UNIT-III

3.1 DEFINITION

Function is a group of related statements that perform a specific task. Functions are named blocks of code that are designed to do specific job.

When you want to perform a particular task that you have defined in a function, you call the name of the function responsible for it. If you need to perform that task multiple times throughout your program, you don't need to type all the code for the same task again and again; you just call the function dedicated to handling that task, and the call tells Python to run the code inside the function. You'll find that using functions makes your programs easier to write, read, test, and fix errors.

3.2 TYPES OF FUNCTIONS

i) . Predefined Functions (Built-in functions) Functions that is inbuilt with in Python.

ii). User-defined functions Functions defined by the users themselves.

i). Built-in Function

It was are already written or defined in python. We need only to remember the names of built-in functions and the parameters used in the functions. As these functions are already defined so we do not need to define these functions.

Function Name	Description
len()	It returns the length of an object/value.
list()	It returns a list.
max()	It is used to return maximum value from a sequence (list, sets) etc.
min()	It is used to return minimum value from a sequence (list, sets) etc.
open()	It is used to open a file.
print()	It is used to print statement.
str()	It is used to return string object/value.

sum()	It is used to sum the values inside sequence.
type()	It is used to return the type of object.
tuple()	It is used to return a tuple.

Example:

```
L = [4, 5, 1, 2, 9, 7, 10, 8]

count = sum(L)                # using sum() method
avg = count/len(L)            # finding average

print("sum = ", count)
print("average = ", avg)
```

Output:
sum = 46
average = 5.75

ii). User-defined functions

The functions defined by a programmer to reduce the complexity of big problems and to use that function according to their need. This type of functions is called user-defined functions.

Example:

```
x=3
y=4
def add():
print(x+y)
add()
```

Output: 7

3.3 PASSING PARAMETERS IN FUNCTIONS

Information can be passed into functions as arguments. Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

Syntax:

```
def function name (parameter(s) separated by comma):
```

The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

Example:

```
def my_fun(fname):
  print(fname + "Applications")
my_fun("Computer")
```

Output:

Computer Applications

3.4 VARIABLE NUMBER OF ARGUMENTS IN PYTHON

An argument is a variable, value, or object passed to a function or method as input. In Python, there are two ways to define a function that can take a variable number of arguments. Different forms of this type are:

i). Positional arguments

ii). Keyword arguments

i). Positional arguments are arguments that need to be included in the proper position or order. The first positional argument always needs to be listed first when the function is called. The second positional argument needs to be listed second, the third positional argument listed third, etc.

Syntax

function(*iterable)

Example

```
def positional_args(*argv):
    for arg in argv:
        print (arg)
positional_args ('Hello', 'Welcome', 'to', 'GITAM')
```

Output

Hello
Welcome
To
GITAM

ii). Keyword arguments A keyword argument is an argument passed to a function or method that is preceded by a keyword and an equal sign.

Syntax

function(keyword = value)

Where function is the function name, the keyword is the keyword argument, and value is the value or object passed as that keyword. In the code below, **kwargs is used to access the keyword arguments.

Example

```
def key_args(**kwargs):
    for key, value in kwargs.items():
        print ("%s == %s" %(key, value))
key_args (first ='GITAM', mid ='for', last='GITAM')
```

Output

```
Last == GITAM
For == for
First == GITAM
```

3.5. FUNCTION SCOPE

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.

It has two types, **i). Local Scope** **ii). Global Scope**

i). Local Scope

A variable created inside a function belongs to the *local scope* of that function, and can only be used inside that function.

Example

A variable created inside a function.

```
def myfunc():
  x = 300
  print(x)
myfunc()
```

Output:

```
300
```

ii). Global Scope

A variable created in the main body of the Python code is a global variable and belongs to the global scope. Global variables are available from within any scope, global and local.

Example:

A variable created outside of a function is global and can be used by anyone.

```
x = 300
def myfunc():
  print(x)
myfunc()
print(x)
```

Output: 300

3.6. TYPE CONVERSION

The process of converting the value of one data type (integer, string, float, etc.) to another data type is called type conversion. Python has two types of type conversion.

i). Implicit Type Conversion

ii). Explicit Type Conversion

i). Implicit Type Conversion

In Implicit type conversion, Python automatically converts one data type to another data type. This process doesn't need any user involvement.

Let's see an example where Python promotes the conversion of the lower data type (integer) to the higher data type (float) to avoid data loss.

Example 1: Converting integer to float

```
num_int = 123
num_flo = 1.23
num_new = num_int + num_flo
print("Value of num_new:",num_new)
```

Output:

Value of num_new: 124.23

ii). Explicit Type Conversion

In Explicit Type Conversion, users convert the data type of an object to required data type. We use the predefined functions like int(), float(), str(), etc to perform explicit type conversion.

This type of conversion is also called typecasting because the user casts (changes) the data type of the objects.

Syntax:

<required_datatype>(expression)

Typecasting can be done by assigning the required data type function to the expression.

Example:

```
num_int = 123
num_str = "456"
num_str = int(num_str)
num_sum = num_int + num_str
print("Sum of num_int and num_str:",num_sum)
```

Output:

Sum of num_int and num_str: 579

3.7. TYPE COERCION

There are occasions when we would like to convert a variable from one data type to another. This is referred to as type coercion. We can coerce a variable to another date type by passing it to a function whose name is identical to the desired data type.

Example:

If we want to convert a variable x to an integer, we would use the command int(x). If we want to convert a variable y to a float, we would use float(y). We will study coercion more later, but for now, let's see what happens when we coerce int to floats and vice-versa.

EX1: # Coercing an int to a float.

```
x_int = 19
x_float = float(x_int)
print(x_float)
print(type(x_float))
```

Output 1:

19.0

<class 'float'>

EX2: # Coercing a float to an int.

```
y_float = 6.8
y_int = int(y_float)
print(y_int)
print(type(y_int))
```

Output 2:

```
6
<class 'int'>
```

Notice that we coerce a float to an int, the Python does **not round the float** to the nearest integer. Instead, it truncates (or chops off) the decimal portion of the number. In other words, when performing float-to-int coercion, Python will ALWAYS round the number DOWN to the next lowest integer, regardless of the value of the decimal portion.

3.8. MAPPING FUNCTIONS IN A DICTIONARY

In python, a dictionary is a mixed collection of elements. Unlike other collection data types such as a list or tuple, the dictionary type stores a key along with its element. The keys in a Python dictionary is separated by a colon (:) while the commas work as a separator for the elements. The key value pairs are enclosed with curly braces { }.

Syntax (defining a dictionary)

Dictionary_Name = { Key_1: Value_1,Key_2:Value_2,……..Key_n:Value_n }

Key in the dictionary must be unique case sensitive and can be of any valid Python type.

3.8.1 Creating a Dictionary

```
# Empty dictionary
Dict1 = { }
# Dictionary with Key
Dict_Stud = { 'RollNo': '201318', 'Name':'Peterson', 'Class':'BCA', 'Marks':'550'}
```

3.8.2 Dictionary Comprehensions

In Python, comprehension is another way of creating dictionary. The following is the syntax of creating such dictionary.

Syntax

Dict = { expression for variable in sequence [if condition] }

Example

```
Dict = { x : 2 * x for x in range(1,10)}
```

Output

```
{1: 2, 2: 4, 3: 6, 4: 8, 5: 10, 6: 12, 7: 14, 8: 16, 9: 18}
```

3.8.3 USING Map() WITH DICTIONARY

A dictionary in Python is created using curly brackets({}). Since the dictionary is an iterator, you can make use of it inside map() function. Let us now use a dictionary as an iterator inside map() function.

Example

```
def myMapFunc(n):
   return n*10
my_dict = {2,3,4,5,6,7,8,9}
finalitems = map(myMapFunc, my_dict)
print(finalitems)
print(list(finalitems))
```

Output:

```
<map object at 0x0000007EB451DEF0>
[20, 30, 40, 50, 60, 70, 80, 90]
```

3.9 USING Map() WITH LAMBDA FUNCTION

In Python, lambda expressions are utilized to construct anonymous functions. You will have to use the lambda keyword just as you use def to define normal functions.

So in the example, we are going to use the lambda function inside the map(). The lambda function will multiply each value in the list with 10.

Example:

```
my_list = [2,3,4,5,6,7,8,9]
updated_list = map(lambda x: x * 10, my_list)
print(updated_list)
print(list(updated_list))
```

Output:

<map object at 0x000000BD18B11898>

[20, 30, 40, 50, 60, 70, 80, 90]

3.10 MODULE

Consider a module to be the same as a code library.

A file containing a set of functions you want to include in your application.

Create a Module

To create a module just save the code you want in a file with the file extension .py:

Example

Save this code in a file named **mymodule.py**

```
def greeting(name):
  print("Hello, " + name)
```

Use a Module

Now we can use the module we just created, by using the import statement:

Example

Import the module named mymodule, and call the greeting function

```
import mymodule
mymodule.greeting("New College")
```

Output

Hello, New College

3.11 STANDARD MODULES IN PYTHON

3.11.1 PYTHON - SYS MODULE

The sys module provides functions and variables used to manipulate different parts of the Python runtime environment. You will learn some of the important features of this module here.

i. sys.argv

sys.argv returns a list of command line arguments passed to a Python script. The item at index 0 in this list is always the name of the script. The rest of the arguments are stored at the subsequent indices.

Here is a Python script (test.py) consuming two arguments from the command line.

test.py

```
import sys
```

print("You entered: ",sys.argv[1], sys.argv[2], sys.argv[3])

Output

You entered: Python C# Java

ii. sys.exit

This causes the script to exit back to either the Python console or the command prompt. This is generally used to safely exit from the program in case of generation of an exception.

sys.maxsize

Returns the largest integer a variable can take.

Example: sys.maxsize

```
>>> import sys
>>>sys.maxsize
9223372036854775807
```

iii. sys.path

This is an environment variable that is a search path for all Python modules.

Example: sys.path

```
>>> import sys
>>>sys.path
['', 'C:\\python36\\Lib\\idlelib', 'C:\\python36\\python36.zip',
'C:\\python36\\DLLs', 'C:\\python36\\lib', 'C:\\python36',
'C:\\Users\\acer\\AppData\\Roaming\\Python\\Python36\\site-packages',
'C:\\python36\\lib\\site-packages']
```

iv. sys.version

This attribute displays a string containing the version number of the current Python interpreter.

Example: sys.version

```
>>> import sys
>>>sys.version
'3.7.0 (v3.7.0:f59c0932b4, Mar 28 2018, 17:00:18) [MSC v.1900 64 bit (AMD64)]'
```

3.10.2 Python - Math Module

Some of the most popular mathematical functions are defined in the math module. These include trigonometric functions, representation functions, logarithmic functions, angle conversion functions, etc. In addition, two mathematical constants are also defined in this module.

i. **Pi** is a well-known mathematical constant, which is defined as the ratio of the circumference to the diameter of a circle and its value is 3.141592653589793.

Example: Getting Pi Value

```
>>> import math
>>>math.pi
3.141592653589793
```

Another well-known mathematical constant defined in the math module is **e**. It is called **Euler's number** and it is a base of the natural logarithm. Its value is 2.718281828459045.

Example: e Value

```
>>> import math
>>> math.e
2.718281828459045
```

The math module contains functions for calculating various trigonometric ratios for a given angle. The functions (sin, cos, tan, etc.) need the angle in radians as an argument. We, on the other hand, are used to express the angle in degrees. The math module presents two angle conversion functions: degrees() and radians(), to convert the angle from degrees to radians and vice versa. For example, the following statements convert the angle of 30 degrees to radians and back (Note: π radians is equivalent to 180 degrees).

Example: Math Radians and Degrees

```
>>> import math
>>> math.radians(30)
0.5235987755982988
>>> math.degrees(math.pi/6)
29.999999999999996
```

The following statements show sin, cos and tan ratios for the angle of 30 degrees (0.5235987755982988 radians):

Example: sin, cos, tan Calculation

```
>>> import math
>>> math.sin(0.5235987755982988)
0.49999999999999994
>>> math.cos(0.5235987755982988)
0.8660254037844387
>>> math.tan(0.5235987755982988)
0.5773502691896257
```

You may recall that sin(30)=0.5, cos(30)=32 (which is 0.8660254037844387) and tan(30)= 13 (which is 0.5773502691896257).

ii. math.log()

The math.log() method returns the natural logarithm of a given number. The natural logarithm is calculated to the base e.

Example: log

```
>>> import math
>>>math.log(10)
2.302585092994046
```

iii. math.log10()

The math.log10() method returns the base-10 logarithm of the given number. It is called the standard logarithm.

Example: log10

```
>>> import math
>>>math.log10(10)
1.0
```

iv. math.exp()

The math.exp() method returns a float number after raising e to the power of the given number. In other words, exp(x) gives e**x.

Example: Exponent

```
>>> import math
```

>>>math.exp(10)

22026.465794806718

This can be verified by the exponent operator.

Example: Exponent Operator **

>>> import math

>>>math.e**10

22026.465794806703

v. math.pow()

The math.pow() method receives two float arguments, raises the first to the second and returns the result. In other words, pow(4,4) is equivalent to 4**4.

Example: Power

>>> import math

>>> math.pow(2,4)

16.0

>>> 2**4

16

vi. math.sqrt()

The math.sqrt() method returns the square root of a given number.

Example: Square Root

>>> import math

>>> math.sqrt(100)

10.0

>>> math.sqrt(3)

1.7320508075688772

The following two functions are called representation functions. The **ceil()** function approximates the given number to the smallest integer, greater than or equal to the given floating point number. The floor() function returns the largest integer less than or equal to the given number.

Example: Ceil and Floor

>>> import math

>>> math.ceil(4.5867)

5

```
>>> math.floor(4.5687)
4
```

3.10.3 Time Module

We will learn to use different time-related functions defined in the time module with the help of examples. Python has a module named time to handle time-related tasks. To use functions defined in the module, we need to import the module first. Here's how:

```
import time
```

Here are commonly used time-related functions.

i. time.time()

The time() function returns the number of seconds passed since epoch.

For Unix system, January 1, 1970, 00:00:00 at **UTC** is epoch (the point where time begins).

```
import time
seconds = time.time()
print("Seconds since epoch =", seconds)
```

ii. time.ctime()

The time.ctime() function takes seconds passed since epoch as an argument and returns a string representing local time.

```
import time
# seconds passed since epoch
seconds = 1545925769.9618232
local_time = time.ctime(seconds)
print("Local time:", local_time)
```

If you run the program, the output will be something like:

```
Local time: Thu Dec 27 15:49:29 2018
```

iii. time.sleep()

The sleep() function suspends (delays) execution of the current thread for the given number of seconds.

```
import time
print("This is printed immediately.")
time.sleep(2.4)
print("This is printed after 2.4 seconds.")
```

3.10.4 The dir() function

It returns all properties and methods of the specified object, without the values. This function will return all the properties and methods, even built-in properties which are default for all object.

Syntax

dir(*object*)

Parameter Values

Parameter	Description
object	The object you want to see the valid attributes of

dir() Function

Example

Display the content of an object:

```
class Person:
  name = "John"
  age = 36
  country = "Norway"
print(dir(Person))
```

Output

['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'age', 'country', 'name']

3.10.5 Python help() Function

Python help() function is used to get help related to the object passed during the call. It takes an optional parameter and returns help information. If no argument is given, it shows the Python help console. It internally calls python's help function.

Syntax

help(object)

Parameters

object: It can be any object for which we want to get help.

Return

It returns the object's info.

Examples of help() function to understand it's functionality.

 # Python help() function example

```
# Calling function
info = help() # No argument
# Displaying result
print(info)
```

Output:

Welcome to Python 3.5's help utility!

 # Python help() function example

```
# Calling function
info = help(1) # integer value
# Displaying result
print(info)
```

Output:

Help on int object:

```
class int(object)
 |  int(x=0) -> integer
 |  int(x, base=10) -> integer
 |  Methods defined here:
 |  __abs__(self, /)
```

```
|      abs(self)
|  __add__(self, value, /)
|      Return self+value.
|  __and__(self, value, /)
|      Return self&value.
```

~~~~ The End of Unit-III ~~~~