Week 9- Functions

Definition:

A function is named block of code that performs a specific task.

Types of functions

There are two types of function in Python programming:

Standard library functions - These are built-in functions in Python that are available to use.

User-defined functions - We can create our own functions based on our requirements.

Need of user-defined functions:

- User-defined functions help to decompose a large program into small segments which makes program easy to understand, maintain and debug.
- If repeated code occurs in a program. Function can be used to include those codes and execute when needed by calling that function.
- Programmers working on large project can divide the workload by making different functions.

How to define a function??

Function Definition:

The usual syntax for defining a Python function is as follows:

```
def <function_name>([<parameters>]):
    <statement(s)>
```

Where,

def :-	The keyword that informs Python that a function is being defined
<function_name>	A valid Python identifier that names the function
<parameters>	An optional, comma-separated list of parameters that may be passed to the function

:	Punctuation that denotes the end of the Python function header (the name and parameter list)
<statement(s)>	A block of valid Python statements

The syntax for calling a Python function is as follows:

<function_name>([<arguments>])

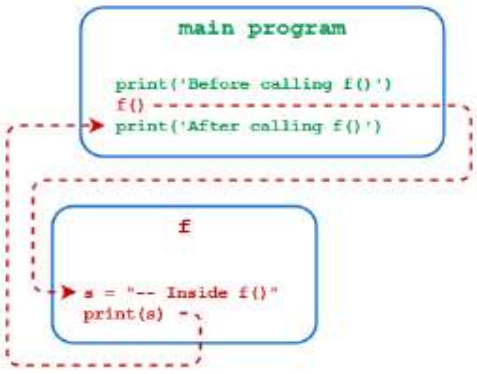
Where,

<arguments> are the values passed into the function. They correspond to the <parameters> in the Python function definition.

Example:

```
1  def f():
2      s = '-- Inside f()'
3      print(s)
4
5  print('Before calling f()')
6  f()
7  print('After calling f()')
```

The sequence of execution (or control flow) for the above program is shown in the figure



Example:

Program to find the sum of two numbers using functions:

```
def add(num1 num2):
    sum= num1 + num2
    print(f "The sum of {num1} and {num2} is {sum}")
# Driver code
add(5,6)
add(8.9,5.6)
add("gpt", " athani")
```

Output:

```
The sum of 5 and 6 is 11
The sum of 8.9 and 5.6 is 14.5
The sum of gpt and  athani is gpt athani
```

** Program to create a function OddEven() to check whether given number is odd or even.**

```
def OddEven(number):
    if (number% 2 == 0):
        print(f"{number} is even")
    else:
        print(f"{number} is odd")
# Driver code to call the function
OddEven(2)
OddEven(3)
```

Output:

```
2 is even
3 is odd
```

** Program to create a function CheckArmstrong() to check whether given number is armstrong or not.**

```
def CheckArmstrong(number):
    sum=0
    temp=number
    while temp!=0:
        rem=temp%10
```

```
153 is Armstrong
916 is not a Armstrong
```

```
        sum=sum+rem**3
        temp=temp//10
  if number==sum:
         print(f"{number} is Armstrong")
  else:
         print(f"{number} is not a Armstrong")

# Driver code to call the function
CheckArmstrong(153)
CheckArmstrong(916)
```

Function Arguments

You can call a function by using the following types of formal arguments −

1) Required arguments
2) Variable-length arguments
3) Keyword arguments
4) Default arguments

Required arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

Example:

```
# Function definition is here
def printme( str ):
   print(str)
# Now you can call printme function
```

```
Traceback (most recent call last):
  File "test.py", line 11, in <module>
    printme();
TypeError: printme() takes exactly 1 argument (0 given)
```

printme()

Arbitrary Arguments (Variable-length arguments):

It is also possible to define a function so that any number of arguments can be passed to it. This is done by adding * before the argument name in the function definition. This way the function will receive a tuple of arguments, and can access the items accordingly.

Example:

1)
```
def func(*args):
    for x in args:
        print(x)
func(1)
func(4,5)
```

Output:

1

4

5

2)
```
def sum(*args):
    sum=0
    for x in args:
        sum=sum+x
    return x
print(" sum is:",sum(1,2,3))
print("sum is:", sum(1,2,3,4,5))
```

Output:

Sum is: 6

Sum is: 15

Keyword Arguments

- Arguments can also be sent to the function with the key = value syntax. The arguments sent this way are called keyword arguments.
- In this case, the order of arguments does not need to be remembered.
- Example:

1)
```
def func(a, b):
    print(a)
    print(b)
func(b='world', a='hello')
```

Output:

hello

world

2)
```
def display(fname,lname):
    print("My first name is" +fname)
    print("My surname is" +lname)
dispaly(lname="shinge", fname="laxmi")
```

Output:

My first name is laxmi

My surname is shinge

Arbitrary Keyword Arguments

- It is possible to define a function so that any number of keyword arguments can be passed to it.
- This is done by adding ** before the argument name in the function definition.
- This way the function will receive a dictionary of arguments, and can access the items accordingly.
- Example:

```
def func(**args):
  print(args['brand'])
func(brand='Ford', model='Mustang')
```

Output:

Ford

Default Arguments

- A function with an argument can be defined so that it automatically gives some default value to the argument if none is passed to it.
- Example:

1)
```
def work_area(prompt, domain="Data Analytics"):
    print(f"{prompt} {domain}")
```

Output:

Sam works in Data Analytics

Ram has interest in IOT

```
work_area("Sam works in")
work_area("Ram has interest in", "IOT")
```

2).
```
def func(a=1):
    print("a=",a)
func(2)
func()
```

Output:

a=2

a=1

return statement:

- Most of the times you may want the function to perform its specified task to calculate a value and return the value to the calling function so that it can be stored in a variable and used later.
- This can be achieved using the optional return statement in the function definition.
- Syntax: return [expression_list]
  - ☞ If an expression list is present, it is evaluated and returned
  - ☞ Otherwise None is returned.
- The return statement leaves the current function definition with the expression_list (or None) as a return value.
- The return statement terminates the execution of the function definition in which it appears and returns control to the calling function.
- Example:

```
# Function definition is here
def sum( arg1, arg2 ):
  # Add both the parameters and return them."
```

```
    total = arg1 + arg2
    return total
# Now you can call sum function
Sum=sum( 10, 20 )
print("Sum is :", Sum)
```

Output:

Sum is: 30

GENERATOR FUNCTIONS : Use of yield keyword

- In Python, a generator is a function that returns an iterator which produces a sequence of values when iterated over it.
- Generators are useful when we want to produce a large sequence of values, but we don't want to store all of them in memory at once.
- We can define a generator function using the def keyword, but instead of the return statement we use the yield statement.
- Syntax:

```
def generator_name(arg):
    # statements
    yield something
```

  Where, yield keyword is used to produce a value from the generator.
- The function will execute till the first yield statement when next() is called on the object, and pause. It will execute again till the next yield statement if next() is called again on the object and pause again, and so on
- Examples:

1)

```
def func(n):
    yield n
```

Output:

<class 'generator'>

2

4

```
    yield n*n
obj = func(2) # obj is a generator object
print(type(obj))
print(next(obj))
print(next(obj))
```

2)
```
def func():
    yield 1
    yield 2
    yield 3
for x in func(): # generator objects are iterable
    print(x)
```

Output:
1
2
3

3)
```
def func(n):
    yield n**2
    yield n**3
for x in func(3): # generator objects are iterable
    print(x)
```

Output:
9
81

4)
```
def my_generator(n):
    # initialize counter
    value = 0
    # loop until counter is less than n
    while value < n:
        # produce the current value of the counter
```

Output:
0
1
2

```
        yield value
        # increment the counter
        value += 1
# iterate over the generator object produced by my_generator
for value in my_generator(3):
    # print each value produced by generator
    print(value)
```

SCOPE OF VARIABLES

- All variables in a program may not be accessible at all locations in that program.
- This depends on where you have declared a variable.
- The scope of a variable is the region of code in which a defined variable is accessible.
- There are two basic scopes of variables in Python –
  1. Local variables
  2. Global variables

Local Variables:

- The variables that are defined inside a function are called local variables.
- These variables cannot be accessed outside of the function.
- Example:

```
def func():
    a = 5
print(a)
```

☞ The above code will throw an error because a is a local variable defined inside of func(). It is undefined outside of the definition of func(), and hence it will be as if we are calling a variable that was never even created.

Global Variables:

- The variables that are created outside the function definition are called global variables
- They can be accessed from within the function.
- Example:

```
a = 5
def func():
    print(a)
func()
```

Note:

☞ If a local variable in a function definition has the same name as a global variable defined outside of the function, then for the scope of the function definition, the name will refer to the local variable. The global variable won't be modified. Example:

```
a = 10
def func():
        a = 5
        print("Inside function a=",a)
print("Outsidefunction a=",a )
```

Output:

```
Inside function a=5
Outsidefunction a=10
```

☞ Although by default, a variable created inside a function definition will be a local variable, it is also possible to define a variable with global scope inside a function using the global keyword.

Example:

```
def func():
   global a  # local variable a is declared as global
    a = 5
func()
print(a)
```

Anonymous Functions: lambda keyword

- A lambda function or an anonymous function is a short hand way of defining a function in python.
- A lambda function can take any number of arguments, but can only have one expression.
- The general syntax for a lambda function is given below:

```
lambda arguments : expression
```

- Lambda functions create objects (of class function) that act as functions.

Examples:

1)

```
x = lambda a : a + 10
print(type(x))
print(x(5))
```

```
Output:
<class 'function'>
15
```

2)

```
x = lambda a, b: a + b
print(x(7,8))
```

```
Output:
15
```

- Since lambda functions are objects, they can even be returned by a function. So it will be like the return value of a function in another function.

  Example:

```
def func(n):
 return lambda a : a*n
double = func(2)
triple = func(3)
print(double(5)) # will multiply 5 by 2 and print
print(triple(5)) # will multiply 5 by 3 and print
```

Output:

10

15

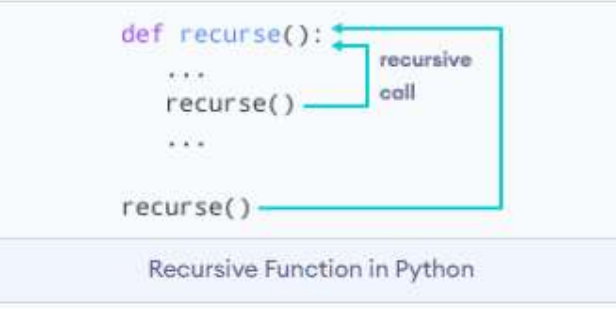
Recursion:

Recursion is the process of defining something in terms of itself**.**

Python Recursive Function

In Python, we know that a function can call other functions. It is even possible for the function to call itself. A function which has call to itself is known as recursive function.

The following figure shows how recursive function works



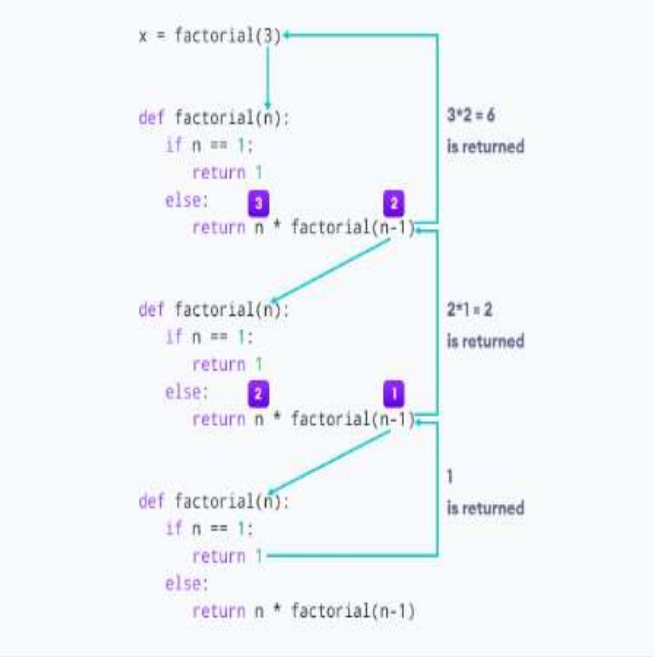
Recursive Function in Python

Example: Program to find the factorial of a number using recursion

```
def factorial(x):
```

```
    if x == 0:
        return 1
    else:
        return (x * factorial(x-1))
num = 3
print("The factorial of", num, "is", factorial(num))
```

The working of recursive factorial function is as shown in the figure:



Working of a recursive factorial function