Unit - 5
Function - Python

Introduction

- In Python, a function is a group of related statements that performs a specific task.
- Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organised and manageable.
- Furthermore, it avoids repetition and makes the code reusable.
- The idea is to put some commonly or repeatedly done tasks together and make a function so that instead of writing the same code again and again for different inputs, we can do the function calls to reuse code contained in it over and over again.

There are mainly two types of functions.

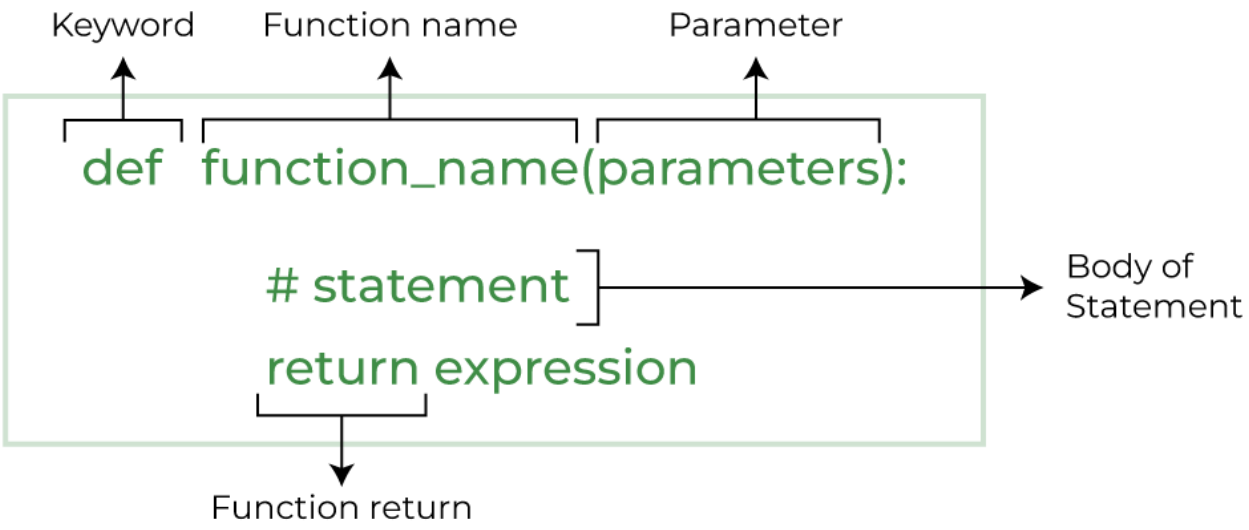
1.User-defined functions - The user-defined functions are those define by
the user to perform the specific task.
2.Built-in functions - The built-in functions are those functions that are predefined in Python.

Advantage of Functions in Python

There are the following advantages of Python functions.

- Using functions, we can avoid rewriting the same logic/code again and again in a
- program.
- We can call Python functions multiple times in a program and anywhere in a
- program.
- We can track a large Python program easily when it is divided into multiple functions.
- Reusability is the main achievement of Python functions.
- However, Function calling is always overhead in a Python program.

Syntax: Python Functions



Above shown is a function definition that consists of the following components.

- Keyword def that marks the start of the function header.
- A function name to uniquely identify the function. Function naming follows the same rules of writing identifiers in Python.
- Parameters (arguments) through which we pass values to a function. They are optional.
- A colon (:) to mark the end of the function header.
- Optional documentation string (docstring) to describe what the function does.
- One or more valid python statements that make up the function body. Statements must have the same indentation level (usually 4 spaces).
- An optional return statement to return a value from the function.

Creating Function

Python provides the def keyword to define the function. The syntax of the define function is given below

Syntax →

```
def function_name(parameters):
    """docstring"""
    statement(s)
```

Function Calling →

In Python, after the function is created, we can call it from another function. A function must be defined before the function call; otherwise, the Python interpreter gives an error. To call the function, use the function name followed by the parentheses.

```
# Create Function that name HEllo
def Hello():
    print("Hello CSE")

# We Call the Function
Hello()
```

Output →

```
Hello CSE
```

The return statement→

The return statement is used to exit a function and go back to the place from where it was called.

Syntax of return→

```
return [expression_list]
```

This statement can contain an expression that gets evaluated and the value is returned. If there is no expression in the statement or the return statement itself is not present inside a function, then the function will return the None object.

```
# Create Function that name fun
def fun():
    return "Final Year CSE"
# Call Function and store in a variable
n=fun()
# Print a variable
print(n)
```

Output→

```
PS C:\Users\Verma Ji\Desktop\rjgp\rjgp> python -u "c:\Users\Verma Ji\Desktop\rjgp\rjgp\main2.py"
Final Year CSE
```

Arguments in function

The arguments are types of information which can be passed into the function. The arguments are specified in the parentheses. We can pass any number of arguments,but they must be separated with a comma.

```
# Create Function that name fun
def sum(a,b):
    return a+b
# Call Function and store in a print statement
print("The Sum of Two Number " ,sum(5,6))
```

Output→

```
PS C:\Users\Verma Ji\Desktop\rjgp\rjgp> python -u "c:\Users\Verma Ji\Desktop\rjgp\rjgp\sum.py"
The Sum of Two Number  11
```

Call by reference in Python

In Python, call by reference means passing the actual value as an argument in the function. All the functions are called by reference, i.e., all the changes made to the reference inside the function revert back to the original value referred by the reference.

```
#defining the function
def change_list(list1):
 list1.append(20)
 list1.append(30)
 print("list inside function = ",list1)
#defining the list
list1 = [10,30,40,50]
#calling the function
```

```
change_list(list1)
print("list outside function = ",list1)
```

Output →

```
list inside function =  [10, 30, 40, 50, 20, 30]
list outside function =  [10, 30, 40, 50, 20, 30]
```

Types of arguments

There may be several types of arguments which can be passed at the time of function call.

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

1. Required arguments →A default argument is a parameter that assumes a default value if a value is not provided in the function call for that argument. The following example illustrates Default arguments.

Example 1 →

```
# the function simple_interest accepts three arguments and
returns the simple interest accordingly
def simple_interest(p,t,r):
    return (p*t*r)/100
p = float(input("Enter the principle amount? "))
r = float(input("Enter the rate of interest? "))
t = float(input("Enter the time in years? "))
print("Simple Interest: ",simple_interest(p,r,t))
```

Output→

```
Enter the principle amount? 5000
Enter the rate of interest? 10
Enter the time in years? 12
Simple Interest:  6000.0
```

Example 2→

```
# Taking kilometers input from the user
kilometers = float(input("Enter value in
kilometers: "))
def convert(kilometers):
    # conversion factor
    conv_fac = 0.621371
    # calculate miles
    miles = kilometers * conv_fac
    return miles
print(f'{kilometers} kilometers is equal to
{convert(kilometers)} miles')
```

Output→

```
Enter value in kilometers: 15
15.00 kilometers is equal to 9.32 miles
```

Example 3 →

```
def sum(a,b):
    return a+b
print(sum(1,2))
```

Output→

```
3
PS C:\Users\Verma Ji\Desktop\rjgp\rjgp>
```

Default Arguments →

- A default argument is a parameter that assumes a default value if a value is not provided in the function call for that argument. The following example illustrates Default arguments.
- Python allows us to initialise the arguments at the function definition. If the value of any of the arguments is not provided at the time of function call, then that argument can be initialised with the value given in the definition even if the argument is not specified at the function call.

Example 1 →

```
def printme(name,age=19):
    print("My name is",name,"and age is",age)
printme(name="Priyanshu")
```

Output→

```
My name is Priyanshu and age is 19
```

Example 2 →

```
# Taking kilometers input from the user
kilometers = float(input("Enter value in
kilometers: "))
def convert(kilometers,conv_fac = 0.621371):
    # calculate miles
    miles = kilometers * conv_fac
    return miles
print(f'{kilometers} kilometers is equal to
{convert(kilometers)} miles')
```

Output→

```
Enter value in kilometers: 15
15.00 kilometers is equal to 9.32 miles
```

Variable-length Arguments (*args)

In large projects, sometimes we may not know the number of arguments to be passed in advance. In such cases, Python provides us the flexibility to offer the comma-separated values which are internally treated as tuples at the function call. By using the variable-length arguments, we can pass any number of arguments.

Code →

```
def printme(*names):
    print("printing the passed arguments...")
    for name in names:
        print(name)
printme("Mechanical","CSE","ECE","FD")
```

Output→

```
Mechanical
CSE
ECE
FD
```

Keyword arguments(**kwargs)

Python allows us to call the function with the keyword arguments. This kind of function call will enable us to pass the arguments in the random order. The name of the arguments is treated as the keywords and matched in the function
calling and definition. If the same match is found, the values of the arguments are copied in the function definition.

Code →

```
def func(name,message):
    print("printing the message with",name,"and
",message)
#name and message is copied with the values
John and hello respectively
```

```
func(name = "CSE ",message="hello")
```

Output →

```
printing the message with CSE and  hello
```

Scope of variables

The scopes of the variables depend upon the location where the variable is being declared. The variable declared in one part of the program may not be accessible to the other parts.

In python, the variables are defined with the two types of scopes.
1. Global variables
2. Local variables

The variable defined outside any function is known to have a global scope, whereas the variable defined inside a function is known to have a local scope.

Global Variables

In Python, a variable declared outside of the function or in global scope is known as a global variable. This means that a global variable can be accessed inside or outside of the function.

Let's see an example of how a global variable is created in Python.

Example →

```
x = "global"

def foo():
    print("x inside:", x)


foo()
print("x outside:", x)
```

Output →

```
x inside: global
x outside: global
```

Local Variables

A variable declared inside the function's body or in the local scope is known as a local variable.

Code →

```
def foo():
    y = "local"
    print(y)

foo()
```

Output →

```
local
```

Example 1 : Using Global and Local variables in the same code

```
x = "global "

def foo():
    global x
    y = "local"
    x = x * 2
    print(x)
    print(y)

foo()
```

Output →

```
global global
local
```

Example 2: Global variable and Local variable with same name

```
x = 5

def foo():
    x = 10
    print("local x:", x)


foo()
print("global x:", x)
```

Output→

```
local x: 10
global x: 5
```

Functions as First-Class Objects

In Python, they function as First-Class Objects. Because it treats the same as the object, and it has the same properties and method as an object. A function can be assigned to a variable, pass them as an argument, store them in data structures and return a value from other functions. It can be manipulated, such as other objects in Python.

Properties of First-Class functions

1. Functions can be assigned to a variable
2. A function is an example of an object type.
3. We also return the function from a function.

4. Functions have the same properties and methods as objects
5. The function is treated as an object to pass as an argument to another function. Create a program to understand Python functions as an object.

Example –>

```
# Python program to illustrate functions
# can be treated as objects
def shout(text):
    return text.upper()

print (shout('Hello'))

yell = shout

print (yell('Hello'))
```

Output →

```
HELLO
HELLO
```

Example 2 →

```
# Python program to illustrate functions
# can be passed as arguments to other functions
def shout(text):
    return text.upper()

def whisper(text):
    return text.lower()
```

```
def greet(func):
    # storing the function in a variable
    greeting = func("""Hi, I am created by a function
                    passed as an argument.""")
    print (greeting)

greet(shout)
greet(whisper)
```

Output →

```
HI, I AM CREATED BY A FUNCTION
                                PASSED AS AN ARGUMENT.

hi, i am created by a function
                                passed as an argument.
```

Python Map →

The map() function executes a specified function for each item in an iterable. The item is sent to the function as a parameter.

Syntax

```
map(function, iterables)
```

map() Parameter

The map() function takes two parameters:

- function - a function that perform some action to each element of an iterable
- iterable - an iterable like sets, lists, tuples, etc
- You can pass more than one iterable to the map() function.

map() Return Value

The map() function returns an object of map class. The returned value can be passed to functions like

- list() - to convert to list

- set() - to convert to a set, and so on.

Code →

```
def calculateAddition(n):
    return n+n

numbers = (1, 2, 3, 4)
result = map(calculateAddition, numbers)
print(list(result))
```

Output →

```
[2, 4, 6, 8]
```

Parameters

function- It is a function in which a map passes each item of the iterable.

iterables- It is a sequence, collection or an iterator object which is to be mapped.

Return

It returns a list of results after applying a given function to each item of an iterable(list, tuple etc.)

```
def calculateAddition(n):
  return n+n

numbers = (1, 2, 3, 4)
result = map(calculateAddition, numbers)
print(list(result))
```

Python filter() Function

Python filter() function is used to get filtered elements. This function takes two arguments, first is a function and the second is iterable. The filter function returns a sequence from those elements of iterable for which function returns **True**.

The first argument can be None if the function is not available and returns only elements that are True.

Signature

1. filter (function, iterable)

Parameters

function: It is a function. If set to None returns only elements that are True.

Iterable: Any iterable sequence like list, tuple, and string.

Both the parameters are required.

Return

It returns the same as returned by the function.

Let's see some examples of filter() function to understand it's functionality.

Python filter() Function Example 1

This simple example returns values higher than 5 using filter function. See the below example.

```
# Python filter() function example
def filterdata(x):
    if x>5:
        return x
# Calling function
result = filter(filterdata,(1,2,6))
# Displaying result
print(list(result))
```

Output:

```
[6]
```

Python filter() Function Example 2

This function takes the first argument as a function and if no function is passed. It requires None to pass which returns all the True values. See the example below.

```
# Python filter() function example
# Calling function
result = filter(None,(1,0,6)) # returns all non-zero values
result2 = filter(None,(1,0,False,True)) # returns all non-zero and True values
# Displaying result
result = list(result)
result2 = list(result2)
print(result)
print(result2)
```

Output:

```
[1, 6]
[1, True]
```

Python Lambda Functions

Python Lambda Functions are anonymous function means that the function is without a name. The *lambda* keyword is used to define an anonymous function in Python.

However, Lambda functions can accept any number of arguments, but they can return only one value in the form of expression.

Syntax

```
lambda arguments : expression
```

Add 10 to argument `a`, and return the result:

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

```
OUTPUT: 15
```

Lambda functions can take any number of arguments:

Example

Multiply argument `a` with argument `b` and return the result:

```
x = lambda a, b : a * b
print(x(5, 6))
```

```
OUTPUT 30
```

INNER FUNCTION:➔To define an inner function ,we just create a function inside another function.

When we construct one function inside another, it is called a nested function.

```
1. def OutFun(): # outer function
2.     print("Hello, it is the outer function")
3.
4.     def InFun(): # inner function
5.         print("Hello, It is the inner function")
```

```
6.      InFun() # call inner
7.
8.  OutFun() # call outer function
```

Output:

```
Hello, it is the outer function
Hello, it is the inner function
```

Python Closures

A Closure is a function object that remembers values in enclosing scopes even if they are not present in memory.

Features of the Closures are:

1. Closures provide a kind of data hiding in our program and so we can avoid using global variables.
2. It is an efficient option when we don't have too many functions in our program.
3. It is a record that stores a function together with an environment: a mapping associating each free variable of the function (variables that are used locally but defined in an enclosing scope) with the value or reference to which the name was bound when the closure was created.
4. A closure—unlike a plain function—allows the function to access those captured variables through the closure's copies of their values or references, even when the function is invoked outside their scope.

When We can use a Closure

We can use the closure in the following conditions:

1. A program must have a nested function.
2. The function should refer to a value in the enclosed function.
3. The enclosing function should return the nested function.

- As observed from the above code, closures help to invoke functions outside their scope.

- The function **innerFunction** has its scope only inside the outerFunction. But with the use of closures, we can easily extend its scope to invoke a function outside its scope.

```
def add_num(n):
    def addition(x):
        return x+n
    return addition
add_2=add_num(2)
add_8=add_num(8)
print(add_2(4))
print(add_2(8))
print(add_8(add_2(7)))
```

Output:

```
6
10
17
```