

## User define functions

We have functions in our previous topic, Python supports three types of functions, built in function, function define in module and user define function, In this section we will discuss the user define function in detail.

The functions define by programmer in regular Python code are called user define functions. It has certain advantages:

The most important reason to use functions is to make program handling easier as only a small part of the program is dealt with at a time, thereby avoiding ambiguity. Another reason to use functions is to reduce program size. Functions make program more readable and understandable.

### Structure of Python Program

In Python program, generally all function definitions are given at the top followed by statements which are not part of any functions. These statements are not indented at all. These are often called from the top level statement. The Python interpreter starts the execution of a program / script from top level statements. The top level statements are part of the main program. **Internally Python gives a special name to top – level statement as `__main__`.**

The structure of Python program is generally like the one shown below:

```
def function1():
    :
def function2():
    :
def function3():
    :
# top level statement here
s1
s2
.
.
```

Python stores this name in a built-in variable called `__name__`. (i.e. you need not to declare this variable; you can directly use it.) You can see it yourself. In the `__main__` segment of your program if you give a statement like this

```
print __name__
def greet():
    print ("Hi there")
print("At the top most level right now")
print("Inside")
```

```
print(__name__)
```

Upon executing above program, Python will display:

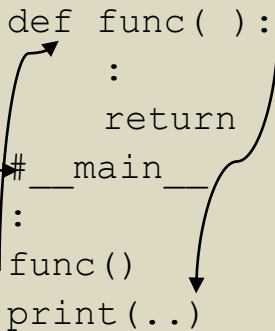
```
At the top most level right now
```

```
Inside __main__
```

## Flow of Execution

Whenever a function call statement is encountered, an execution frame, for the called function is created and the control(program control) is transferred to it. Within the function execution frame, the statement in the function body are executed, and with the return statement or the last statement of function body, the control returns to the statement wherefrom the function was called.

```
def func( ):
    :
    return
# __main__
:
func()
print(..)
```



Program execution begin with first statement if `__main__` segment. (def statements are also read but ignored until called.)

if we give line number to each line in the program then flow of execution can be represented just through the line numbers e.g.

```
1. # program add.py to add two numbers through a
   #function
2. def calSum(x, y):
3.     s = x + y
4.     return s
5.
6. num1 = float(input("Enter first number :"))
7. num2 = float(input("Enter second number :"))
8. sum = calSum(num1, num2)
9. print("Sum of two given number is ", sum)
```

As per above discussion the flow of execution for above program can also be represented as follows:

2 → 6 → 7 → 8 → 3 → 4 → 9

## Arguments and Parameters

As you can see that there are values being passed (through function call) and values being received (in function definition). Let us define these two types of values more formally

**argument** : Python refers to the values being passed as arguments and

**parameter**: values being received as parameters.

So you can say that arguments appear in function call statement and parameters appear in function header.

Argument in Python can be one of these value types:

- literals
- variables
- expressions

But the parameters in Python have to be some names i.e. variables to hold incoming values.

The alternative names for argument are actual parameter and actual argument.

Alternative names for parameters are formal parameters and formal argument.

Thus for a function as defined below:

```
def multiply(a, b):
    print( a * b)
multiply(3, 4)    # both literal argument
p = 9
multiply(p, 5)    # one literal and one variable
argument
multiply(p, p+1)  # one variable and one expression
```

But a function header like the one shown below is invalid:

```
def multiply(a+1, b):
    :

'''Error!! A function header cannot have
expressions. It can have just names or identifiers
to hold the incoming values.'''
```

## Passing Parameters

Python supports three types of formal argument / parameters:

1. Positional argument(Required arguments)
2. Default argument
3. Keyword( or named) argument

## Positional/Required Arguments

Till now you have seen that when you create a function call statement for given function definition, you need to match the number of arguments with number of parameters required. for example, if a function definition header is like:

```
def check(a, b, c):  
    :
```

then possible function calls for this can be:

```
check(x, y, z)  
check(2, x, y)  
check(2, 5, 7)
```

See, in all the above function calls, the number of passed values(arguments) has matched with the number of received values (parameters). Also, the values are given position-wise or order-wise i.e. the first parameter receives the value of first argument, second parameter, the value of second argument and so on e.g.

In function call1 above:

```
a will get value of x  
b will get value of y  
c will get value of z
```

Thus, through such function calls,

- the arguments must be provided for all parameters(Required)
- the values of arguments are matched with parameters, position(order) wise.

This way of parameter and argument specification is called Positional arguments or Required arguments or Mandatory arguments as no value can be skipped from the function call or you cannot change the order e.g. you cannot assign value of first argument to third parameter.

## Default Argument

Python allows us to assign default value(s) to a function's parameter(s) which is useful in case a matching argument is not passed in the function call statement. The default values are specified in the function header of function definition.

```
def interest(principal, time, rate = 0.10):
    :
```

The default value is specified in a manner syntactically similar to a variable initialization. The above function declaration provides a default value of 0.10 to the parameter rate. Now, if any function call appears as follows:

```
st_int = interest(5400, 2)    # third argument missing
```

then the value 5400 is passed to the parameter principal, the value 2 is passed to the second parameter time and since the third argument rate is missing, its default value 0.10 is used for rate.

But if the call provides all three arguments

```
st_int = interest(5400, 2, 0.15) # no argument is missing
```

that means the default values are considered only if no value is provided for that parameter in the function call statement.

One very important thing you must know about default parameter is :

In a function header, any parameter cannot have a default value unless all parameter appearing on its right have their default values.

```
def interest(prin, time, rate = 0.10) :    # legal
def interest(prin, time=2, rate): #illegal (default
                                # argument before
                                # required parameter
def interest(prin = 2000, time =2, rate)    #illegal
def interest(prin, time=2, rate=0.10)    # legal
def interest(prin=200, time=2, rate=0.10) #legal
```

Default arguments are useful in situations where some parameters always have same value. Also they provides greater flexibility to the programmers.

### Keyword (Named) Arguments

The default arguments give you flexibility to specify the default value for a parameter so that it can be skipped in the function call, if needed. However, still you cannot change the order of the arguments in the function call; you have to remember the correct order of the arguments.

Python offers a way of writing function calls where you can write any argument in any order provided you name the arguments when calling the function, as shown below:

```
interest(prin = 2000, time = 2, rate = 0.10)
```

```
interest(time=4, prin = 2600, rate = 0.09)
```

```
interest(time=2, rate=0.12, prin=2000)
```

All the above function calls are valid now, even if the order of argument does not match the order of parameters as defined in the function header.

This way of specifying names for the values being passed, in the function call is known as **keyword argument**.

**In keyword argument function call the name of argument should be same as parameter define in function header** otherwise it will gives an error.

### Using Multiple Argument Type Together

Python allows you to combine multiple argument types in a function call. Rules for combining all three types of arguments

Python states that in a function call statement:

- an argument list must first contain positional argument followed by any keyword argument.
- Keyword argument should be taken from the required argument preferably.
- You cannot specify a value for an argument more than once.

For instance, consider the following function header:

```
def interest(prin, cc, time=2, rate = 0.09):
    return prin * time * rate
```

The values for parameters prin and cc can be provided either as positional arguments or as keyword argument but these values cannot be skipped from the function call.

interest(prin=3000, cc=5)	Legal	
Interest(rate=0.12, prin=5000, cc=4)	Legal	
interest(cc=4, rate = 0.12, prin=5000)	Legal	
interest(5000, 3, rate = 0.05)	Legal	
interest(rate=0.05, 5000, 3)	Illegal	Keyword argument before Positional argument
interest(5000, prin = 300, cc=2)	Illegal	Multiple values for prin
interest(5000, principal = 300, cc=2)	Illegal	Undefined name used
Interest(500, time=2, rate=0.05)	Illegal	Required args is missing

### Types of functions

There can be broadly two types of functions in Python:

- Function returning some value(non void function)



- Function not returning any value(void function)

## 1. Function returning some value(Non void function)

The functions that return some computed result in terms of a value fall in this category.

Syntax `return <value>`

The value being return can be a literal, variable, an expression

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

And we are invoking this function as

```
result = sum(5, 3)
```

Important Points about return

- The return value of a function should be used in the caller function/program inside an expression or a statement.

```
print(sum(3, 4))
sum(3, 4) > 6
```

- If you do not use their value in any of these ways and just give a stand alone function call, Python will not report an error but their return value is completely wasted.
- The return statement ends a function execution even if it is in the middle of the function. A function ends the moments it reaches a return statement or all statements in function –body have been executed whoever occurs earlier.

```
def check(a):
    a = math.fabs(a)
    return a
    print(a)
check(-15)
```

The statement is unreachable

## 2. Function not returning any value(Void functions)

The function that perform some action or do some work but do not return any computed value of final value to the caller are called void functions. A void function may or may not have return statement. If a void function has a return statement, then it takes the following form:

```
return
```

that is, keyword return without any value or expression.



for e.g.

```
def greet():  
    print("helloz")
```

```
def greet1(name):  
    print("hello", name)
```

```
def quote():  
    print("Goodness counts!!")  
    return
```

```
def prinSum(a, b, c):  
    print("Sum is ")  
    print(a + b + c)  
    return
```

The void functions are generally not used inside a statement or expression in the caller; their function call statement is standalone complete statement itself, e.g.

```
greet()  
greet1("ABC")  
quote()  
prinSum(4, 6 , 5)
```

The void functions do not return a value but they return a legal empty value of Python i.e. None. Every void function returns value None to its caller. So if need arises you can assign this return value somewhere as per your need.

```
def greet():  
    print("helloz")  
a = greet()  
print(a)
```

The above program will give the output as :

helloz

None