**Que1.what is function?**

Ans-A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result.

Function improves efficiency and reduces errors because of the reusability of a code. Once we create a function, we can call it anywhere and anytime. The benefit of using a function is reusability and modularity.

In Python a function is defined using the def keyword:

Example

def my\_function():  
  print("Hello from a function")

## Calling a Function

To call a function, use the function name followed by parenthesis:

def my\_function():  
  print("Hello from a function")  
  
**my\_function()**

Python support two types of functions

1. Built-in function
2. User-defined function

1.Built-in function

The functions which are come along with Python itself are called a [built-in function](https://docs.python.org/3/library/functions.html)orpredefined function. Some of them are listed below.  
range(), id(), type(), input(), eval()

2.User-defined function

Functions which are created by programmer explicitly according to the requirement are called a user-defined function.

#Creating a function without any parameters

def message():

print("Welcome to Python class.")

# call function using its name

message()

#Creating a function with parameters

def course\_func(name, course\_name):

print("Hello", name, "Welcome to PYnative")

print("Your course name is", course\_name)

# call function

course\_func('John', 'Python')

## Docstrings

In Python, the documentation string is also called a **docstring**. It is a descriptive text (like a comment) written by a programmer to let others know what block of code does.

We write docstring in source code and define it immediately after module, class, function, or method definition.

It is being declared using triple single quotes (''' ''') or triple-double quote(""" """).

### Single-Line Docstring

The single-line docstring is a docstring that fits in one line. We can use the triple single or triple-double quotes to define it.

def factorial(x):

"""This function returns the factorial of a given number."""

return x

# access doc string

print(factorial.\_\_doc\_\_)

Return Value From a Function

In Python, to return value from the function, a return statement is used. It returns the value of the expression following the returns keyword.

Syntax of return statement

def fun():

statement-1

statement-2

statement-3

.

.

return [expression]

The return value is nothing but a outcome of function.

The return statement ends the function execution.

For a function, it is not mandatory to return a value.

If a return statement is used without any expression, then the None is returned.

The return statement should be inside of the function block.

**Return Multiple Values**

You can also return multiple values from a function. Use the return statement by separating each expression by a comma.

def arithmetic(num1, num2):

add = num1 + num2

sub = num1 - num2

multiply = num1 \* num2

division = num1 / num2

# return four values

return add, sub, multiply, division

# read four return values in four variables

a, b, c, d = arithmetic(10, 2)

print("Addition: ", a)

print("Subtraction: ", b)

print("Multiplication: ", c)

print("Division: ", d)

## **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/ Positional arguments
4. Variable-length arguments

# [Python function argument passing sequence](https://stackoverflow.com/questions/24844096/python-function-argument-passing-sequence)-

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

### 1.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.

def function( num1, num2 = 40 ):

print("num1 is: ", num1)

print("num2 is: ", num2)

# Calling the function and passing only one argument

print( "Passing one argument" )

function(10)

# Now giving two arguments to the function

print( "Passing two arguments" )

function(10,30)

### 2. Keyword Arguments

A keyword argument is an argument value, passed to function preceded by the variable name and an equals sign.

In keyword arguments order of argument is not matter, but the number of arguments must match. Otherwise, we will get an error.

def message(name, surname):

print("Hello", name, surname)

message(name="Madhuri", surname="Burande")

message(surname="Swara", name="Adat"

### 3.Required/ Positional 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.

if we change the order, then the result may change. Also, If we change the number of arguments, then we will get an error.

def function( num1, num2 ):

print("num1 is: ", num1)

print("num2 is: ", num2)

print( "Passing out of order arguments" )

function( 30, 20 )

#order change

function(22)

### 4.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

def addition(\*numbers):

total = 0

for no in numbers:

total = total + no

print("Sum is:", total)

# 0 arguments

addition()

# 5 arguments

addition(10, 5, 2, 5, 4)

# 3 arguments

addition(78, 7, 2.5)

## **Recursive Function**

A recursive function is a function that calls itself, again and again.

Consider, calculating the factorial of a number is a repetitive activity, in that case, we can call a function again and again, which calculates factorial.

def factorial(num):

if num==0:

return 1

elif num==1:

return 1

else:

return num\*factorial(num-1)

print("Factorial is:",factorial(5))

## Python Anonymous/Lambda Function

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.

**Syntax**

Lambda functions have exactly one line in their syntax:

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

l = [10, 5, 12, 78, 6, 1, 7, 9]

even\_nos = list(filter(lambda x: x % 2 == 0, l))

print("Even numbers are: ", even\_nos)

# **Python Scope**

A variable is only available from inside the region it is created. This is called **scope**.

## **Local Scope**

A variable created inside a function belongs to the local scope of that function, and can only be used inside that function.

def myfunc():  
  x = 300  
  print(x)  
  
myfunc()

### Function Inside Function

As explained in the example above, the variable x is not available outside the function, but it is available for any function inside the function:

def myfunc():

x = 300

def myinnerfunc():

print(x)

myinnerfunc()

myfunc()

## **Global Scope**

## A Global variable is a variable that declares outside of the function

x = 500

def myfunc():

print(x)

myfunc()

print(x)

## global and local variable.

x = 300

def myfunc():

x = 200

print(x)

myfunc()

print(x)

#### **Global Keyword in Function**

In Python, global is the keyword used to access the actual global variable from outside the function. we use the global keyword for two purposes:

global\_var = 5

def function1():

print("Value in 1st function :", global\_var)

def function2():

# Modify global variable

# function will treat it as a local variable

global\_var = 555

print("Value in 2nd function :", global\_var)

def function3():

print("Value in 3rd function :", global\_var)

function1()

function2()

function3()

# **reduce() in Python**

In Python, the reduce() function is used to **minimize sequence elements** into a **single value** by applying the specified condition.

The reduce() function is present in the functools module; hence, we need to import it using the import statement before using it

Example-

from functools import reduce

lis = [1, 3, 5, 6, 2, ]

# using reduce to compute sum of list

print("The sum of the list elements is : ", end="")

print(reduce(lambda a, b: a+b, lis))

# **Python map() function**

**map()** function returns a map object(which is an iterator) of the results after applying the given function to each item of a given iterable (list, tuple etc.)

**Syntax :**

def addition(n):

return n + n

# We double all numbers using map()

numbers = (1, 2, 3, 4)

result = map(addition, numbers)

print(list(result))

### filter() function in Python

In Python, the filter() function is used to return the filtered value. We use this function to filter values based on some conditions.

**Syntax of filter() function:**

**filter**(funtion, sequence)

l = [-10, 5, 12, -78, 6, -1, -7, 9]

positive\_nos = list(filter(lambda x: x > 0, l))

print("Positive numbers are: ", positive\_nos)

## **Namespaces in Python**

A namespace is a collection of currently defined symbolic names along with information about the object that each name references. You can think of a namespace as a [dictionary](https://realpython.com/python-dicts) in which the keys are the object names and the values are the objects themselves

1. Built-In
2. Global
3. Enclosing
4. Local

### 1.The Built-In Namespace

The **built-in namespace** contains the names of all of Python’s built-in objects

Example=

Tuple,list,def etc.

### The Global Namespace

### The ****global namespace**** contains any names defined at the level of the main program. Python creates the global namespace when the main program body starts, and it remains in existence until the interpreter terminates.

### 3.Enclosing Namespaces

### When a function is defined inside a function, it creates an enclosed namespace.

### When a function is defined inside a function, it creates an enclosed namespace. Its lifecycle is the same as the local namespace.

## **4.Local namespaces**

### A variable created inside a function belongs to the local scope of that function, and can only be used inside that function.