Python Functions

This tutorial will go over the fundamentals of Python functions, including what they are, their syntax, their primary parts, return keywords, and significant types. We'll also look at some examples of Python function definitions.

What are Python Functions?

A collection of related assertions that carry out a mathematical, analytical, or evaluative operation is known as a function. An assortment of proclamations called Python Capabilities returns the specific errand. Python functions are necessary for intermediate-level programming and are easy to define. Function names meet the same standards as variable names do. The objective is to define a function and group-specific frequently performed actions. Instead of repeatedly creating the same code block for various input variables, we can call the function and reuse the code it contains with different variables.

Client-characterized and worked-in capabilities are the two primary classes of capabilities in Python. It aids in maintaining the program's uniqueness, conciseness, and structure.

Advantages of Python Functions

Pause We can stop a program from repeatedly using the same code block by including functions.

* Once defined, Python functions can be called multiple times and from any location in a program.
* Our Python program can be broken up into numerous, easy-to-follow functions if it is significant.
* The ability to return as many outputs as we want using a variety of arguments is one of Python's most significant achievements.
* However, Python programs have always incurred overhead when calling functions.

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

**Syntax**

1. #  An example Python Function
2. **def** function\_name( parameters ):
3. # code block

The accompanying components make up to characterize a capability, as seen previously.

* The start of a capability header is shown by a catchphrase called def.
* function\_name is the function's name, which we can use to distinguish it from other functions. We will utilize this name to call the capability later in the program. Name functions in Python must adhere to the same guidelines as naming variables.
* Using parameters, we provide the defined function with arguments. Notwithstanding, they are discretionary.
* A colon (:) marks the function header's end.
* We can utilize a documentation string called docstring in the short structure to make sense of the reason for the capability.
* Several valid Python statements make up the function's body. The entire code block's indentation depth-typically four spaces-must be the same.
* A return expression can get a value from a defined function.

Illustration of a User-Defined Function

We will define a function that returns the argument number's square when called.

1. # Example Python Code for User-Defined function
2. **def** square( num ):
3. """
4. This function computes the square of the number.
5. """
6. **return** num\*\*2
7. object\_ = square(6)
8. **print**( "The square of the given number is: ", object\_ )

**Output:**

The square of the given number is: 36

Calling a Function

Calling a Function To define a function, use the def keyword to give it a name, specify the arguments it must receive, and organize the code block.

When the fundamental framework for a function is finished, we can call it from anywhere in the program. An illustration of how to use the a\_function function can be found below.

1. # Example Python Code for calling a function
2. # Defining a function
3. **def** a\_function( string ):
4. "This prints the value of length of string"
5. **return** len(string)
7. # Calling the function we defined
8. **print**( "Length of the string Functions is: ", a\_function( "Functions" ) )
9. **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. Pass by Value

In the Python programming language, all parameters are passed by reference. It shows that if we modify the worth of contention within a capability, the calling capability will similarly mirror the change. For instance,

**Code**

1. # Example Python Code for Pass by Reference vs. Value
2. # defining the function
3. **def** square( item\_list ):
4. '''''''This function will find the square of items in the list'''
5. squares = [ ]
6. **for** l **in** item\_list:
7. squares.append( l\*\*2 )
8. **return** squares
10. # calling the defined function
11. my\_list = [17, 52, 8];
12. my\_result = square( my\_list )
13. **print**( "Squares of the list are: ", my\_result )

**Output:**

Squares of the list are: [289, 2704, 64]

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

1) Default Arguments

A default contention is a boundary that takes as information a default esteem, assuming that no worth is provided for the contention when the capability is called. The following example demonstrates default arguments.

**Code**

1. # Python code to demonstrate the use of default arguments
2. # defining a function
3. **def** function( n1, n2 = 20 ):
4. **print**("number 1 is: ", n1)
5. **print**("number 2 is: ", n2)

8. # Calling the function and passing only one argument
9. **print**( "Passing only one argument" )
10. function(30)
12. # Now giving two arguments to the function
13. **print**( "Passing two arguments" )
14. function(50,30)

**Output:**

Passing only one argument

number 1 is: 30

number 2 is: 20

Passing two arguments

number 1 is: 50

number 2 is: 30

2) Keyword Arguments

Keyword arguments are linked to the arguments of a called function. While summoning a capability with watchword contentions, the client might tell whose boundary esteem it is by looking at the boundary name.

We can eliminate or orchestrate specific contentions in an alternate request since the Python translator will interface the furnished watchwords to connect the qualities with its boundaries. One more method for utilizing watchwords to summon the capability() strategy is as per the following:

**Code**

1. # Python code to demonstrate the use of keyword arguments
2. # Defining a function
3. **def** function( n1, n2 ):
4. **print**("number 1 is: ", n1)
5. **print**("number 2 is: ", n2)
7. # Calling function and passing arguments without using keyword
8. **print**( "Without using keyword" )
9. function( 50, 30)
11. # Calling function and passing arguments using keyword
12. **print**( "With using keyword" )
13. function( n2 = 50, n1 = 30)

**Output:**

Without using keyword

number 1 is: 50

number 2 is: 30

With using keyword

number 1 is: 30

number 2 is: 50

3) Required Arguments

Required arguments are those supplied to a function during its call in a predetermined positional sequence. The number of arguments required in the method call must be the same as those provided in the function's definition.

We should send two contentions to the capability() all put together; it will return a language structure blunder, as seen beneath.

**Code**

1. # Python code to demonstrate the use of default arguments
2. # Defining a function
3. **def** function( n1, n2 ):
4. **print**("number 1 is: ", n1)
5. **print**("number 2 is: ", n2)
7. # Calling function and passing two arguments out of order, we need num1 to be 20 and num2 to be 30
8. **print**( "Passing out of order arguments" )
9. function( 30, 20 )
11. # Calling function and passing only one argument
12. **print**( "Passing only one argument" )
13. **try**:
14. function( 30 )
15. **except**:
16. **print**( "Function needs two positional arguments" )

**Output:**

Passing out of order arguments

number 1 is: 30

number 2 is: 20

Passing only one argument

Function needs two positional arguments

4) Variable-Length Arguments

We can involve unique characters in Python capabilities to pass many contentions. However, we need a capability. This can be accomplished with one of two types of characters:

"args" and "kwargs" refer to arguments not based on keywords.

To help you understand arguments of variable length, here's an example.

**Code**

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

**Output:**

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

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

return Statement

When a defined function is called, a return statement is written to exit the function and return the calculated value.

**Syntax:**

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

The return statement can be an argument, a statement, or a value, and it is provided as output when a particular job or function is finished. A declared function will return an empty string if no return statement is written.

A return statement in Python functions is depicted in the following example.

**Code**

1. # Python code to demonstrate the use of return statements
2. # Defining a function with return statement
3. **def** square( num ):
4. **return** num\*\*2
6. # Calling function and passing arguments.
7. **print**( "With return statement" )
8. **print**( square( 52 ) )
10. # Defining a function without return statement
11. **def** square( num ):
12. num\*\*2
14. # Calling function and passing arguments.
15. **print**( "Without return statement" )
16. **print**( square( 52 ) )

**Output:**

With return statement

2704

Without return statement

None

The Anonymous Functions

Since we do not use the def keyword to declare these kinds of Python functions, they are unknown. The lambda keyword can define anonymous, short, single-output functions.

Arguments can be accepted in any number by lambda expressions; However, the function only produces a single value from them. They cannot contain multiple instructions or expressions. Since lambda needs articulation, a mysterious capability can't be straightforwardly called to print.

Lambda functions can only refer to variables in their argument list and the global domain name because they contain their distinct local domain.

In contrast to inline expressions in C and C++, which pass function stack allocations at execution for efficiency reasons, lambda expressions appear to be one-line representations of functions.

**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. # Python code to demonstrate ananymous functions
2. # Defining a function
3. lambda\_ = **lambda** argument1, argument2: argument1 + argument2;
5. # Calling the function and passing values
6. **print**( "Value of the function is : ", lambda\_( 20, 30 ) )
7. **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

A variable's scope refers to the program's domain wherever it is declared. A capability's contentions and factors are not external to the characterized capability. They only have a local domain as a result.

The length of time a variable remains in RAM is its lifespan. The lifespan of a function is the same as the lifespan of its internal variables. When we exit the function, they are taken away from us. As a result, the value of a variable in a function does not persist from previous executions.

An easy illustration of a function's scope for a variable can be found here.

**Code**

1. # Python code to demonstrate scope and lifetime of variables
2. #defining a function to print a number.
3. **def** number( ):
4. num = 50
5. **print**( "Value of num inside the function: ", num)
7. num = 10
8. number()
9. **print**( "Value of num outside the function:", num)

**Output:**

Value of num inside the function: 50

Value of num outside the function: 10

Here, we can see that the initial value of num is 10. Even though the function number() changed the value of num to 50, the value of num outside of the function remained unchanged.

This is because the capability's interior variable num is not quite the same as the outer variable (nearby to the capability). Despite having a similar variable name, they are separate factors with discrete extensions.

Factors past the capability are available inside the capability. The impact of these variables is global. We can retrieve their values within the function, but we cannot alter or change them. The value of a variable can be changed outside of the function if it is declared global with the keyword global.

Python Capability inside Another Capability

Capabilities are viewed as top-of-the-line objects in Python. First-class objects are treated the same everywhere they are used in a programming language. They can be stored in built-in data structures, used as arguments, and in conditional expressions. If a programming language treats functions like first-class objects, it is considered to implement first-class functions. Python lends its support to the concept of First-Class functions.

A function defined within another is called an "inner" or "nested" function. The parameters of the outer scope are accessible to inner functions. Internal capabilities are developed to cover them from the progressions outside the capability. Numerous designers see this interaction as an embodiment.

**Code**

1. # Python code to show how to access variables of a nested functions
2. # defining a nested function
3. **def** word():
4. string = 'Python functions tutorial'
5. x = 5
6. **def** number():
7. **print**( string )
8. **print**( x )
10. number()
11. word()

**Output:**

Python functions tutorial

5

https://www.javatpoint.com/python-functions