Programming Fundamental

Lab#06

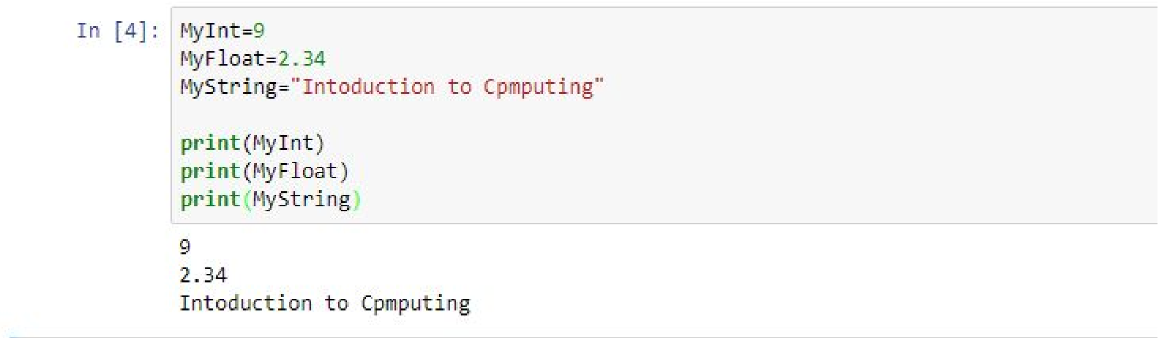
Contents

Variables & Expression

Variable

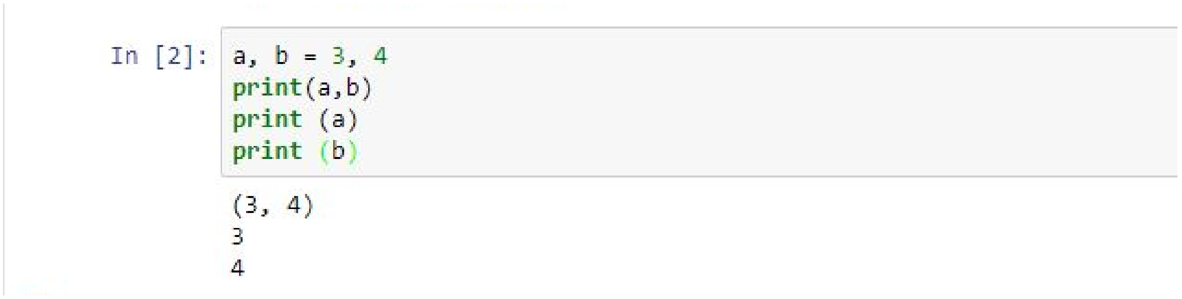
Python is completely object oriented, and not "statically typed".
You do not need to declare variables before using them, or declare their type.
Every variable in Python is an object.

MyInt=12
MyFloat=2.43
MyString=”Introduction to computing”



```
In [4]: MyInt=9
        MyFloat=2.34
        MyString="Intoduction to Cpmputing"

        print(MyInt)
        print(MyFloat)
        print(MyString)

9
2.34
Intoduction to Cpmputing
```

Assignments can be done on more than one variable "simultaneously" on the same line like this



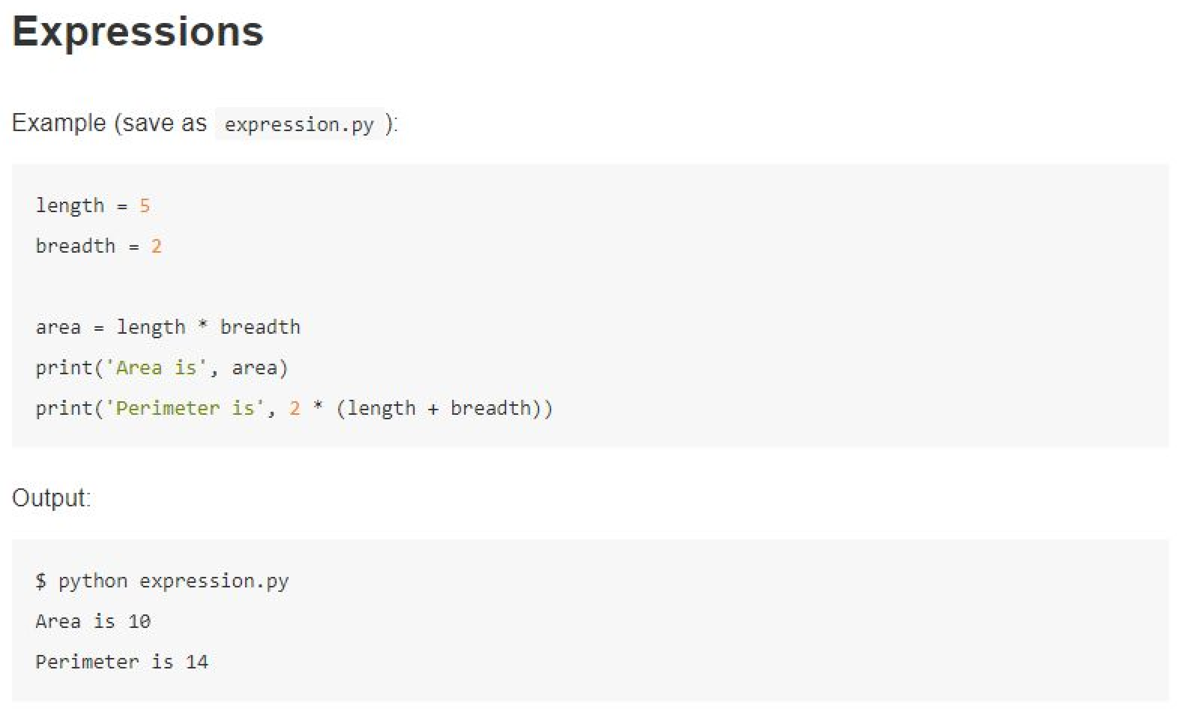
```
In [2]: a, b = 3, 4
        print(a,b)
        print (a)
        print (b)

(3, 4)
3
4
```

When we say variables are not statically typed, that is they are dynamically typed.

```
i = 42              # data type is implicitly set to integer

i = 42 + 0.11       # data type is changed to float

i = "forty"         # and now it will be a string
```

Expression

An expression is a combination of values, variables, operators, and calls to functions. Expressions need to be evaluated. If you ask Python to print an expression, the interpreter evaluates the expression and displays the result.



Expressions

Example (save as `expression.py`):

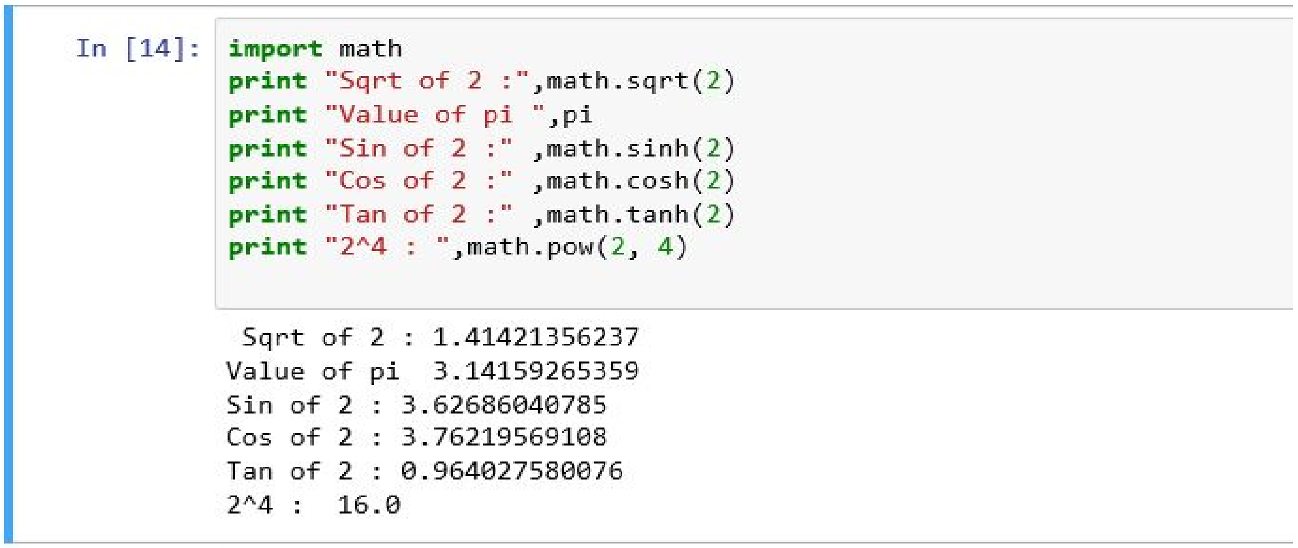
```
length = 5
breadth = 2

area = length * breadth
print('Area is', area)
print('Perimeter is', 2 * (length + breadth))
```

Output:

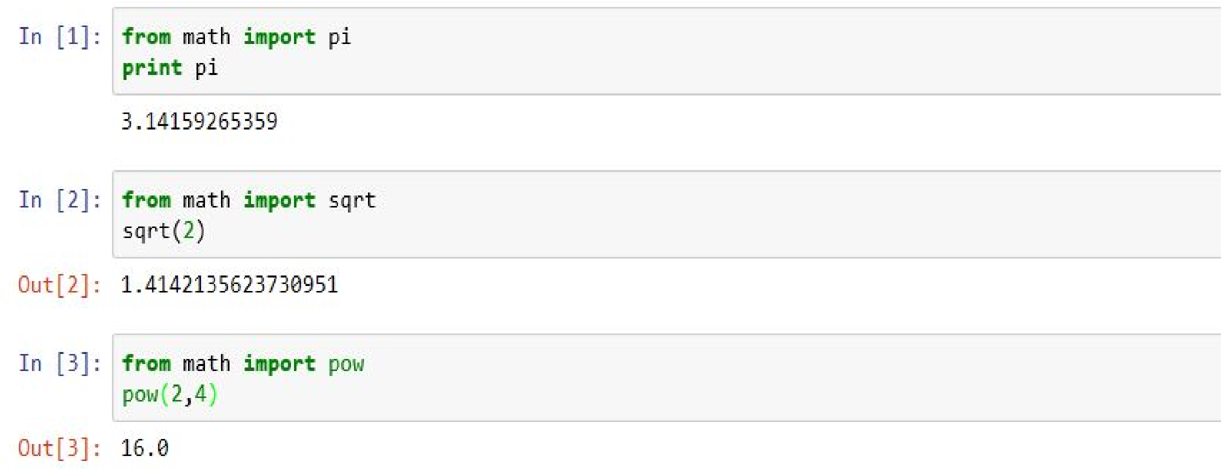
```
$ python expression.py
Area is 10
Perimeter is 14
```

Import Library

- Import math



```
In [14]: import math
         print "Sqrt of 2 :",math.sqrt(2)
         print "Value of pi ",pi
         print "Sin of 2 :" ,math.sinh(2)
         print "Cos of 2 :" ,math.cosh(2)
         print "Tan of 2 :" ,math.tanh(2)
         print "2^4 : ",math.pow(2, 4)

          Sqrt of 2 : 1.41421356237
         Value of pi  3.14159265359
         Sin of 2 : 3.62686040785
         Cos of 2 : 3.76219569108
         Tan of 2 : 0.964027580076
         2^4 :  16.0
```

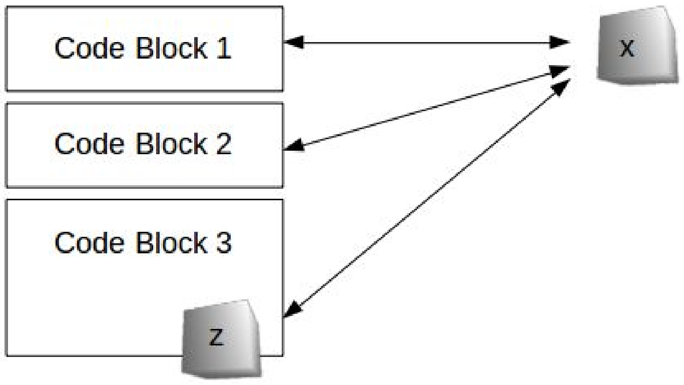


```
In [1]: from math import pi
        print pi

        3.14159265359

In [2]: from math import sqrt
        sqrt(2)

Out[2]: 1.4142135623730951

In [3]: from math import pow
        pow(2,4)

Out[3]: 16.0
```

Scope of variables

There are two types of variables: **global variables and local variables.**

A global variable can be reached anywhere in the code, a local only in the scope.



A **global variable (x**) can be reached and modified anywhere in the code, **local variable (z)** exists only in block 3.

Local variable

Local variables can only be reached in their scope.

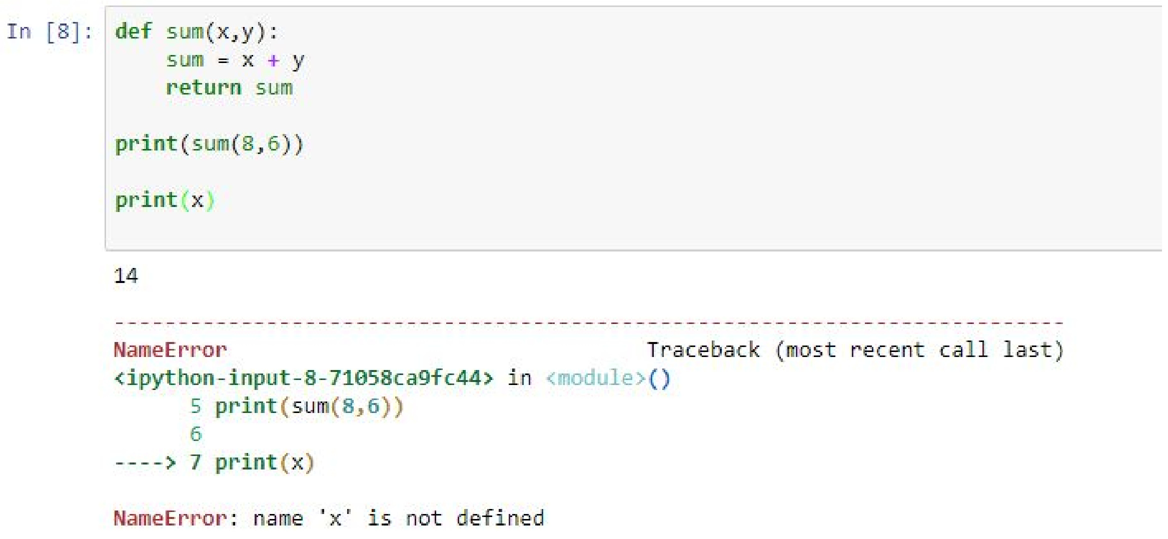
The example below has two local variables: x and y.

```
def sum(x,y):
    sum = x + y
    return sum

print(sum(8,6))
```

The variables x and y can only be used inside the function sum, they don’t exist outside of the function.

Local variables cannot be used outside of their scope, this line will not work:

```
print(x)
```



```
In [8]: def sum(x,y):
            sum = x + y
            return sum

        print(sum(8,6))

        print(x)

14

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-8-71058ca9fc44> in <module>()
      5 print(sum(8,6))
      6
----> 7 print(x)

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

Global variables

A global variable can be used anywhere in the code.

In the example below we define a global variable z

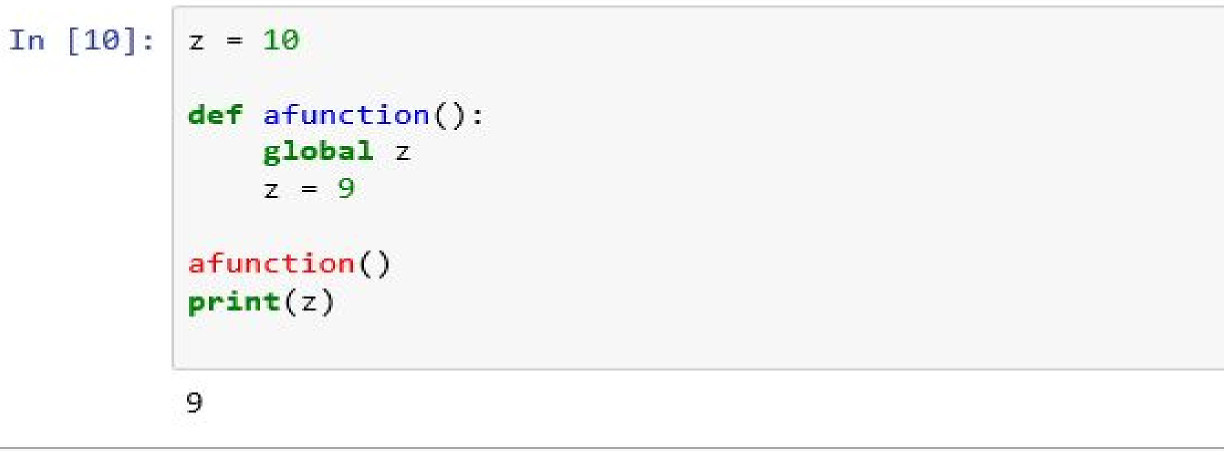
```
z = 10

def afunction():
    print(z)

afunction()
```

The global variable z can be used all throughout the program, inside functions or outside.

A global variable can modify inside a function and change for the entire program:

```
z = 10

def afunction():
    global z
    z = 9

afunction()
print(z)
```

After calling afunction(), the global variable is changed for the entire program.



```
In [10]: z = 10

         def afunction():
             global z
             z = 9

         afunction()
         print(z)

         9
```

Function

Function is a group of related statements that perform 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.

it avoids repetition and makes code reusable

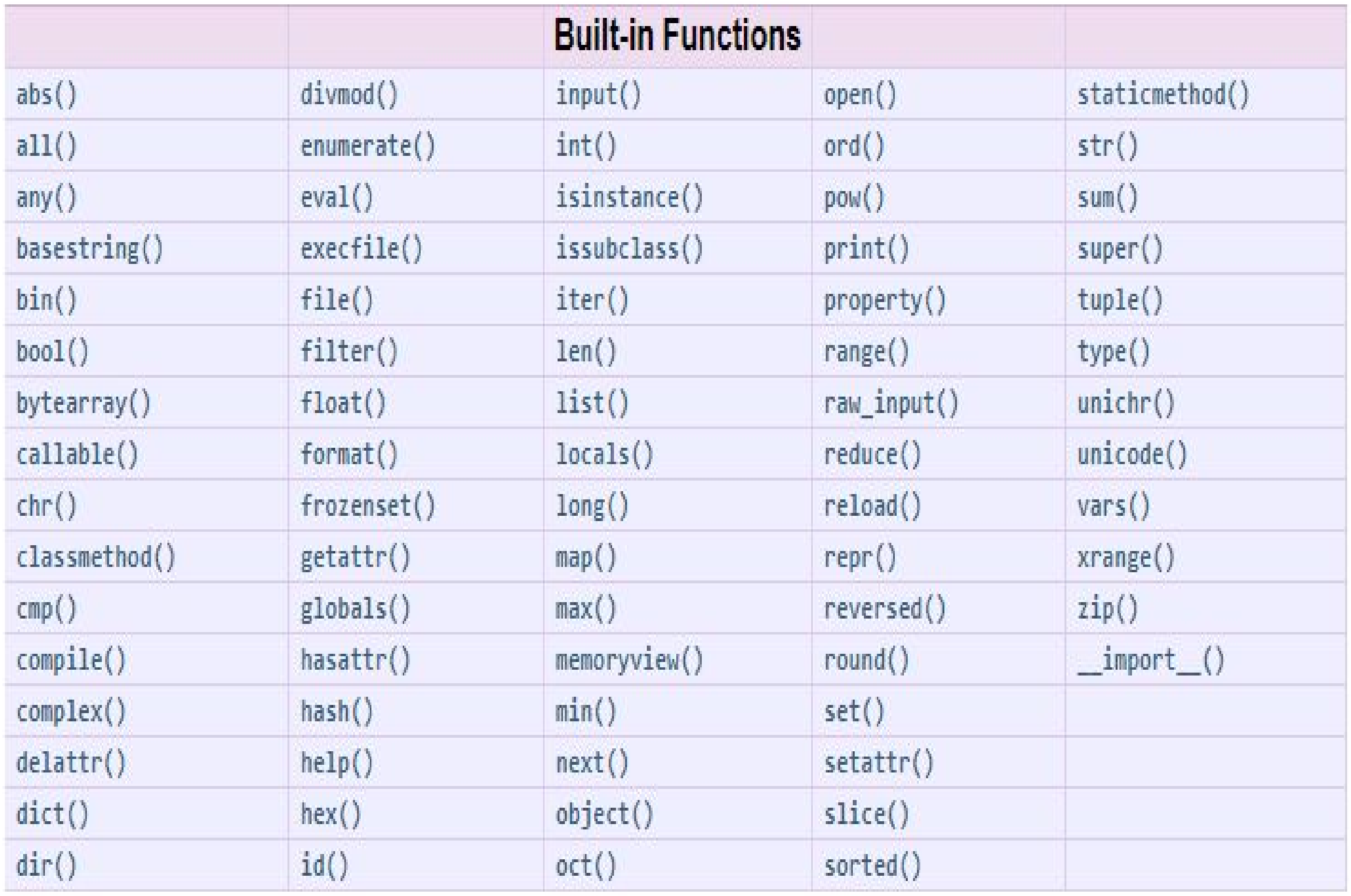
Types of function

Two types of function

1. Built-in functions
2. User define functions

Built-in Functions

The Python interpreter has a number of functions that are always available for use. These functions are called built-in functions



		Built-in Functions		
abs()	divmod()	input()	open()	staticmethod()
all()	enumerate()	int()	ord()	str()
any()	eval()	isinstance()	pow()	sum()
basestring()	execfile()	issubclass()	print()	super()
bin()	file()	iter()	property()	tuple()
bool()	filter()	len()	range()	type()
bytearray()	float()	list()	raw_input()	unichr()
callable()	format()	locals()	reduce()	unicode()
chr()	frozenset()	long()	reload()	vars()
classmethod()	getattr()	map()	repr()	xrange()
cmp()	globals()	max()	reversed()	zip()
compile()	hasattr()	memoryview()	round()	__import__()
complex()	hash()	min()	set()	
delattr()	help()	next()	setattr()	
dict()	hex()	object()	slice()	
dir()	id()	oct()	sorted()	

Reference

User-defined Function:

You can also create your own functions. These functions are called user-defined functions.

Rules to define a function

- 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**.
- 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.

Syntax

```
def functionname( parameters ):
 “function docstring”
 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")
```

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

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

Return multiple value

Working with the return statement is the fact that you can use it to return multiple values. To do this, you make use of **tuples**.

Tuples are **immutable**, which means that you can't modify any values that are stored in it.

You construct it with the help of double parentheses (). You can unpack tuples into multiple variables with the help of the comma and the assignment operator.

Example to understand how your function can return multiple values:

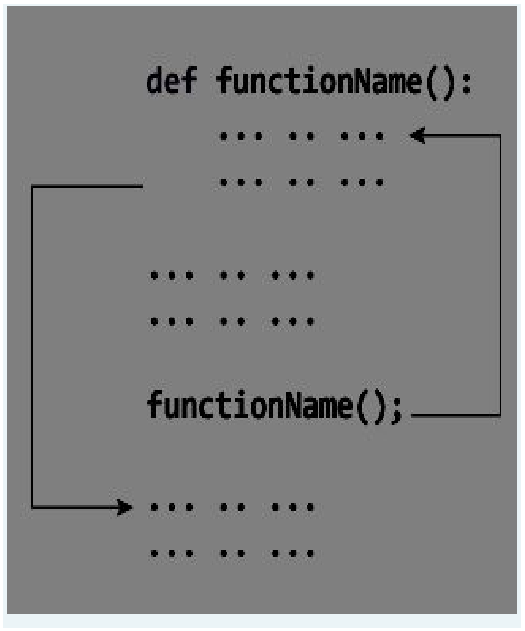
```
def plus( a,b ):
    add=a+b
    return (add,a,b);

# Now you can call function
Sum,a,b=plus(3,4)

print("Sum of",a, "&",b,"is:",Sum)
```

Output:

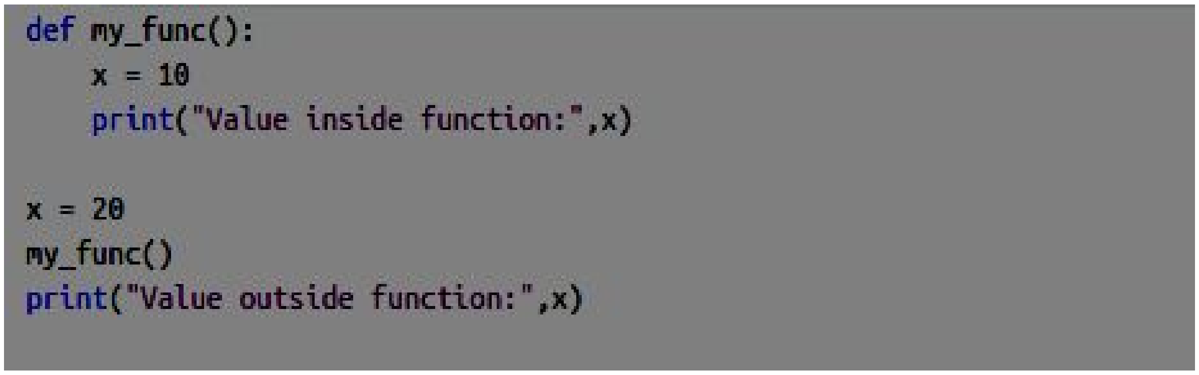
```
Sum of 3 & 4 is: 7
```

Note that the return statement return sum, a,b. would have the same result as return (sum, a,b)

Working of function



Scope and Lifetime of variables



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

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

Function Arguments

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

- Default arguments
- Required arguments
- Keyword arguments
- Variable-length arguments

Default Arguments

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

```
#!/usr/bin/python
# 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" )
```

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

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

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 –

```
#!/usr/bin/python

# 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 –

```
#!/usr/bin/python

# 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" )
```

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

```
Name:  miki
Age  50
```

Variable-length arguments

In cases where you don’t know the exact number of arguments that you want to pass to a function, you can use the following syntax with *args:

```
#!/usr/bin/python

# Function definition is here
def plus(*arg):
   "sum”
   Return sum(arg);

# Now you can call function
print plus(1,4,5)
```

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

```
10
```

Reference

- Functions

-