FUNCTIONS IN PYTHON

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

Functions are a convenient way to divide your code into useful blocks, allowing us to order our code, make it more readable, reuse it and save some time.

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 organized and manageable.

Furthermore, it avoids repetition and makes the code reusable.

Why use function in Python?

1. **Code re-usability**: Lets say we are writing an application in Python where we need to perform a specific task in several places of our code, assume that we need to write 10 lines of code to do that specific task. It would be better to write those 10 lines of code in a function and just call the function wherever needed, because writing those 10 lines every time you perform that task is tedious, it would make your code lengthy, less-readable and increase the chances of human errors.

2. **Improves Readability**: By using functions for frequent tasks you make your code structured and readable. It would be easier for anyone to look at the code and be able to understand the flow and purpose of the code.

3. **Avoid redundancy**: When you no longer repeat the same lines of code throughout the code and use functions in places of those, you actually avoiding the redundancy that you may have created by not using functions.

Types of Functions

- As you already know, Python gives you many built-in functions like range(),print(), etc. but you can also create your own functions. These functions are called *user-defined functions*
- There are mainly two types of functions.
- **User-define functions** - The user-defined functions are those define by the **user** to perform the specific task.
- **Built-in functions** - The built-in functions are those functions that are **pre-defined** in Python.

Creating & Calling a Function

- In Python a function is defined using the def keyword:

Example

```
#function definition
def hello_world():
    print("hello world")
# function calling
hello_world()
```

Output

hello_world

```
def my_function(parameters):
    function_block
return expression
```

Let's understand the syntax of functions definition.

- The **def** keyword, along with the function name is used to define the function.
- The identifier rule must follow the function name.
- A function accepts the parameter (argument), and they can be optional.
- The function block is started with the colon (:), and block statements must be at the same indentation.
- The **return** statement is used to return the value. A function can have only one **return**

return statement

- The return statement is used at the end of the function and returns the result of the function. It terminates the function execution and transfers the result where the function is called. The return statement cannot be used outside of the function.

Syntax

return [expression_list]

Example

```
# Defining function
def sum():
    a = 10
    b = 20
    c = a+b
    return c
# calling sum() function in print statement
print("The sum is:",sum())
```

Output

The sum is: 30

Example

```
def my_function(x):
  return 5 * x

print(my_function(3))
print(my_function(5))
print(my_function(9))
```

Output

```
15
25
45
```

Creating function without return statement

```
# Defining function
def sum():
    a = 10
    b = 20
    c = a+b
# calling sum() function in print statement
print(sum())
```

Output

None

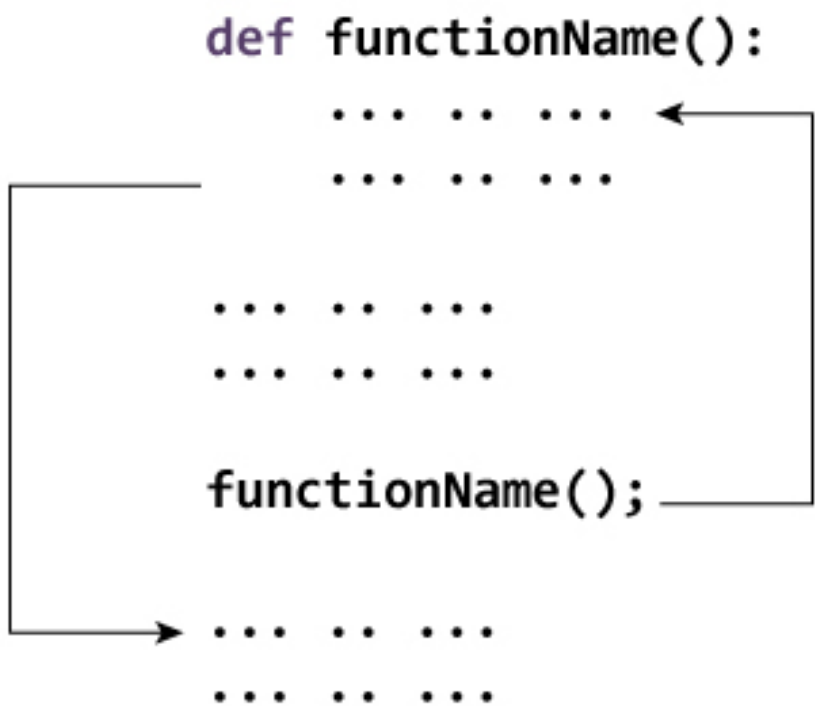
Here we have a function add() that adds two numbers passed to it as parameters.
Later after function declaration we are calling the function twice in our program to perform the addition.

```
def add(num1, num2):
    return num1 + num2


sum1 = add(100, 200)
sum2 = add(8, 9)
print(sum1)
print(sum2)
```

Output

```
300
17
```


def functionName():
... .
... .
... .
... .
functionName();
... .
... .

Scope and Lifetime of variables

Scope of a variable is the portion of a program where the variable is recognized. Parameters and variables defined inside a function are not visible from outside the function. Hence, they have a local scope.

The lifetime of a variable is the period throughout which the variable exits in the memory. The lifetime of variables inside a function is as long as the function executes.

They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

Here is an example to illustrate the scope of a variable inside a function.

```
def my_func():
  x = 10
  print("Value inside function:",x)
  x = 20
 my_func()
print("Value outside func
```

Output

```
Value inside function: 10 Value outside function: 20
```

Pass Statement

function definitions cannot be empty, but if you for some reason have a function definition with no content, put in the pass statement to avoid getting an error.

Example

```
def myfunc():
  pass
```

Default arguments in Function

- Now that we know how to declare and call a function, lets see how can we use the **default arguments**.
- By using default arguments we can avoid the errors that may arise while calling a function without passing all the parameters.

default argument for second parameter

```
def add(num1, num2=1):
   return num1 + num2
sum1 = add(100, 200)
sum2 = add(8) # used default argument for second param
sum3 = add(100) # used default argument for second param
print(sum1)
print(sum2)
print(sum3)
```

Output

```
300
9
101
```

Example

```
def my_function(country = "Norway"):
    print("I am from " + country)

my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```

Output

I am from Sweden
I am from India
I am from Norway
I am from Brazil

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 separate them with a comma.

```
#defining the function
def func (name):
    print("Hi ",name)
#calling the function
func("Deva")
```

Output

Hi Deva

Example 2

```
#Python function to calculate the sum of two variables
#defining the function
def sum (a,b):
    return a+b;

#taking values from the user
a = int(input("Enter a: "))
b = int(input("Enter b: "))

#printing the sum of a and b
print("Sum = ",sum(a,b))
```

Output

```
Enter a: 10
Enter b: 20
Sum = 30
```

Parameters or Arguments?

The terms *parameter* and *argument* can be used for the same thing: information that are passed into a function.

From a function's perspective:

A parameter is the variable listed inside the parentheses in the function definition.

An argument is the value that is sent to the function when it is called.

Number of Arguments

By default, a function must be called with the correct number of arguments.
Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

Arbitrary Arguments, *args

If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.
This way the function will receive a *tuple* of arguments, and can access the items accordingly:

```
def my_function(*kids):
  print("The youngest child is " + kids[2])

my_function("Emi", "Tom", "Linus")
```

Output

The youngest child is Linus

Keyword Arguments

- You can also send arguments with the *key* = *value* syntax.
- This way the order of the arguments does not matter.

Example

```
def my_function(child3, child2, child1):
    print("The youngest child is " + child3)

my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

Output

The youngest child is Linus

Passing a List as an Argument

- You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.
- E.g. if you send a List as an argument, it will still be a List when it reaches the function:

Example

```
def my_function(food):
  for x in food:
    print(x)

fruits = ["apple", "banana", "cherry"]

my_function(fruits)
```

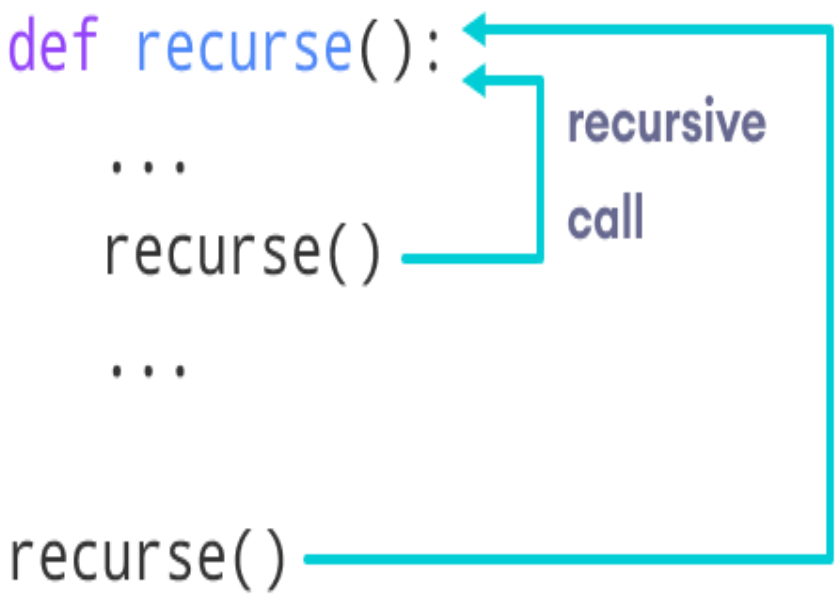
Output

```
apple
banana
cherry
```

Python Recursion

What is recursion?

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



Types of formal arguments

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

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

```
def printinfo( name, age ):
   print "Name: ", name
   print "Age ", age
   return;
printinfo( age=50, name="miki" )
```

When the above code is executed, it produces the following result –

```
Name: miki
Age 50
```

The *Anonymous* Functions

These functions are called anonymous because they are not declared in the standard manner by using the *def* keyword. You can use the ***lambda*** keyword to create small anonymous functions.

- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.

- An anonymous function cannot be a direct call to print because lambda requires an expression

- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.

- Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

Syntax

The syntax of *lambda* functions contains only a single statement, which is as follows −

```
lambda [arg1 [,arg2,.....argn]]:expression
```

Example:

```
sum = lambda no1, no2: no1 + no2;

print "Value of total : ", sum( 10, 20 )
print "Value of total : ", sum( 20, 20 )
```

Output

Value of total : 30
Value of total : 40

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

A lambda function that multiplies argument a with argument b and print the result:

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

A lambda function that sums argument a, b, and c and print the result:

```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

Python Global, Local and Nonlocal variables

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.

```
x = "global“
def ex1():
print("x inside:", x)

ex1()
print("x outside:", x)
```

Output

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

scope is known as a local variable.

Example 2: Accessing local variable outside the scope

```
def ex2():
 y = "local“

 ex2()
 print(y)
```

Output

```
NameError: name 'y' is not defined
```

Normally, we declare a variable inside the function to create a local variable.

```
def ex1():
y = "local"
print(y)

 ex1()
```

Output

```
local
```

Using Global and Local variables in the same code

```
x = "global"
def ex1():
  global x
  y = "local"
 x = x * 2
  print(x)
  print(y)
 ex1()
```

Output

```
global global
 local
```

Global variable and Local variable with same name

```
x = 5
 def ex1():
  x = 10
 print("local x:", x)
 ex1()
 print("global x:", x)
```

Output

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

Nonlocal Variables

- Nonlocal variables are used in nested functions whose local scope is not defined. This means that the variable can be neither in the local nor the global scope.