Introduction to PYTHON

Function

Call by/Pass by

Object Reference

By: Atul Kumar Uttam

Function in Python

Formal arguments

```
def   function_name( parameters ):
        '''function_docstring'''
        Statement/s
        return [expression]
```

Documentation string   Optional

Actual arguments

```
function_name( parameters )
```

Function Example - 1

```
>>>def fun(a, b):
        '''Function to add two values'''
        c = a + b
        return c

>>>fun.__doc__
'Function to add two values'
>>>z = fun(10, 20)
>>>z
30
```

Function Example - 2

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


x = 20
my_func()
print("Value outside function:",x)
```

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

Call by Value vs Call by Reference

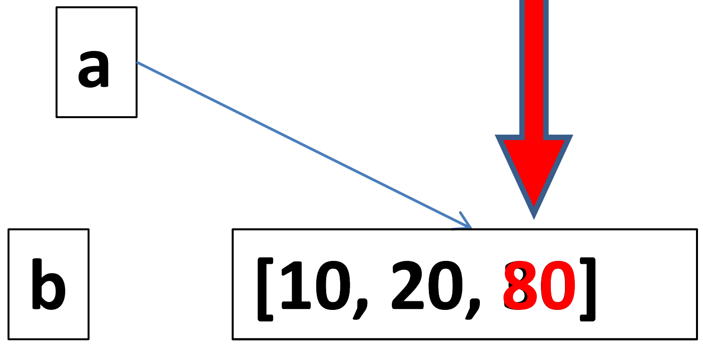
- Function arguments receives a copy of values
- Function arguments receives the reference of values
- There is no changes in actual arguments
- If there is any modification by formal arguments then there will be the same changes in actual arguments.
- In Python Neither of these two concepts are applicable,
- the values are sent to functions by means of **object reference.**

Call by/Pass by Object Reference

- In Python Values are passed to function by object reference.
  - if **object is immutable** then the modified value is not available outside the function.
  - if **object is mutable** then modified value is available outside the function.

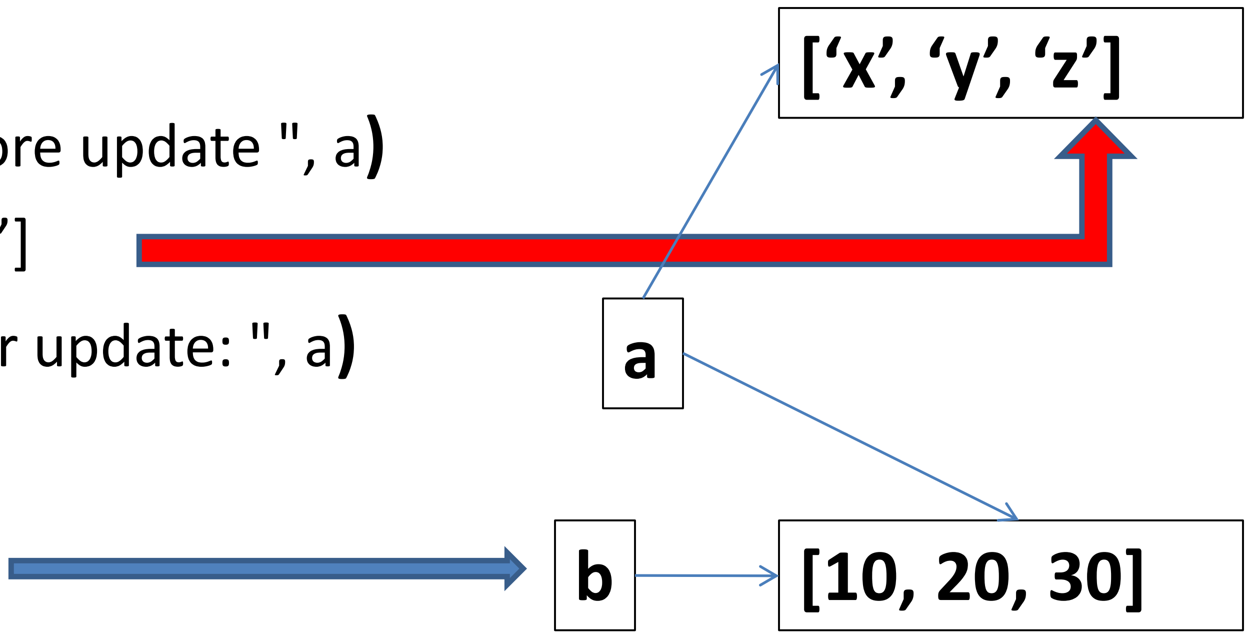
Pass by Object Reference

```
def fun( a ):
    print ("before update ", a)
    a[2] = 80
    print ("after update: ", a)

b = [10,20,30]
fun( b )
print ("Values outside the function: ", b)
```



Pass by Object Reference

```
def fun( a ):
    print ("before update ", a)
    a = [‘x’, ‘y’, ‘z’]
    print ("after update: ", a)

b = [10,20,30]
fun( b )
print ("Values outside the function: ", b)
```



Introduction to PYTHON

Function Arguments

Required/ Positional arguments

Keyword arguments

Default arguments

Variable-length arguments

By: Atul Kumar Uttam

Function Arguments

- A function can be called by using the following types of formal arguments
  - Required/ Positional arguments
  - Keyword arguments
  - Default arguments
  - Variable-length arguments

Required / Positional Arguments

- The arguments passed to a function in correct positional order.
- The number of arguments in the function call should match exactly with the function definition.

Required / Positional Arguments

```
def fun( a, b, c ):
        print (a, b, c )



fun(10, ‘hello’, 10.5)



Output: 10  ‘hello’  10.5
```

Keyword Arguments

- The caller identifies the arguments by the parameter name.

- This allows you to
  - place the arguments out of order

- Python interpreter is able to use the keywords provided to match the values with parameters.

Keyword Arguments

```
def fun(x , y , z ):
        print (x , y , z )



fun(x = 10, y = ‘hello’, z = 10.5)
fun(y = ‘hello’, z = 10.5, x = 10)
Output: 10  ‘hello’  10.5
```

Default Arguments

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

```
>>>def fun(x = 0 , y = 0 , z = 0 ):
        print (x , y , z )



>>>fun(100, 200, 300)
>>>fun(100, 200)
>>>fun(100)
>>>fun()
```

Variable-length Arguments

- You may need to process a function for more arguments than you specified while defining the function.
- Function can receive any number of arguments
- An asterisk (*) is placed before the variable name that holds the values of all **non keyword variable arguments**.
- This tuple remains empty if no additional arguments are specified during the function call.

Variable-length Arguments

```
def function_name([formal_args,] *var_args_tuple ):
    function_stmt
    return [expression]
```

Variable-length Arguments

```
>>>def fun(*a):
        print(a)

>>>fun(10, 20, 30)
(10, 20, 30)

>>>fun(10,20)
(10, 20)

>>>fun(10)
(10,)

>>>fun()
()
```

Variable-length Keyword Arguments

```
>>>def fun(**a):
        for i, j in a.items():
            print(i, j)

>>>fun(x = 10, y = 20, z = 30)
x 10
y 20
z 30
```

All in one Example

```
>>>p=100
>>>q=200
>>>def fun(a, b=0, *c, d, **e):
        print(a,b,c,d,e)

>>>fun(p, q, 1,2,3,4, d=999, x=300, y=400, z=500)
100 200 (1, 2, 3, 4) 999 {'x': 300, 'y': 400, 'z': 500}
```

Introduction to PYTHON

Anonymous / Lambda Function
map() filter() & reduce()

By: Atul Kumar Uttam

Anonymous/ Lambda Function

- No def keyword.
- Use lambda keyword to create anonymous functions
- It can have any number of arguments
- It can have only one expression
- It cannot contain any statements
- It returns a function object which can be assigned to any variable.

- 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.

- Mostly lambda functions are passed as parameters to a function which expects a function objects as parameter like map, reduce, filter functions

- The syntax of lambda function contains only a single statement, which is as follows-

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

```
>>>def add(x, y, z):
            return x + y + z

# Call the function
>>>add(2, 3, 4)
9
```

```
>>>a = lambda x, y, z : x+y+z

>>>a(10,20,30)
60
```

Use of Lambda Function

- We use lambda functions when we require a nameless function for a short period of time.
- In Python, we generally use it as an argument to a higher-order function (a function that takes in other functions as arguments).
- Lambda functions are used along with built-in functions like filter(), map() etc.

filter()

filter(function, iterable)

- creates a new list from the elements for which the function returns True.

```
# Program to filter out only the even items from a list
my_list = [1, 5, 4, 6, 8, 11, 3, 12]
new_list = list(filter(lambda x: (x%2 == 0) , my_list))
print(new_list)
```

```
Output: [4, 6, 8, 12]
```

map()

map(function, iterable)

- It applies a function to every item in the iterable.
- We can use map() to on a lambda function with a list:

```
>>>list = [1,2,3,4,5]
>>>squaredList = list(map(lambda x: x*x, list))
>>>print(squaredList)
```

reduce()

reduce(function, iterable)

- applies two arguments cumulatively to the items of iterable, from left to right.

```
>>>from functools import reduce
>>>list = [1,2,3,4,5]
>>>s = reduce(lambda x, y: x+y, list)
>>>print(s)
```

- In this case the expression is always true, thus it simply sums up the elements of the list.

Use of lambda function

```
>>>def fahrenheit(T):
    return ((float(9)/5)*T + 32)



>>>temp = [36.5, 37, 37.5,39]
>>>F = map(fahrenheit, temp)
>>>print(list(F))

[97.7, 98.600, 99.5, 102.2]
```

```
>>>temp = [36.5, 37, 37.5,39]
>>>F=map(lambda T:((float(9)/5)*T +32 ), temp)
>>>list(F)


[97.7, 98.600, 99.5, 102.2]
```

Introduction to PYTHON

Local, Global & Nonlocal Variables in Function

By: Atul Kumar Uttam

Global Vs Local variable

```
>>>x=100

>>>def fun():
        print(x)



>>>fun()

100
```

Global Vs Local variable

```
>>>x=100
>>>def fun():
        x=200
        print(x)


>>>fun()
200
>>>print(x)
100
```

Global Vs Local variable

```
>>>x=100
>>>def fun():
        x = x+10
        print(x)

>>>fun()
```

UnboundLocalError

Global Vs Local variable

```
>>>x=100
>>>def fun():
        global x
        x = x+10
        print(x)

>>>fun()
110
>>>print(x)
110
```

Nonlocal variable

```
>>>def fun1():
        a=100
        def fun2():
            nonlocal a
            a = 200
            print(a)
        fun2()
        print(a)

>>>fun1()
200
200
```

- The execution of a function introduces a new symbol table used for the local variables of the function.
- More precisely, all variable assignments in a function store the value in the local symbol table; whereas variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names.
- The symbol table is responsible for calculating the scope of every identifier in the code.