Function in Python

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for our application and a high degree of code reusing.
Python gives many **built-in functions** like print(), etc.

Most programs perform tasks that are large enough to be broken down into subtasks.
Because of this, programmers often organize their programs into smaller, more manageable chunks by writing their own functions
Instead of writing one large set of statements we can break down a program into several small functions, allowing us to “divide and conquer” a programming problem.
Thus we can create our own functions. These functions are called **user-defined functions.**

Defining a Function

Functions, like variables, must be named and created before we can use them
The same naming rules apply for both variables and functions

- We can’t use any of Python’s keywords
- No spaces
- The first character must be A-Z or a-z or the “_” character
- After the first character, you can use A-Z, a-z, “_” or 0-9
- Uppercase and lowercase characters are distinct
- You can define functions to provide the required functionality. Here are simple rules to define a function in Python.
- Function blocks begin with the keyword **def** followed by the **function name** and list of **arguments within parentheses** (**( )**) followed by a **colon**(**:**).
- Any input parameters or arguments should be placed within these parentheses.
- we can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation
- string of the function or docstring.
- The code block within every function starts with a colon (:) and is indented.
- The statement return [expression] exits a function, optionally passing back an
- expression to the caller. A return statement with no arguments is the same as a return None.

Syntax

```
def function_name( parameters ):
    Statements
    return [expression]
```

By default, parameters have a positional behaviour and you need to inform them in the the same order that they were defined.

Example

The following function takes a string as an input parameter and prints it on the standard screen.

```
def printme( str ):
    "This prints a passed string into this function"
     print (str)
    return
```

Calling a Function

Defining a function gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is an example to call the function-

```
# Function definition is here
def myfunction():
    print ("Printed from inside a function")

# call the function
myfunction()
-------------------------------------------------
#!/usr/bin/python3
# Function definition is here
def printme( str ):
    "This prints a passed string into this function"
     print (str)
    return

# Now you can call printme function
printme("This is first call to the user defined function!")
printme("Again second call to the same function")
```

When the above code is executed, it produces the following result-

This is first call to the user defined function!
Again second call to the same function

- When you run a function you say that you “call” it
- When a function is called programmers commonly say that the “control” of the program has been transferred to the function. The function is responsible for the program’s execution.
- Once a function has completed, Python will return back to the line directly after the initial function call
- Functions must be defined before they can be used. In Python we generally place all of our functions at the beginning of our programs.
- All functions in Python have a return value, even if no return line inside the code
- Functions without a return statement return the special value None
- None is a special constant in the language; None is used like NULL, void, or nil in other languages
- None is also logically equivalent to False. The interpreter’s REPL doesn’t print None

Using multiple Function

```
def add(a,b):
    c=a+b
    return c

def sub(a,b):
    c=a-b
    return c

[m,n]=map(int,input('Enter two number').split())
result=add(m,n)
print('Sum of ',m,' and ',n, ' is ',result);
result=sub(m,n)
print('Difference of ',m,' and ',n, ' is ',result);
```

Calling Function within a Function

```
def compute(a):
    b=int(input('Enter 2nd Number :'))
```

```
    result=add(a,b)
    print('Result within Function ',result)
    return result


def add(m,n):
    p=m+n
    return p

# main function call
a=int(input('Enter a number : '))
value=compute(a)
print('Result is ',value)
```

Nested Function

```
# print nth fibonacci number
def fibonacci(n):
     def step(a,b):
         print(a,end=' ')
         return b, a+b
   a,b=0,1
   for i in range(n):
       a, b = step(a, b)
     #return a


p=fibonacci(10)
print(p)
```

Returning multiple value

```
def foo(x=1,y=2,z=3):
    return(2*x,4*y,8*z) #

a,b,c=foo(z=12,x=7)
print(a,b,c)
```

Local Variables

- Functions are like “mini programs”. We can create variables inside functions just like we can do within the main program.
- However, variables that are defined inside of a function are considered “local” to that function.
- This means that they only exist within that function. Objects outside the “scope” of the function will not be able to access that variable.
- Different functions can have their own local variables that use the same variable name
- These local variables will not overwrite one another since they exist in different “scopes”

```
def firstfunction(a):
  a = 1000
  print ("firstfunction: Value of variable is = ", a)
  return

def secondfunction(b):
   b= 2000
   print ("secondfunction value of variale = ",b)
   return

n=int(input('Enter a Integer number : '))
print('Value of Variable in main Function (before call) =',n)
firstfunction(n)
print('Value of Variable in main Function (after 1st call)=',a)
secondfunction(n)
print('Value of Variable in main Function (after 2nd call)=',n)
```

Output:

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

Pass by Reference vs Value

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. For example-

```
#!/usr/bin/python3
# Function definition is here
def changeme( mylist ):
        "This changes a passed list into this function"
        print ("Values inside the function before change: ", mylist)
        mylist[2]=50
        print ("Values inside the function after change: ", mylist)
        return

# Now you can call changeme function
mylist = [10,20,30]
changeme( mylist )
print ("Values outside the function: ", mylist)
```

Here, we are maintaining reference of the passed object and appending values in the same object. Therefore, this would produce the following result-

Values inside the function before change:[10, 20, 30]
Values inside the function after change:[10, 20, 50]
Values outside the function:[10, 20, 50]

There is one more example where argument is being passed by reference and the reference is being overwritten inside the called function. Here list name is considered a local variable as a whole. If it refers element-wise it will be a call by reference.

```
#!/usr/bin/python3
# Function definition is here
def changeme( mylist ):
        "This changes a passed list into this function"
        mylist = [1,2,3,4] # This would assign new reference in mylist
        print ("Values inside the function: ", mylist)
        return
```

```
# Now you can call changeme function
mylist = [10,20,30]
changeme( mylist )
print ("Values outside the function: ", mylist)
```

The parameter mylist is local to the function changeme. Changing mylist within the function does not affect mylist. The function accomplishes nothing and finally this would produce the following result-

Values inside the function: [1, 2, 3, 4]
Values outside the function: [10, 20, 30]

Default Arguments

Default arguments are those that take a default value if no argument value is passed during the function call. You can assign this default value by with the assignment operator =, just like in the following example:

```
# Define `plus()` function
def plus(a,b = 2):
  return a + b

# Call `plus()` with only `a` parameter
c=plus(a=1)
print('Sum =',c)
# Call `plus()` with `a` and `b` parameters
c=plus(a=1, b=3)
print('Sum =',c)
```

Output:
Sum = 3
Sum = 4

Required Arguments

As the name kind of gives away, the required arguments of a UDF are those that have to be in there. These arguments need to be passed during the function call and in precisely the right order, just like in the following example:

```
# Define `plus()` with required arguments
def plus(a,b):
  return a + b


c=plus(5,6)
print('Sum= ',c)
```

```
Output: Sum=  11
```

You need arguments that map to the **a** as well as the **b** parameters to call the function without getting any errors.

Keyword Arguments

If you want to make sure that you call all the parameters in the right order, you can use the keyword arguments in your function call. You use these to identify the arguments by their parameter name. Let's take the example from above to make this a bit more clear:

```
# Define `plus()` function
def plus(a,b):
  return a - b

# Call `plus()` function with parameters
c=plus(10,6)
print('Result is ',c)
# Call `plus()` function with keyword arguments
c=plus(b=10, a=6)
print('Result is ',c)
```

Output:

Result is 4
Result is -4

Note that by using the keyword arguments, you can also switch around the order of the parameters and still get the same result when you execute your function:

Variable Number of Arguments

In cases where you don't know the exact number of arguments that you want to pass to a function, you can use the following syntax with *args:

```
# Define `plus()` function to accept a variable number of arguments
def plus(*args):
   su=0
   for abc in args:
      su=su+abc
   return su

# Calculate the sum
a=plus(1,4,5,7)
print(a)
```

Output: 17

The asterisk (`*`) is placed before the variable name that holds the values of all nonkeyword variable arguments. Note here that you might as well have passed `*varint`, `*var_int_args` or any other name to the `plus()` function.

Tip: try replacing `*args` with another name that includes the asterisk. You'll see that the above code keeps working!

You see that the above function makes use of the built-in Python `sum()` function to sum all the arguments that get passed to `plus()`. If you would like to avoid this and build the function entirely yourself, you can use this alternative:

```
# Define `plus()` function to accept a variable number of arguments
def plus(*args):
  total = 0
  for i in args:
    total += i
  return total

# Calculate the sum
plus(20,30,40,50)
```

Output: 140

Global vs Local Variables

In general, variables that are defined inside a function body have a local scope, and those defined outside have a global scope. That means that local variables are defined within a function block and can only be accessed inside that function, while global variables can be obtained by all functions that might be in your script:

```
#!/usr/bin/python3
total = 0 # This is global variable.
# Function definition is here
def sum( arg1, arg2 ):
    # Add both the parameters and return them."
    total = arg1 + arg2; # Here total is local variable.
    print ("Inside the function local total : ", total)
    return total
# Now you can call sum function
sum( 10, 20 )
print ("Outside the function global total : ", total )
```

When the above code is executed, it produces the following result-

Inside the function local total : 30

Outside the function global total : 0

Recursive functions

A recursive function is a function that calls itself in its definition. For example the mathematical function, factorial, defined by factorial(n) = n*(n-1)*(n-2)*...*3*2*1 . can be programmed as

```
def factorial(n):
    if n==0:
        return 1
    else:
        return n*factorial(n-1)


c=factorial(5)
print(c)
```

the outputs here are:
factorial(0)
#out 1
factorial(1)
#out 1
factorial(2)
#out 2
factorial(3)
#out 6

Recursion limit

There is a limit to the depth of possible recursion, which depends on the Python implementation. When the limit is reached, a RuntimeError exception is raised
It is possible to change the recursion depth limit by using ***sys.setrecursionlimit(limit)*** and check this limit by ***sys.getrecursionlimit(int numer)*** .

The Anonymous Functions

- These functions are called anonymous because they are not declared in the standard manner by using the def keyword. You can use the lambda keyword to create small anonymous functions.
- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression.
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
- Although it appears that lambdas are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is to stack allocation by passing function, during invocation for performance reasons.

Syntax

The syntax of lambda function contains only a single statement, which is as follows-

lambda [arg1 [,arg2,.....argn]] : expression

Following is an example to show how lambda form of function works-

```
#!/usr/bin/python3
# Function definition is here
sum = lambda arg1, arg2: arg1 + arg2
# Now you can call sum as a function
print ("Value of total : ", sum( 10, 20 ))
print ("Value of total : ", sum( 20, 20 ))
```