

PYTHON PROGRAMMING (18MCA31C)

UNIT – II Functions

FACULTY:

Dr. R. A. Roseline, M.Sc., M.Phil., Ph.D.,

Associate Professor and Head,
Post Graduate and Research Department of Computer Applications,
Government Arts College (Autonomous), Coimbatore – 641 018.



Functions




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

- ▶ 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.
- ▶ 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]`



- ▶ Syntax:

- ▶ `def functionname(parameters):`

- ▶ `"function_documentation"`

- ▶ `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:

- ▶ `def printfunc(str):`

- ▶ `"This prints a string passed to the function"`

- ▶ `print str`

- ▶ `return`

Calling a Function

- Following is the example to call printfunc() function:

```
def printfunc( str ): "This is a print function"  
    print str;  
    return;
```

```
printfunc(" Welcome to Python");  
printfunc(" MCA at GAC,CBE");
```

- This would produce following result:
- Welcome to Python
- MCA at GAC,CBE

Pass by reference vs value

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. For example:

```
def changeme( mylist ): "This changes a passed list"
    mylist.append([1,2,3,4]);
    print "Values inside the function: ", mylist
    return
mylist = [10,20,30];
changeme( mylist );
print "Values outside the function: ", mylist
```

► So this would produce following result:

```
Values inside the function: [10, 20, 30, [1, 2, 3, 4]]
```

```
Values outside the function: [10, 20, 30, [1, 2, 3, 4]]
```



There is one more example where argument is being passed by reference but inside the function, but the reference is being over-written.

```
def changeme( mylist ): "This changes a passed list"
    mylist = [1,2,3,4];
    print "Values inside the function: ", mylist
    return
mylist = [10,20,30];
changeme( mylist );
print "Values outside the function: ", mylist
```

→ The parameter mylist is local to the function changeme. Changing mylist within the function does not affect mylist. The function accomplishes nothing and finally this would produce following result:

```
Values inside the function: [1, 2, 3, 4]
```

```
Values outside the function: [10, 20, 30]
```

Function Arguments:

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.

```
def printfunc( str ): "This prints a passed string"  
    print str;  
    return;  
printfunc();
```

- ▶ This would produce following result:

```
Traceback (most recent call last):  
File "test.py", line 11, in <module> printfunc();  
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.

```
def printfunc( str ): "This prints a passed  
    string"  
    print str;  
    return;  
printfunc( str = "MCA at GAC");
```

- This would produce following result:

```
MCA at GAC
```



Following example gives more clear picture. Note, here order of the parameter does not matter:

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

➔ This would produce following result:

```
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.
- Following example gives idea on default arguments, it would print default age if it is not passed:

```
def printinfo( name, age = 15 ): "Test function"  
    print "Name: ", name;  
    print "Age ", age;  
    return;  
  
printinfo( age=20, name="ANNRIA" );  
printinfo( name="ANNRIA" );
```

- This would produce following result:

```
Name: ANNRIA Age 15 Name: ANNRIA Age 20
```



Variable-length arguments:

- ▶ You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments.
- ▶ The general syntax for a function with non-keyword variable arguments is this:

```
def functionname([formal_args,] *var_args_tuple ):  
    "function_docstring"  
    function_suite  
    return [expression]
```

- ▶ An asterisk (*) is placed before the variable name that will hold the values of all nonkeyword variable arguments.
- ▶ This tuple remains empty if no additional arguments are specified during the function call. For example:

```
def printinfo( arg1, *vartuple ):  
    "This is test"  
    print "Output is: "  
    print arg1  
    for var in vartuple:  
        print var  
    return;  
printinfo( 10 );  
printinfo( 70, 60, 50 );
```

- ▶ This would produce following result:

```
Output is:  
10  
Output is:  
70  
60  
50
```

RECURSIVE FUNCTIONS

In Python, a function can call other functions. It is even possible for the function to call itself. These types of construct are termed as Recursive functions.

```
def factorial(x):  
    """This is a recursive function to find the factorial of an integer"""  
    if x == 1:  
        return 1  
    else:  
        return (x * factorial(x-1))  
  
num = 3  
  
print("The factorial of", num, "is", factorial(num))
```

The *Anonymous* Functions:

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

- ▶ 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:**

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

Example:

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

```
sum = lambda arg1, arg2: arg1 + arg2;
print "Value of total : ", sum( 10, 20 )
print "Value of total : ", sum( 20, 20 )
```

- This would produce following result:

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


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 whereas 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.



Example:

```
total = 0; # This is global variable.
def sum( arg1, arg2 ):
    "Add both the parameters"
    total = arg1 + arg2;
    print "Inside the function local total : ", total
    return total;

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

➤ This would produce following result:

```
Inside the function local total : 30
```

```
Outside the function global total : 0
```



Thank you

The Content in this Material are from the Textbooks and Reference books given in the Syllabus