FUNCTIONS IN PYTHON

Prof. Juilee Mahajan

Introduction

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.

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.

•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.

Example:

```
# 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
```

Example 2 :

```
# A simple Python function to check whether x is even or odd

def evenOdd( x ):
    if (x % 2 == 0):
        print "even"
    else:
        print "odd"

# Driver code
evenOdd(2)
evenOdd(3)
```

Output:

```
even
odd
```

Function Arguments

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

1) Required arguments
2) Keyword arguments
3) Default arguments
4) 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.

Example:

```
# 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()
```

Output:

```
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 −

Example 1:

```
# 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
```

Example 2:

```
# 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
```

Note: The order of parameters does not matter.

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 −

Example:

```
# 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 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.

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 holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call.

Example:

```
# Function definition is here
def printinfo( arg1, *vartuple ):
    "This prints a variable passed arguments"
      print "Output is: "
      print arg1
      for var in vartuple:
          print var
     return;

# Now you can call printinfo function
printinfo( 10 )
printinfo( 70, 60, 50 )
```

Output:

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

The Anonymous Functions

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

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.

Example:

```
def cube(y):
    return y*y*y;

g = lambda x: x*x*x
print(g(7))

print(cube(5))
```

Output:

```
343
125
```

Use of lambda() with filter()

The filter() function in Python takes in a function and a list as arguments. This offers an elegant way to filter out all the elements of a sequence “sequence”, for which the function returns True. Here is a small program that returns the odd numbers from an input list:

```
# Python code to illustrate filter() with lambda()
li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]
final_list = list(filter(lambda x: (x%2 != 0) , li))
print(final_list)
```

Output:

```
[5, 7, 97, 77, 23, 73, 61]
```

Use of lambda() with map()

The map() function in Python takes in a function and a list as argument. The function is called with a lambda function and a list and a new list is returned which contains all the lambda modified items returned by that function for each item.

Example:

Python code to illustrate map() with lambda() to get double of a list.

```
li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]
final_list = list(map(lambda x: x*2 , li))
print(final_list)
```

Output:

[10, 14, 44, 194, 108, 124, 154, 46, 146, 122]

Use of lambda() with reduce()

The reduce() function in Python takes in a function and a list as argument. The function is called with a lambda function and a list and a new reduced result is returned. This performs a repetitive operation over the pairs of the list. This is a part of functools module.

Example:

```
# Python code to illustrate reduce() with lambda() to get sum of a list

from functools import reduce
li = [5, 8, 10, 20, 50, 100]
sum = reduce((lambda x, y: x + y), li)
print (sum)
```

Output:

193

Returning Multiple Values in Python

1) Using Object:

This is similar to C/C++ and Java, we can create a class (in C, struct) to hold multiple values and return an object of the class.

```
# A Python program to return multiple values from a method using class

class Test:
    def __init__(self):
        self.str = “example"
        self.x = 20

# This function returns an object of Test
def fun():
    return Test()

# Driver code to test above method
t = fun()
print(t.str)
print(t.x)
```

Output:

```
example
20
```

2) Using Tuple: A Tuple is a comma separated sequence of items. It is created with or without (). Tuples are immutable.

```
# A Python program to return multiple values from a method using tuple

# This function returns a tuple
def fun():
    str = “func_example"
    x   = 20
    return str, x;  # Return tuple, we could also
                    # write (str, x)

# Driver code to test above method
str, x = fun() # Assign returned tuple
print(str)
print(x)
```

Output:

```
func_example
20
```

3) Using a list: A list is like an array of items created using square brackets. They are different from arrays as they can contain items of different types. Lists are different from tuples as they are mutable.

```
# A Python program to return multiple values from a method using list

# This function returns a list
def fun():
    str = “func_example"
    x = 20
    return [str, x];

# Driver code to test above method
list = fun()
print(list)
```

Output:

```
[' func_example', 20]
```

4) Using a Dictionary: A Dictionary is similar to hash or map in other languages.

```
# A Python program to return multiple values from a method using dictionary

# This function returns a dictionary
def fun():
    d = dict();
    d['str'] = “Func_Example"
    d['x']   = 20
    return d

# Driver code to test above method
d = fun()
print(d)
```

Output:

```
{'x': 20, 'str': ' Func_Example'}
```

THANK YOU!!