Functions, Modules, and Packages

Introduction to Functions

A function is a block of organized and reusable program code that performs a specific, single, and well-defined task. A function provides an interface for communication in terms of how information is transferred to it and how results are generated.

Need for Functions

- Simplifies program development by making it easy to test separate functions.
- Understanding programs becomes easier.
- Libraries contain several functions which can be used in our programs to increase the productivity.
- By dividing a large program into smaller functions, different programmers can work on different functions.
- Users can create their own functions and use them in various locations in the main program.

Defining Functions

A function definition consists of a function header that identifies the function, followed by the body of the function. The body of a function contains the code that is to be executed when a function is called. To define a function, we have to remember following points:

- Function definition starts with the keyword def
- The keyword def is followed by the function name and parentheses.
- After parentheses, a colon (:) should be placed.
- Parameters or arguments that the function accepts should be placed inside the parentheses.
- A function might have a return statement.
- The function code should indented properly.

A function definition contains two parts:

- Function header
- Function body

The syntax of a function definition is as follows:

```
def  function_name(arg1, arg2, ...):
        ["""documentation string"""]
        statement block
        return [expression]
```

Example for defining a function without arguments is as follows:

```
def  printstars():
        for i in range(1,101):
                print("*", end=' ')
```

In the above function definition, *printstars* is the function name. It prints 100 stars in the same line.

Example for defining a function with arguments is as follows:

```
def  printstars(n):
        for i in range(1,n+1):
                print(“*”, end=‘ ‘)
```

Above function prints n number of stars when called. In Python, a function can be defined multiple times.

Calling a Function

Before calling a function, it should be defined. We call a function to execute the code inside the function. Syntax for a function call is as follows:

function_name([arg1, arg2, ...])

The arguments or parameters passed in a function call are called actual parameters. The arguments used in the function header of a function definition are called formal parameters. When a function is called, control moves to the first line inside the function definition. After the body of function definition is executed, control returns back to the next statement after the function call. Points to remember while calling a function:

- The function name and number of parameters must be same in the function call and function definition.
- When the number parameters passed doesn’t match with the parameters in the function definition, error is generated.
- Names of arguments in the function call and function definition can be different.
- Arguments can be passed as expressions. The expression will get executed first and then the value is passed to the formal parameter.
- The parameter list must be separated by commas.
- If the function returns a value it must be assigned to some variable in the calling function.

Let’s consider the following example which demonstrates function definition and function call:

```
#Function definition
def prod(x,y):
        return x*y

a = int(input())
b = int(input())
c = prod(a,b)   #Function call
print("Product of",a,"and",b,"is",c)
```

In the above example, a and b are actual parameters, while x and y are formal parameters.

Functions Returning Value

A function may or may not return a value. To return a value, we must use *return* statement in the function definition. Every function by default contains an implicit *return* statement as the last line which returns *None* object to the calling function. A return statement is used for two reasons:

Return a value to the caller.
To end the execution of a function and give control to caller.

The syntax of *return* statement is as follows:

return [expression]

Example of a function returning a value is as follows:

```
def cube(x):
        return x*x*x
```

In the above example, the function cube is returning the *cube* of value passed to it from caller.

Passing Arguments

If a function definition contains arguments, then in a function call there is a need to pass arguments. For example, in the previous function definition of *cube*, there is a single argument. So, while calling this function we need to pass a single parameter. The *cube* function can be called as follows:

cube(10)

In the above function call, 10 is the actual parameter.

Default Arguments

The formal parameters in a function definition can be assigned a default value. Such parameters to which default values are assigned are called default arguments. Default arguments allows a function call to pass less parameters than the number of parameters in the function definition.

Default value can be assigned to a parameter by using the assignment (=) operator. A function definition can have one or more default arguments. All the default arguments should be at the end of the arguments list. Following example demonstrates a function with default arguments and its usage:

```
#Function to calculate simple interest
def si(p, n, r=1): #r is default argument
        prod = p*n*r
        return prod/100

p = int(input())
n = int(input())
print("Simple interest with default rate is:",si(p,n))
print("Simple interest with 5% rate is:",si(p,n,5))
```

In the example program, si(p,n) call takes the default value of 1 for r. For si(p,n,5) call, it takes the value 5 for r.

Keyword Arguments

In general, when parameters are passed in a function call, they are assigned to formal parameters based on their position. Such parameters are called positional parameters. We can also pass the parameters based on the name (keyword) of the parameter in the function definition. Such parameters are called keyword arguments.

In the case of keyword arguments, the actual parameters are assigned to formal parameters based on their name. Keyword arguments can be used if you don't want to pass the parameters based on position of formal parameters. Points to remember when using keyword arguments:

- All keyword parameters must match one of the parameters in the function definition.
- Order of keyword arguments is not important.
- A value should not be passed more than once to a keyword parameter.

Following example demonstrates keyword arguments and their use:

```
#Function to calculate simple interest
def si(p, n, r):
        prod = p*n*r
        return prod/100

print("Simple interest is:",si(r=2, n=2, p=5000))
```

In the above example, we can see in the function call that we are passing using names (keywords) of formal parameters.

Variable-length Arguments

In some cases we cannot exactly tell how many parameters are needed in a function definition. In such cases we can use variable length arguments. Variable length arguments allows to pass random number of arguments in a function call. While creating a variable-length argument, the argument name must be preceded with * symbol. Points to remember when working with variable-length arguments:

- The random arguments passed to the function forms a tuple.
- A for loop can be used to iterate over and access the elements in the variable-length argument.
- A variable-length argument should be at the end of the parameters list in the function definition.
- Other formal parameters written after variable-length argument must be keyword arguments only.

Following example demonstrates variable-length argument:

```
def fun(name, *friendslist):
        print("Friends of",name,"are: ")
        for x in friendslist:
                print(x, end = ' ')

fun("Ramesh", "Mahesh", "Suresh")
```

Output of the above example is as follows:

```
Friends of Ramesh are:
Mahesh  Suresh
```

Scope of Variables

Variables in a program has two things:

- Scope: Parts of the program in which it is accessible.
- Lifetime: How long a variable stays in the memory.

Based on scope of a variable, there are two types of variables:

1. Global variables
2. Local variables

Following are the differences between a global variable and local variable:

Global Variable	Local Variable
Variable which is defined in the main body of the program file.	Variable which is defined inside a function.
Accessible throughout the program file.	Accessible from the point it is defined to the end of the block it is defined in.
Accessible to all functions in the program.	They are not related in any way to other variables outside the function.

Following example demonstrates local and global variables:

```
x = 20
def fun():
        x = 10
        print("Local x =",x)
fun()
print("Global x =",x)
```

Output of above code is as follows:

```
Local x = 10
Global x = 20
```

Global Statement

A local variable inside a function can be made *global* by using the global keyword. If a local variable which is made global using the *global* statement have the same name as another global variable, then changes on the variable will be reflected everywhere in the program.

Syntax of global statement is as follows:

```
global  variable-name
```

Following example demonstrates global keyword:

```
x = 20
def fun():
        global x
        x = 10
        print("Local x =",x)
fun()
print("Global x =",x)
```

Output of the above program is:

```
Local x = 10
Global x = 10
```

In case of nested functions:

- Inner function can access variables n both outer and inner functions.
- Outer function can access variables defined only in the outer function.

Anonymous Functions or LAMBDA Functions

Functions that don't have any name are known as lambda or anonymous functions. Lambda functions are created using the keyword *lambda*. Lambda functions are one line functions that can be created and used anywhere a function is required. Lambda is simply the name of letter 'L' in the Greek alphabet. A lambda function returns a function object.

A lambda function can be created as follows:

lambda arguments-list : expression

The arguments-list contains comma separated list of arguments. Expression is an arithmetic expression that uses the arguments in the arguments-list. A lambda function can be assigned to a variable to give it a name.

Following is an example for creating lambda function and using it:

```
power = lambda x : x*x*x
print(power(3))
```

In the above example the lambda function is assigned to power variable. We can call the lambda function by writing power(3), where 3 is the argument that goes in to formal parameter x. Points to remember when creating lambda functions:

- Lambda functions doesn't have any name.
- Lambda functions can take multiple arguments.
- Lambda functions can returns only one value, the value of expression.
- Lambda function does not have any return statement.
- Lambda functions are one line functions.
- Lambda functions are not equivalent to inline functions in C and C++.
- They cannot access variables other than those in the arguments.
- Lambda functions cannot even access global variables.
- Lambda functions can be passed as arguments in other functions.
- Lambda functions can be used in the body of regular functions.

- Lambda functions can be used without assigning it to a variable.
- We can pass lambda arguments to a function.
- We can create a lambda function without any arguments.
- We can nest lambda functions.
- Time taken by lambda function to execute is almost similar to regular functions.

Recursive Functions

Recursion is another way to repeat code in our programs. A recursive functions is a function which calls itself. A recursive functions should contain two major parts:

- base condition part: A condition which terminates recursion and returns the result.
- recursive part: Code which calls the function itself with reduced data or information.

Recursion uses divide and conquer strategy to solve problems. Following is an example for recursive function which calculates factorial of a number:

```
def fact(n):
        """Returns the factorial of a number
        using recursion."""
        if n==0 or n==1: return 1
        else: return n*fact(n-1)

print(fact(4))
```

In the above example test inside triple quotes is called docstring or documentation string. The docstring of function can be printed by writing *function_name.__doc__*

Following is an example for recursive function which calculates GCD of two numbers:

```
def gcd(x,y):
        rem = x%y
        if rem == 0: return y
        else: return gcd(y, rem)

print(gcd(6,40))
```

Modules

Introduction

A function allows to reuse a piece of code. A module on the other hand contains multiple functions, variables, and other elements which can be reused. A module is a Python file with .py extension. Each .py file can be treated as a module.

We can print the name of existing module by using the *__name__* attribute of the module as follows:

```
print("Name of module is:", __name__)
```

Above code prints the name of module as *__main__*. By default the name of the current module will be *__main__*.

Module Loading and Execution

A module imported in a Python program should available where the program file is located. If the module is not available where the program file is, it looks for the module in the directories available in PYTHONPATH. If the module is still not found, it will search in the *lib* directory of Python installation. If still not found, it will generated error names *ImportError*.

Once a module is located, it is loaded into memory. A compiled version of the module will be created with *.pyc* extension. When the module is referenced next time, the *.pyc* file will be loaded into memory instead of recompiling it. A new compiled version (.pyc) will be generated whenever it is out of date. Programmer can also force the Python shell to recompile the .py file by using the *reload* function.

import Statement

The *import* keyword allows to use functions or variables available in a module. Syntax of import statement is as follows:

import module_name

After importing the module, we can access functions and variables available in it as follows:

module_name.variable_name
module_name.function_name(arguments)

Following program imports the pre-defined *sys* module and prints the PYTHONPATH information by using the *path* variable as follows:

```
import sys
print(sys.path)
```

Following program imports the pre-defined *random* module and prints a random number by calling the *choice* function as follows:

```
import random
print(random.choice(range(1,101)))
```

Above code generates a random number in the range 1 to 100.

from...import Statement

A module contains several variables and functions. When we use *import* statement we are importing everything in the module. To import only selected variables or functions from the module, we use *from...import* statement. To import everything from the module we can write:

*from module-name import **

For example to import the *pi* variable from the math module we can write:

from math import pi

Now, we can directly use pi variable in our program as:

*pi * r * r*

We can import multiple elements from the module as:

from module-name import ele1, ele2, ...

We can also import a module element with another name (alias) using the *as* keyword as follows:

from module-name import element as newname

We can access the command line parameters passed to a Python script using the *argv* variable available in the *sys* module as follows:

```
import sys
print(sys.argv[0], sys.argv[1], ...)
```

We can terminate a Python script abruptly by using the *exit()* method as follows:

```
import sys
sys.exit("Error")
```

Creating Modules

A python file with .py extension is treated as a module. For example, the below script is saved as *circle.py*:

```
from math import pi
def area(r):
        return pi*r*r*r
def peri(r):
        return 2*pi*r
```

Following is the main module file named *test.py* in which we import our own module *circle* as follows:

```
import circle
r = int(input())
print("Area of circle is:", circle.area(r))
```

Notice that we are using *area()* function which was defined in the module *circle*. Remember that *circle.py* and *test.py* files should be at the same location.

Steps for creating and using our own module:

- Define the variables, functions, and other elements that you want and save the file as filename.py.
- The filename will server as the module name.

- Now, create a new file (our main module) in which we can import our module by using import or from...import statement.
- Use the variables or functions in the created module using dot (.) operator.

Namespace

A namespace is a logical collection of names. Namespaces are used to eliminate name collisions. Python does not allow programmers to create multiple identifiers with same name. But, in some case we need identifiers having same name.

For example, consider two modules *module1* and *module* have the same function *foo()* as follows:

```
#Module 1
def foo():
        print("Foo in module 1")
#Module 2
def foo():
        print("Foo in module 2")
```

When we import both modules into same program and try to refer *foo()*, error will be generated:

```
import module1
import module2
foo()
```

To avoid the error, we have to use fully qualified name as follows:

```
module1.foo()
module2.foo()
```

Each module has its own namespace.

Built-in, Global, Local Namespaces

The *built-in namespace* contains all the built-in functions and constants from the *__builtin__* library. The *global namespace* contains all the elements in the current module. The *local namespace* contains all the elements in the current function or block. Runtime will search for the identifier first in the local, then in the global, then finally in the built-in namespace.

Packages

A package is a directory structure which can contain modules and sub packages. Every package in Python is a directory which must contain a special file called __init.py__. The file __init.py__ can be empty. To allow only certain modules to be imported, we can use the __all__ variable in __init.py__ file as follows:

```
__all__ = ["Stat"]
```

The above line says that only *Stat* module in the package can be imported.

Every package must be a folder in the system. Packages should be placed in the path which is pointed by *sys.path*. If the package is not available, then *ImportError* is raised.

Creating a Package

A package (folder) named *CustomMath* is created with an empty __init__.py file inside it. In the package *CustomMath*, a single module named *BasicMath* is created which contains three functions as follows:

```
def prod(x, y):
    return x*y

def rem(x, y):
    return x%y


def lastdigit(x):
    return x%10
```

A new Python script is written which uses our custom package named *CustomMath* as follows:

```
from CustomMath.BasicMath import *
print(prod(4,3))
print(rem(4,3))
print(lastdigit(99))
```

PyPI and Pip

Third party packages are available in Python Package Index (PyPI). We can import the modules available in the third party packages by using the *pip* tool. *Pip* is a command line tool for maintaing Python packages.

To install a package using pip we write:

```
pip install package-name
```

To see the list of existing packages, we write:

```
pip list
```

Standard Library Modules

Python comes with several built-in modules. We can use them when necessary. These built-in modules forms the Python's standard library. Some of the modules in standard library are:

- string
- re
- datetime
- math
- random
- os

- multiprocessing
- subprocess
- email
- json
- doctest
- unittest
- pdb
- argparse
- socket
- sys

globals(), locals(), and reload()

The *globals()* function returns all the names in the global namespace that can be accessed from inside a function.

The *locals()* function returns all the names in the local namespace i.e., within the function. Both the above functions returns the names as a dictionary.

The *reload()* function can be used to reload a module again the program if need as follows:

reload(module_name)