nested_foo:  40
a in the global scope before calling foo:  20
```

Non local variables

Non local variables are used in nested functions where the local scope is not defined, so the variable can't be in either the local nor the global scope.

The keyword `nonlocal` is used to declare a non local variable in python.let's take a look at an example that explains non local variables in python.

```
1  def foo():
2      #-----------------------------------foo's local scope starts
3      a = 20          # foo's local variable
4      print("a in foo: ", a)
5      #-----------------------------------foo's local scope ends
6      def nested_foo():
7          nonlocal a   # Now 'a' is a non local variable
8          a = a + 20
9          print("a in nested_foo: ", a)
10     nested_foo()
11     print("a in foo: ", a)
12 #-------------------------------------------------Global scope
   foo()
   #-------------------------------------------------Global scope
```

The output of the above program is given below.

```
a in foo:  20
a in nested_foo:  40
a in foo:  40
```

Here the function `nested_foo()` is defined in the local scope of the function `foo()`. The `nonlocal` keyword is used to declare 'a' to be a non local variable. Observe that when a nonlocal variable is changed, this change also reflects on the local variable 'a' of function `foo()`. This is because the variable 'a' which is in the local scope of function `foo()` is now changed by function `nested_foo()`.