FUNCTIONS

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.

As you already know, Python gives you many **built-in functions** like print(), etc. but you can also create your own functions. These functions are called **user-defined functions.**

Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

Function blocks begin with the keyword def followed by the function name and parentheses (( )).

Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.

The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.

The code block within every function starts with a colon (:) and is indented.

Syntax

```
def functionname( parameters ):
    function_suite
    return [expression]
```

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

Example

The following function takes a string as input parameter and prints it on standard screen.

```
def printme( str ):
   "This prints a passed string into this function"
   print(str)
   return
```

Calling a Function

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt.

Following is the example to call printme() function

```
# Function definition is here
def printme( str ):
   "This prints a passed string into this function"
   print str
   return;

# Now you can call printme function
printme("I'm first call to user defined function!")
printme("Again second call to the same function")
```

Output:

```
I'm first call to  user defined function!
Again second call to the same function
```

Function Arguments

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

Required arguments

Keyword arguments

Default arguments

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.

To call the function *printme()*, you definitely need to pass one argument, otherwise it gives a syntax error as follows −

```
# Function definition is here
def printme( str ):
   "This prints a passed string into this function"
   print str
   return;

# Now you can call printme function
printme()
```

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

```
Traceback (most recent call last):
  File "test.py", line 11, in <module>
    printme();
TypeError: printme() takes exactly 1 argument (0 given)
```

Keyword arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters.

You can also make keyword calls to the *printme()* function in the following ways –

```
# Function definition is here
def printme( str ):
   "This prints a passed string into this function"
   print str
   return;

# Now you can call printme function
printme( str = "My string")
```

```
Output:
My string
```

The following example gives a more clear picture. Note that the order of parameters does not matter.

```
# Function definition is here
def printinfo( name, age ):
   "This prints a passed info into this function"
   print "Name: ", name
   print "Age ", age
```

```
   return;

# Now you can call printinfo function
printinfo( age=50, name="miki" )

Output:
Name:  miki
Age  50
```

Default arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument.

The following example gives an idea on default arguments, it prints default age if it is not passed −

```
# Function definition is here
def printinfo( name, age = 35 ):
   "This prints a passed info into this function"
   print "Name: ", name
   print "Age ", age
   return;

# Now you can call printinfo function
printinfo( age=50, name="miki" )
printinfo( name="miki" )

Output−

Name:  miki
Age  50
Name:  miki
Age  35
```

Variable Length Argument(*args and **kwargs in Python)

*args

- The special syntax **args* in function definitions in python is used to pass a variable number of arguments to a function. It is used to pass a non-keyworded, variable-length argument list.
- The syntax is to use the symbol * to take in a variable number of arguments; by convention, it is often used with the word args.
- What **args* allows you to do is take in more arguments than the number of formal arguments that you previously defined. With **args*, any number of extra arguments can be tacked on to your current formal parameters (including zero extra arguments).
- Using the *, the variable that we associate with the * becomes an iterable meaning you can do things like iterate over it, run some higher order functions such as map and filter, etc

```
nts def myFun(*args):
      for arg in args:
          print(arg)
myFun("hello",'welcome','to','python','programming')
```

Output:

Hello
Welcome
to
Python
Programming

Example:

 # Python program to illustrate
 # *args with first extra argument

```
def myFun(arg1, *argv):
        print ("First argument :", arg1)
        for arg in argv:
                print("Next argument through *argv :", arg)

myFun('Hello', 'Welcome', 'to', 'Python')
```

Output:
First argument : Hello
Next argument through *argv : Welcome
Next argument through *argv : to
Next argument through *argv : GeeksforGeeks

**kwargs

The special syntax ***kwargs* in function definitions in python is used to pass a keyworded, variable-length argument list. We use the name *kwargs* with the double star. The reason is because the double star allows us to pass through keyword arguments (and any number of them).

- A keyword argument is where you provide a name to the variable as you pass it into the function.
- One can think of the *kwargs* as being a dictionary that maps each keyword to the value that we pass alongside it. That is why when we iterate over the *kwargs* there doesn't seem to be any order in which they were printed out

Example:

```
# Python program to illustrate
# *kargs for variable number of keyword arguments
```

```
def myFun(**kwargs):
    for key, value in kwargs.items():
        print ("%s == %s" %(key, value))

myFun(first ='Geeks', mid ='for', last='Geeks')
```

Output

```
last == Geeks
mid == for
first == Geeks
```

Example:

```
# Python program to illustrate **kargs for
# variable number of keyword arguments with
# one extra argument.

def myFun(arg1, **kwargs):
    for key, value in kwargs.items():
        print ("%s == %s" %(key, value))

# Driver code
myFun("Hi", first ='Geeks', mid ='for', last='Geeks')
```

Output:

```
last == Geeks
mid == for
first == Geeks
```

Anonymous Functions In Python, anonymous function means that a function is without a name. As we already know that *def* keyword is used to define the normal functions and the *lambda* keyword is used to create anonymous functions. It has the following syntax:

lambda arguments: expression

- This function can have any number of arguments but only one expression, which is evaluated and returned.
- One is free to use lambda functions wherever function objects are required. You need to keep in your knowledge that lambda functions are syntactically restricted to a single expression.
- It has various uses in particular fields of programming besides other types of expressions in functions.

Syntax

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

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

Following is the example to show how *lambda* form of function works −

```
# Function definition is here
sum = lambda arg1, arg2: arg1 + arg2;

# Now you can call sum as a function
print "Value of total : ", sum( 10, 20 )
print "Value of total : ", sum( 20, 20 )
```

Output:

```
Value of total :  30
Value of total :  40
```

Example:

```
# Python code to illustrate cube of a number
# showing the difference between def() and lambda().
def cube(y):
    return y*y*y;
g = lambda x: x*x*x
print(g(7))
```

```
print(cube(5))
```

Output:

```
343
125
```

Without using Lambda : Here, both of them return the cube of a given number. But, while using def, we needed to define a function with a name cube and needed to pass a value to it. After execution, we also needed to return the result from where the function was called using the *return* keyword.

Using Lambda : Lambda definition does not include a "return" statement, it always contains an expression which is returned. We can also put a lambda definition anywhere a function is expected, and we don't have to assign it to a variable at all. This is the simplicity of lambda functions.

The *return* Statement

The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

All the above examples are not returning any value. You can return a value from a function as follows −

```
# Function definition is here
def sum( arg1, arg2 ):
   total = arg1 + arg2
   print "Inside the function : ", total
   return total;
```

```
# Now you can call sum function
total = sum( 10, 20 );
print "Outside the function : ", total
```

Output:

```
Inside the function :  30
Outside the function :  30
```

Scope of Variables

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.

The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python −

Global variables

Local variables

Global vs. Local variables

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

This means that local variables can be accessed only inside the function in which they are declared.

Global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope.

Following is a simple example −

Example: Create a Global Variable

```
x = "global"
def foo():
 print("x inside:", x)
foo()
print("x outside:", x)
```

Output:

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

Example: Changing value inside a function

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

Output:

UnboundLocalError: local variable 'x' referenced before assignment

Example: **Local variable**

False **True**

def foo(): y = "local" foo() print(y) Output: NameError: name 'y' is not defined	def foo(): y = "local" print(y) foo() Output: local

Global and Local variable:

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

Output:

```
globalglobal
local
globalglobal
```