USER DEFINED FUNCTIONS

Introduction

- A function is a set of statements that take inputs, do some specific computation and produce output.
- The idea is to put some commonly or repeatedly done tasks together and make a function so that instead of writing the same code again and again for different inputs, we can call the function.
- Functions that readily comes with Python are called built-in functions. Python provides built-in functions like `print()` , etc. but we can also create your own functions. These functions are known as **user defines functions**.

User defined functions

- All the functions that are written by any us comes under the category of user defined functions.
- Below are the steps for writing user defined functions in Python.
    1. In Python, def keyword is used to declare user defined functions.
    2. An indented block of statements follows the function name and arguments which contains the body of the function.

Syntax:

```
def function_name():
        statements
        .
        .
```

Example:

```
# Python program to
# demonstrate functions

# Declaring a function
def fun():
    print("Inside function")

# Driver's code
# Calling function
fun()
```

Parameterized Function

- The function may take arguments(s) also called parameters as input within the opening and closing parentheses, just after the function name followed by a colon.

Syntax:

```
def function_name(argument1, argument2, ...):
        statements
        .
        .
```

Example:

```
# Python program to
# demonstrate functions

# 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)
```

Default arguments

A default argument is a parameter that assumes a default value if a value is not provided in the function call for that argument.The following example illustrates Default arguments.

Example:

```
# Python program to demonstrate
# default arguments
def myFun(x, y = 50):
    print("x: ", x)
    print("y: ", y)

# Driver code (We call myFun() with only
# argument)
myFun(10)
```

Keyword arguments

The idea is to allow caller to specify argument name with values so that caller does not need to remember order of parameters.

Example:

```
# Python program to demonstrate Keyword Arguments
def student(firstname, lastname):
    print(firstname, lastname)


# Keyword arguments
student(firstname ='Geeks', lastname ='Practice')
student(lastname ='Practice', firstname ='Geeks')
```

We need to keep the following points in mind while calling functions:

1. In the case of passing the keyword arguments, the order of arguments is not important.
2. There should be only one value for one parameter.
3. The passed keyword name should match with the actual keyword name.
4. In the case of calling a function containing non-keyword arguments, the order is important.

Example #1: **Calling functions without keyword arguments**

```
def student(firstname, lastname ='Mark', standard ='Fifth'):
    print(firstname, lastname, 'studies in', standard, 'Standard')

# 1 positional argument
student('John')

# 3 positional arguments
student('John', 'Gates', 'Seventh')

# 2 positional arguments
student('John', 'Gates')
student('John', 'Seventh')
```

Example #2: **Calling functions with keyword arguments**

```
def student(firstname, lastname ='Mark', standard ='Fifth'):
    print(firstname, lastname, 'studies in', standard, 'Standard')

# 1 keyword argument
student(firstname ='John')

# 2 keyword arguments
student(firstname ='John', standard ='Seventh')

# 2 keyword arguments
student(lastname ='Gates', firstname ='John')
```

Example #3: **Some Invalid function calls**

```
def student(firstname, lastname ='Mark', standard ='Fifth'):
    print(firstname, lastname, 'studies in', standard, 'Standard')

# required argument missing
student()

# non keyword argument after a keyword argument
student(firstname ='John', 'Seventh')

# unknown keyword argument
student(subject ='Maths')
```

The above code will throw an error because:

- In the first call, value is not passed for parameter *firstname* which is the required parameter.
- In the second call, there is a non-keyword argument after a keyword argument.
- In the third call, the passing keyword argument is not matched with the actual keyword name arguments.

Function with return value

- Sometimes we might need the result of the function to be used in further process. Hence, a function should also returns a value when it finishes it's execution. This can be achieved by return statement.
- A return statement is used to end the execution of the function call and "returns" the result (value of the expression following the return keyword) to the caller. The statements after the return statements are not executed. If the return statement is without any expression, then the special value None is returned.

Syntax:

```
def fun():
        statements
        .
        .
        return [expression]
```

Example:

```
# Python program to
# demonstrate return statement

def add(a, b):

    # returning sum of a and b
    return a + b

def is_true(a):

    # returning boolean of a
    return bool(a)

# calling function
res = add(2, 3)
print("Result of add function is {}".format(res))

res = is_true(2<5)
print("\nResult of is_true function is {}".format(res))
```

Using mutable objects as default argument values in python

This must be done very carefully. The reason is the default values of the arguments are evaluated only once when the control reaches the function.

Definition for the first time. After that, the same values(or mutable objects) are referenced in the subsequent function calls.

```
def appendItem(itemName, itemList = []):
    itemList.append(itemName)
    return itemList


print(appendItem('notebook'))
print(appendItem('pencil'))
print(appendItem('eraser'))
```

Example using dictionary

```
# mutable default argument values example using python dictionary

# itemName is the name of item and quantity is the number of such
# items are there


def addItemToDictionary(itemName, quantity, itemList = {}):
    itemList[itemName] = quantity
    return itemList


print(addItemToDictionary('notebook', 4))
print(addItemToDictionary('pencil', 1))
print(addItemToDictionary('eraser', 1))
```

Example : Using None as default argument

```
# using None as values of the default arguements

print('#list')
def appendItem(itemName, itemList=None):
    if itemList == None:
          itemList = []
    itemList.append(itemName)
    return itemList


print(appendItem('notebook'))
print(appendItem('pencil'))
print(appendItem('eraser'))


# using None as value of default parameter

print('\n\n#dictionary')
def addItemToDictionary(itemName, quantity, itemList = None):
    if itemList == None:
        itemList = {}
    itemList[itemName] = quantity
    return itemList


print(addItemToDictionary('notebook', 4))
print(addItemToDictionary('pencil', 1))
print(addItemToDictionary('eraser', 1))
```

Variable length arguments

We can have both normal and keyword variable number of arguments.

The special syntax *args in function definitions in Python is used to pass a variable number of arguments to a function. It is used to pass a non-keyworded, variable-length argument list.

The special syntax **kwargs in function definitions in python is used to pass a keyworded, variable-length argument list. We use the name kwargs with the double star. The reason is because the double star allows us to pass through keyword arguments (and any number of them).

Example:

```
# Python program to illustrate
# *args and **kwargs
def myFun1(*argv):
    for arg in argv:
        print (arg)

def myFun2(**kwargs):
    for key, value in kwargs.items():
        print ("% s == % s" %(key, value))

# Driver code
print("Result of * args: ")
myFun1('Hello', 'Welcome', 'to', 'GeeksforGeeks')

print("\nResult of **kwargs")
myFun2(first ='Geeks', mid ='for', last ='Geeks')
```

Pass by Reference or pass by value?

- One important thing to note is, in Python every variable name is a reference. When we pass a variable to a function, a new reference to the object is created. Parameter passing in Python is same as reference passing in Java. To confirm this Python's built-in id() function is used in below example.

Example:

```
# Python program to
# verify pass by reference

def myFun(x):
    print("Value recieved:", x, "id:", id(x))

# Driver's code
x = 12
print("Value passed:", x, "id:", id(x))
myFun(x)
```

Recursion in Python

- it is a process in which a function calls itself directly or indirectly.

Advantages of using recursion

- A complicated function can be split down into smaller sub-problems utilizing recursion.
- Sequence creation is simpler through recursion than utilizing any nested iteration.
- Recursive functions render the code look simple and effective.

Disadvantages of using recursion

- A lot of memory and time is taken through recursive calls which makes it expensive for use.
- Recursive functions are challenging to debug.
- The reasoning behind recursion can sometimes be tough to think through.

Syntax:

```
def func(): <--
              |
              | (recursive call)
              |
       func() ----
```

```
# Program to print factorial of a number
# recursively.

# Recursive function
def recursive_factorial(n):
   if n == 1:
      return n
   else:
      return n * recursive_factorial(n-1)

# user input
num = 6

# check if the input is valid or not
if num < 0:
   print("Invalid input ! Please enter a positive number.")
elif num == 0:
   print("Factorial of number 0 is 1")
else:
   print("Factorial of number", num, "=", recursive_factorial(num))
```

Python Lambda Functions

- In Python, an anonymous function means that a function is without a name.
- the *def* keyword is used to define a normal function in Python.
- Similarly, the *lambda* keyword is used to define an anonymous function in Python.
- It has the following syntax:

Syntax: lambda arguments: expression

- This function can have any number of arguments but only one expression, which is evaluated and returned.
- One is free to use lambda functions wherever function objects are required.
- You need to keep in your knowledge that lambda functions are syntactically restricted to a single expression.
- It has various uses in particular fields of programming besides other types of expressions in functions.

Example :

```
# Python code to illustrate cube of a number
# showing difference between def() and lambda().
def cube(y):
    return y*y*y

lambda_cube = lambda y: y*y*y

# using the normally
# defined function
print(cube(5))

# using the lamda function
print(lambda_cube(5))
```

- As we can see in the above example both the cube() function and lambda_cube() function behave the same and as intended. Let’s analyze the above example a bit more:

- **Without using Lambda:** Here, both of them return the cube of a given number. But, while using def, we needed to define a function with a name cube and needed to pass a value to it. After execution, we also needed to return the result from where the function was called using the *return* keyword.

- **Using Lambda:** Lambda definition does not include a “return” statement, it always contains an expression that is returned. We can also put a lambda definition anywhere a function is expected, and we don’t have to assign it to a variable at all. This is the simplicity of lambda functions.