Python Functions

This tutorial will learn about the basics of Python functions, including what they are, their syntax, their main components, return keywords, and major types. We will also see examples of how to define a Python function.

What are Python Functions?

A function is a collection of related assertions that performs a mathematical, analytical, or evaluative operation. Python functions are simple to define and essential to intermediate-level programming. The exact criteria hold to function names as they do to variable names. The goal is to group up certain often performed actions and define a function. Rather than rewriting the same code block over and over for varied input variables, we may call the function and repurpose the code included within it with different variables.

The functions are broad of two types, user-defined and built-in functions. It aids in keeping the software succinct, non-repetitive, and well-organized.

Advantages of Functions in Python

Python functions have the following benefits.

54.4M

1.1K

OOPs Concepts in Java

* By including functions, we can prevent repeating the same code block repeatedly in a program.
* Python functions, once defined, can be called many times and from anywhere in a program.
* If our Python program is large, it can be separated into numerous functions which is simple to track.
* The key accomplishment of Python functions is we can return as many outputs as we want with different arguments.

However, calling functions has always been overhead in a Python program.

Syntax of Python Function

**Code**

1. **def** name\_of\_function( parameters ):
2. """This is a docstring"""
3. # code block

The following elements make up define a function, as seen above.

* The beginning of a function header is indicated by a keyword called def.
* name\_of\_function is the function's name that we can use to separate it from others. We will use this name to call the function later in the program. The same criteria apply to naming functions as to naming variables in Python.
* We pass arguments to the defined function using parameters. They are optional, though.
* The function header is terminated by a colon (:).
* We can use a documentation string called docstring in the short form to explain the purpose of the function.
* The body of the function is made up of several valid Python statements. The indentation depth of the whole code block must be the same (usually 4 spaces).
* We can use a return expression to return a value from a defined function.

Example of a User-Defined Function

We will define a function that when called will return the square of the number passed to it as an argument.

**Code**

1. **def** square( num ):
2. """
3. This function computes the square of the number.
4. """
5. **return** num\*\*2
6. object\_ = square(9)
7. **print**( "The square of the number is: ", object\_ )

**Output:**

The square of the number is: 81

Calling a Function

A function is defined by using the def keyword and giving it a name, specifying the arguments that must be passed to the function, and structuring the code block.

After a function's fundamental framework is complete, we can call it from anywhere in the program. The following is an example of how to use the a\_function function.

**Code**

1. # Defining a function
2. **def** a\_function( string ):
3. "This prints the value of length of string"
4. **return** len(string)
6. # Calling the function we defined
7. **print**( "Length of the string Functions is: ", a\_function( "Functions" ) )
8. **print**( "Length of the string Python is: ", a\_function( "Python" ) )

**Output:**

Length of the string Functions is: 9

Length of the string Python is: 6

Pass by Reference vs. Value

In the Python programming language, all arguments are supplied by reference. It implies that if we modify the value of an argument within a function, the change is also reflected in the calling function. For instance,

**Code**

1. # defining the function
2. **def** square( my\_list ):
3. '''''This function will find the square of items in list'''
4. squares = []
5. **for** l **in** my\_list:
6. squares.append( l\*\*2 )
7. **return** squares
9. # calling the defined function
10. list\_ = [45, 52, 13];
11. result = square( list\_ )
12. **print**( "Squares of the list is: ", result )

**Output:**

Squares of the list is: [2025, 2704, 169]

Function Arguments

The following are the types of arguments that we can use to call a function:

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

Default Arguments

A default argument is a kind of parameter that takes as input a default value if no value is supplied for the argument when the function is called. Default arguments are demonstrated in the following instance.

**Code**

1. # Python code to demonstrate the use of default arguments
2. # defining a function
3. **def** function( num1, num2 = 40 ):
4. **print**("num1 is: ", num1)
5. **print**("num2 is: ", num2)

8. # Calling the function and passing only one argument
9. **print**( "Passing one argument" )
10. function(10)
12. # Now giving two arguments to the function
13. **print**( "Passing two arguments" )
14. function(10,30)

**Output:**

Passing one argument

num1 is: 10

num2 is: 40

Passing two arguments

num1 is: 10

num2 is: 30

Keyword Arguments

The arguments in a function called are connected to keyword arguments. If we provide keyword arguments while calling a function, the user uses the parameter label to identify which parameters value it is.

Since the Python interpreter will connect the keywords given to link the values with its parameters, we can omit some arguments or arrange them out of order. The function() method can also be called with keywords in the following manner:

**Code**

1. # Python code to demonstrate the use of keyword arguments
3. # Defining a function
4. **def** function( num1, num2 ):
5. **print**("num1 is: ", num1)
6. **print**("num2 is: ", num2)
8. # Calling function and passing arguments without using keyword
9. **print**( "Without using keyword" )
10. function( 50, 30)
12. # Calling function and passing arguments using keyword
13. **print**( "With using keyword" )
14. function( num2 = 50, num1 = 30)

**Output:**

Without using keyword

num1 is: 50

num2 is: 30

With using keyword

num1 is: 30

num2 is: 50

Required Arguments

The arguments given to a function while calling in a pre-defined positional sequence are required arguments. The count of required arguments in the method call must be equal to the count of arguments provided while defining the function.

We must send two arguments to the function function() in the correct order, or it will return a syntax error, as seen below.

**Code**

1. # Python code to demonstrate the use of default arguments
3. # Defining a function
4. **def** function( num1, num2 ):
5. **print**("num1 is: ", num1)
6. **print**("num2 is: ", num2)
8. # Calling function and passing two arguments out of order, we need num1 to be 20 and num2 to be 30
9. **print**( "Passing out of order arguments" )
10. function( 30, 20 )
12. # Calling function and passing only one argument
13. **print**( "Passing only one argument" )
14. **try**:
15. function( 30 )
16. **except**:
17. **print**( "Function needs two positional arguments" )

**Output:**

Passing out of order arguments

num1 is: 30

num2 is: 20

Passing only one argument

Function needs two positional arguments

Variable-Length Arguments

We can use special characters in Python functions to pass as many arguments as we want in a function. There are two types of characters that we can use for this purpose:

1. **\*args -**These are Non-Keyword Arguments
2. **\*\*kwargs -** These are Keyword Arguments.

Here is an example to clarify Variable length arguments

**Code**

1. # Python code to demonstrate the use of variable-length arguments
3. # Defining a function
4. **def** function( \*args\_list ):
5. ans = []
6. **for** l **in** args\_list:
7. ans.append( l.upper() )
8. **return** ans
9. # Passing args arguments
10. object = function('Python', 'Functions', 'tutorial')
11. **print**( object )
13. # defining a function
14. **def** function( \*\*kargs\_list ):
15. ans = []
16. **for** key, value **in** kargs\_list.items():
17. ans.append([key, value])
18. **return** ans
19. # Paasing kwargs arguments
20. object = function(First = "Python", Second = "Functions", Third = "Tutorial")
21. **print**(object)

**Output:**

['PYTHON', 'FUNCTIONS', 'TUTORIAL']

[['First', 'Python'], ['Second', 'Functions'], ['Third', 'Tutorial']]

return Statement

We write a return statement in a function to leave a function and give the calculated value when a defined function is called.

**Syntax:**

1. **return** < expression to be returned as output >

An argument, a statement, or a value can be used in the return statement, which is given as output when a specific task or function is completed. If we do not write a return statement, then None object is returned by a defined function.

Here is an example of a return statement in Python functions.

**Code**

1. # Python code to demonstrate the use of return statements
3. # Defining a function with return statement
4. **def** square( num ):
5. **return** num\*\*2
7. # Calling function and passing arguments.
8. **print**( "With return statement" )
9. **print**( square( 39 ) )
11. # Defining a function without return statement
12. **def** square( num ):
13. num\*\*2
15. # Calling function and passing arguments.
16. **print**( "Without return statement" )
17. **print**( square( 39 ) )

**Output:**

With return statement

1521

Without return statement

None

The Anonymous Functions

These types of Python functions are anonymous since we do not declare them, as we declare usual functions, using the def keyword. We can use the lambda keyword to define the short, single output, anonymous functions.

Lambda expressions can accept an unlimited number of arguments; however, they only return one value as the result of the function. They can't have numerous expressions or instructions in them. Since lambda needs an expression, an anonymous function cannot be directly called to print.

Lambda functions contain their unique local domain, meaning they can only reference variables in their argument list and the global domain name.

Although lambda expressions seem to be a one-line representation of a function, they are not like inline expressions in C and C++, which pass function stack allocations at execution for efficiency concerns.

**Syntax**

Lambda functions have exactly one line in their syntax:

1. **lambda** [argument1 [,argument2... .argumentn]] : expression

Below is an illustration of how to use the lambda function:

**Code**

1. # Defining a function
2. lambda\_ = **lambda** argument1, argument2: argument1 + argument2;
4. # Calling the function and passing values
5. **print**( "Value of the function is : ", lambda\_( 20, 30 ) )
6. **print**( "Value of the function is : ", lambda\_( 40, 50 ) )

**Output:**

Value of the function is : 50

Value of the function is : 90

Scope and Lifetime of Variables

The scope of a variable refers to the domain of a program wherever it is declared. A function's arguments and variables are not accessible outside the defined function. As a result, they only have a local domain.

The period of a variable's existence in RAM is referred to as its lifetime. Variables within a function have the same lifespan as the function itself.

When we get out of the function, they are removed. As a result, a function does not retain a variable's value from earlier executions.

Here's a simple example of a variable's scope within a function.

**Code**

1. #defining a function to print a number.
2. **def** number( ):
3. num = 30
4. **print**( "Value of num inside the function: ", num)
6. num = 20
7. number()
8. **print**( "Value of num outside the function:", num)

**Output:**

Value of num inside the function: 30

Value of num outside the function: 20

Here, we can observe that the initial value of num is 20. Even if the function number() modified the value of num to 30, the value of num outside the function remained unchanged.

This is because the variable num within the function is distinct from the variable outside the function (local to the function). Despite their identical variable name, they are 2 distinct variables having distinct scopes.

Variables beyond the function, on the contrary, are accessible within the function. These variables have a global reach.

We can retrieve their values inside the function but cannot alter (change) them. If we declare a variable global using the keyword global, we can also change the variable's value outside the function.

Python Function within Another Function

Functions are considered first-class objects in Python. In a programming language, first-class objects are treated the same wherever they are used. They can be used in conditional expressions, as arguments, and saved in built-in data structures. If a programming language handles functions as first-class entities, it is said to implement first-class functions. Python supports the notion of First Class functions.

Inner or nested function refers to a function defined within another defined function. Inner functions can access the parameters of the outer scope. Inner functions are constructed to cover them from the changes that happen outside the function. Many developers regard this process as encapsulation.

**Code**

1. # Python code to show how to access variables of a nested functions
2. # defining a nested function
3. **def** function1():
4. string = 'Python functions tutorial'
6. **def** function2():
7. **print**( string )
9. function2()
10. function1()

**Output:**