B Sc Computer Science – VI Semester

Elective Papers - CS6PET01: Python and LateX

Module III – Functions
Basic in-build functions, User defined functions, Function Calls, Parameterized function calls, Function returns, Recursive functions, Scope concepts - local, global

Functions

The process of dividing a computer program into separate independent blocks of code or separate sub-problems with different names and specific functionalities is known as modular programming.

In programming, the use of function is one of the means to achieve modularity and reusability. Function can be defined as a named group of instructions that accomplish a specific task when it is invoked. Once defined, a function can be called repeatedly from different places of the program without writing all the codes of that function every time, or it can be called from inside another function, by simply writing the name of the function and passing the required parameters, if any. The programmer can define as many functions as desired while writing the code.

Advantages of Function

The following are the advantages of using functions in a program:

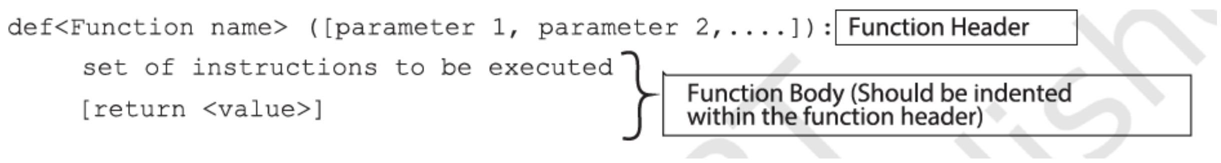
- Increases readability, particularly for longer code as by using functions, the program is better organised and easy to understand.
- Reduces code length as same code is not required to be written at multiple places in a program. This also makes debugging easier.
- Increases reusability, as function can be called from another function or another program. Thus, we can reuse or build upon already defined functions and avoid repetitions of writing the same piece of code.
- Work can be easily divided among team members and completed in parallel.

USER DEFINED FUNCTIONS

Taking advantage of reusability feature of functions, there is a large number of functions already available in Python under standard library. We can directly call these functions in our program without defining them. However, in addition to the standard library functions, we can define our own functions while writing the program. Such functions are called user defined functions. Thus, a function defined to achieve some task as per the programmer's requirement is called a user defined function.

Creating User Defined Function

A function definition begins with def (short for define).
The syntax for creating a user defined function is
as follows:



```
def<Function name> ([parameter 1, parameter 2,....]):
      set of instructions to be executed
      [return <value>]
```

Function Header
Function Body (Should be indented within the function header)

- The items enclosed in "[ ]" are called parameters and they are optional. Hence, a function may or may not have parameters. Also, a function may or may not return a value.
- Function header always ends with a colon (:).
- Function name should be unique. Rules for naming identifiers also applies for function naming.
- The statements outside the function indentation are not considered as part of the function.

Write a user defined function to add 2 numbers and display their sum.

```
#Function to add two numbers
#The requirements are listed below:
#1. We need to accept 2 numbers from the user.
#2. Calculate their sum
#3. Display the sum.

#function definition
def addnum():
fnum = int(input("Enter first number: "))
snum = int(input("Enter second number: "))
sum = fnum + snum
print("The sum of ",fnum,"and ",snum,"is ",sum)
#function call
  addnum()
```

Output:

```
Enter first number: 5
Enter second number: 6
The sum of 5 and 6 is 11
```

In order to execute the function addnum(), we need to call it. The function can be called in the program by writing function name followed by () as shown in the last line of program.

Arguments and Parameters

In the above example, the numbers were accepted from the user within the function itself, but it is also possible for a user defined function to receive values at the time of being called. An argument is a value passed to the function during the function call which is received in corresponding parameter defined in function header.

Write a program using a user defined function that displays sum of first *n* natural numbers, where *n* is passed as an argument.

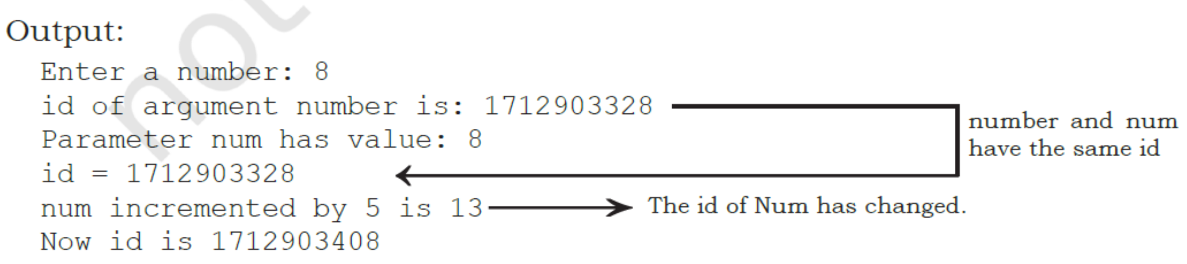
```
#Program to find the sum of first n natural numbers
#The requirements are:
#1. n be passed as an argument
#2. Calculate sum of first n natural numbers
#3. Display the sum

#function header
def sumSquares(n): #n is the parameter
sum = 0
for i in range(1,n+1):
sum = sum + i
print("The sum of first",n,"natural numbers is: ",sum)

num = int(input("Enter the value for n: "))
#num is an argument referring to the value input by the user
sumSquares(num) #function call
```

Note : Since both `num` and `n` are referring to the same value, they are bound to have the same identity. We can use the id() function to find the identity of the object that the argument and parameter are referring to. Let us understand this with the help of the *following example*.

Write a program using user defined function that accepts an integer and increments the value by 5. Also display the `id` of argument (before function call), `id` of parameter before increment and after increment.

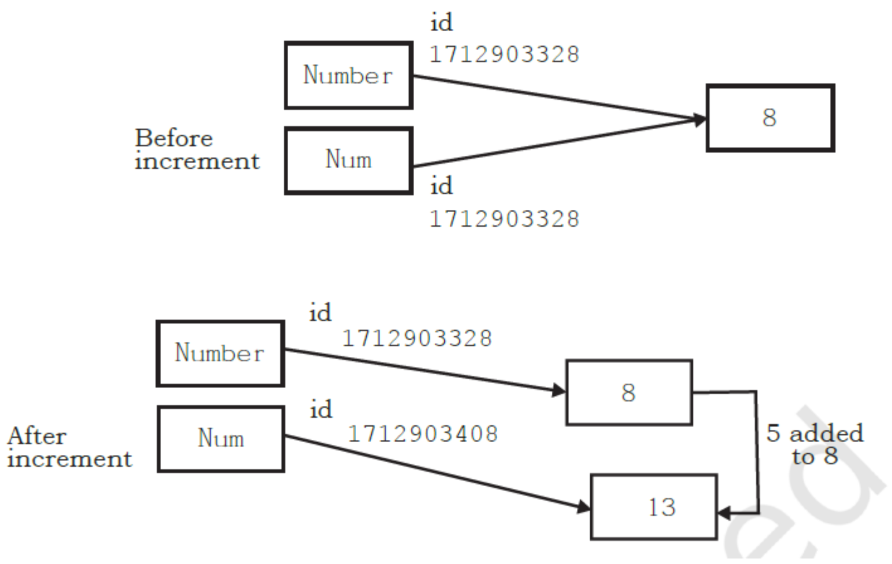
```
#Function to add 5 to a user input number
#The requirements are listed below:
#1. Display the id()of argument before function call.
#2. The function should have one parameter to accept the argument
#3. Display the value and id() of the parameter.
#4. Add 5 to the parameter
#5. Display the new value and id()of the parameter to check
#whether the parameter is assigned a new memory location or
#not.
def incrValue(num):
#id of Num before increment
print("Parameter num has value:",num,"\nid =",id(num))
num = num + 5
#id of Num after increment
```

```
print("num incremented by 5 is",num,"\nNow id is ",id(num))
number = int(input("Enter a number: "))
print("id of argument number is:",id(number)) #id of Number
incrValue(number)
```



Output:

```
Enter a number: 8
id of argument number is: 1712903328
Parameter num has value: 8
id = 1712903328
num incremented by 5 is 13
Now id is 1712903408
```



Write a program using a user defined function calcFact() to calculate and display the factorial of a number num passed as an argument.

```
#Function to calculate factorial
#The requirements are listed below:
#1. The function should accept one integer argument from user.
#2. Calculate factorial. For example:
#3. Display factorial

def calcFact(num):
fact = 1
for i in range(num,0,-1):
fact = fact * i
print("Factorial of",num,"is",fact)
num = int(input("Enter the number: "))
calcFact(num)
```

Output:

```
Enter the number: 5
Factorial of 5 is 120
```

Note: Since multiplication is commutative 5! = 5*4*3*2*1 = 1*2*3*4*5

(A) String as Parameters

Write a program using a user defined function that accepts the first name and lastname as arguments, concatenate them to get full name and displays the output as: Hello full name

For example, if first name is Sachin and lastname is Tendulkar, the output should be:
Hello Sachin Tendulkar

```
#Function to display full name
#The requirements are listed below:
#1. The function should have 2 parameters to accept first name and
#last name.
#2. Concatenate names using + operator with a space between first
#name and last name.
#3. Display full name.

def fullname(first,last):
#+ operator is used to concatenate strings
fullname = first + " " + last
print("Hello",fullname)
#function ends here

first = input("Enter first name: ")
last = input("Enter last name: ")
#function call
fullname(first,last)

Output:
Enter first name: Sachin
Enter last name: Tendulkar
Hello Sachin Tendulkar
```

(B) Default Parameter

Python allows assigning a default value to the parameter. A default value is a value that is redecided and assigned to the parameter when the function call does not have its corresponding argument.

Write a program that accepts numerator and denominator of a fractional number and calls a user defined function mixedFraction() when the fraction formed is not a proper fraction. The default value of denominator is 1. The function displays a mixed fraction only if the fraction formed by the parameters does not evaluate to a whole number.

```
#Function to display mixed fraction for an improper fraction
#The requirements are listed below:
#1. Input numerator and denominator from the user.
#2. Check if the entered numerator and denominator form a proper
#fraction.
#3. If they do not form a proper fraction, then call mixedFraction().
#4. mixedFraction()display a mixed fraction only when the fraction
#does not evaluate to a whole number.

def mixedFraction(num,deno = 1):
    remainder = num % deno
    #check if the fraction does not evaluate to a whole number
  if remainder!= 0:
      quotient = int(num/deno)
      print("The mixed fraction=", quotient,"(",remainder, "/", deno,")")
  else:
      print("The given fraction evaluates to a whole number")
#function ends here

num = int(input("Enter the numerator: "))
deno = int(input("Enter the denominator: "))
print("You entered:",num,"/",deno)
if num > deno: #condition to check whether the fraction is improper
  mixedFraction(num,deno) #function call
else:
  print("It is a proper fraction")

Output:
Enter the numerator: 17
Enter the denominator: 2
You entered: 17 / 2
The mixed fraction = 8 ( 1 / 2 )
```

In the above program, the denominator entered is 2, which is passed to the parameter "`deno`" so the default value of the argument `deno` is overwritten. Let us consider the following function call:

```
mixedFraction(9)
```

Here, `num` will be assigned 9 and `deno` will use the default value 1.

Functions Returning Value

A function may or may not return a value when called. The return statement returns the values from the function. In the examples given so far, the function performs calculations and display result(s).They do not return any value. Such functions are called void functions. But a situation may arise, wherein we need to send value(s) from the function to its calling function. This is done using `return` statement. The `return` statement does the following:

- returns the control to the calling function.
- return value(s) or `None`.

- Function with argument(s) and return value(s)

Flow of Execution

Flow of execution can be defined as the order in which the statements in a program are executed. The Python interpreter starts executing the instructions in a program from the first statement. The statements are executed one by one, in the order of appearance from top to bottom.
When the interpreter encounters a function definition, the statements inside the function are not executed until the function is called. Later, when the interpreter encounters a function call, there is a little deviation in the flow of execution. In that case, instead of going to the next statement, the control jumps to the called function and executes the statement of that function. After that, the control comes back the point of function call so that the remaining statements in the program can be executed. Therefore, when we read a program, we should not simply read from top to bottom. Instead, we should follow the flow of control or execution. It is also important to note that a function must be defined before its call within a program.

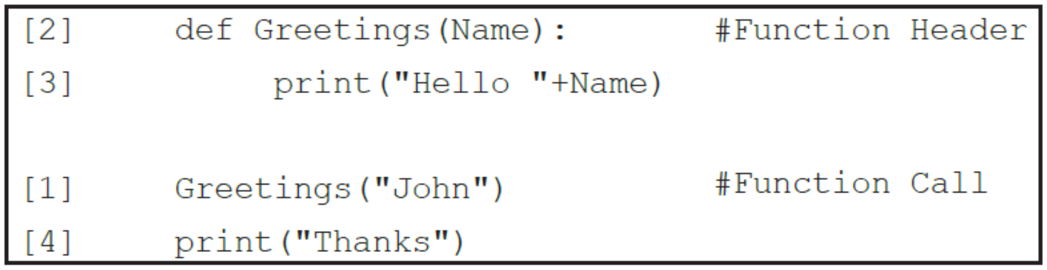
Program to understand the low of execution using functions.

```
#print using functions
helloPython() #Function Call

def helloPython(): #Function definition
    print("I love Programming")

OUTPUT :

NameError: name 'helloPython' is not defined
```

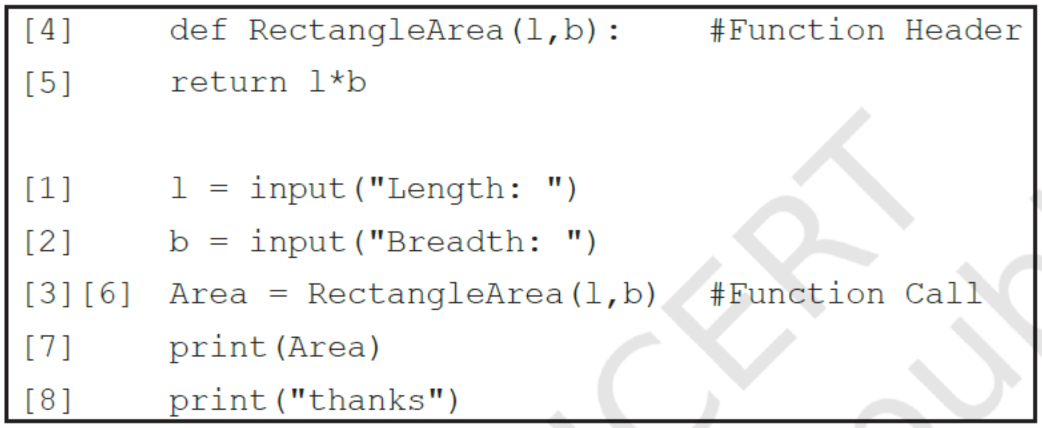
The error 'function not defined' is produced even though the function has been defined. When a function call is encountered, the control has to jump to the function definition and execute it. In the above program, since the function call precedes the function definition, the interpreter does not find the function definition and hence an error is raised.

That is why, the function definition should be made before the function call as shown below:

```
def helloPython(): #Function definition
    print("I love Programming")

helloPython() #Function Call
```



```
[2]     def Greetings(Name):        #Function Header
[3]         print("Hello "+Name)

[1]     Greetings("John")           #Function Call
[4]     print("Thanks")
```



```
[4]      def RectangleArea(l,b):     #Function Header
[5]      return l*b

[1]      l = input("Length: ")
[2]      b = input("Breadth: ")
[3][6]   Area = RectangleArea(l,b)   #Function Call
[7]      print(Area)
[8]      print("thanks")
```

Order of execution of statements (given in square [] brackets)

Write a program using user defined function that accepts length and breadth of a rectangle and returns the area and perimeter of the rectangle.

```
#Function to calculate area and perimeter of a rectangle
#The requirements are listed below:
#1. The function should accept 2 parameters.
#2. Calculate area and perimeter.
#3. Return area and perimeter.
def calcAreaPeri(Length,Breadth):
        area = length * breadth
        perimeter = 2 * (length + breadth)
        #a tuple is returned consisting of 2 values area and perimeter
        return (area,perimeter)

l = float(input("Enter length of the rectangle: "))
b = float(input("Enter breadth of the rectangle: "))

#value of tuples assigned in order they are returned
area,perimeter = calcAreaPeri(l,b)
print("Area is:",area,"\nPerimeter is:",perimeter)
```

Output:

```
Enter Length of the rectangle: 45
Enter Breadth of the rectangle: 66
Area is: 2970.0
Perimeter is: 222.0
```

SCOPE OF A VARIABLE

A variable defined inside a function cannot be accessed outside it. Every variable has a well-defined accessibility. The part of the program where a variable is accessible can be defined as the scope of that variable. A variable can have one of the following two scopes:

A variable that has global scope is known as a global variable and a variable that has a local scope is known as a local variable.

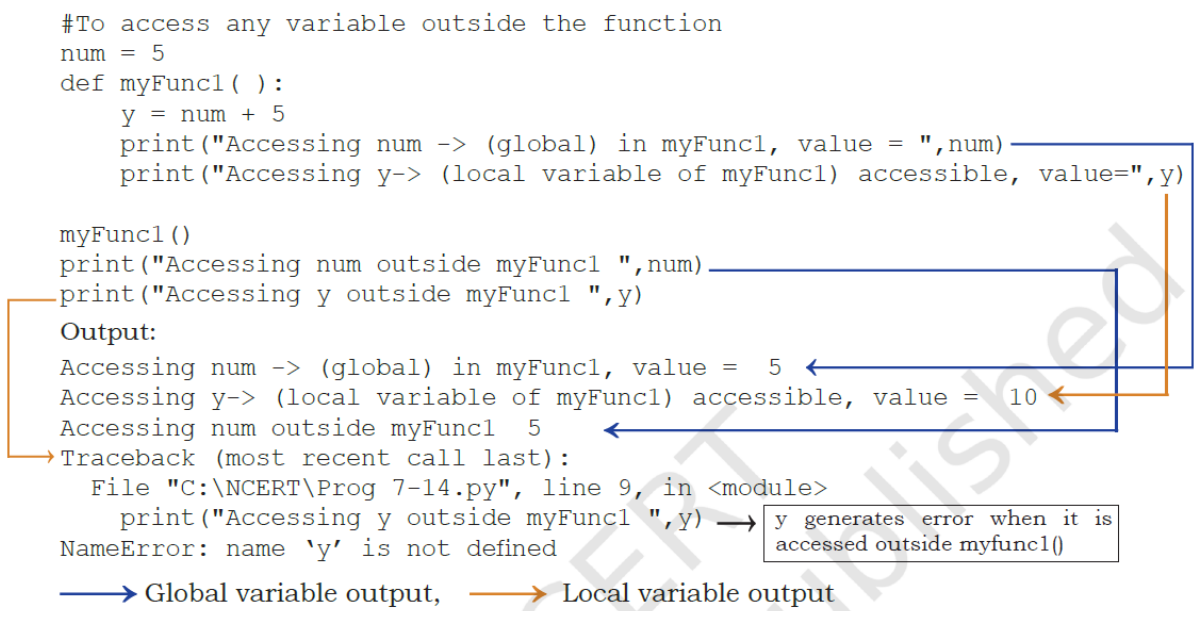
(A) Global Variable

In Python, a variable that is defined outside any function or any block is known as a global variable. It can be accessed in any functions defined onwards. Any change made to the global variable will impact all the functions in the program where that variable can be accessed.

(B) Local Variable

A variable that is defined inside any function or a block is known as a local variable. It can be accessed only in the function or a block where it is defined. It exists only till the function executes.

Program to access any variable outside the function



```
#To access any variable outside the function
num = 5
def myFunc1( ):
    y = num + 5
    print("Accessing num -> (global) in myFunc1, value = ",num)
    print("Accessing y-> (local variable of myFunc1) accessible, value=",y)

myFunc1()
print("Accessing num outside myFunc1 ",num)
print("Accessing y outside myFunc1 ",y)
```

Output:

```
Accessing num -> (global) in myFunc1, value =  5
Accessing y-> (local variable of myFunc1) accessible, value =  10
Accessing num outside myFunc1  5
Traceback (most recent call last):
  File "C:\NCERT\Prog 7-14.py", line 9, in <module>
    print("Accessing y outside myFunc1 ",y)
NameError: name 'y' is not defined
```

***Note*:**

• Any modification to global variable is permanent and affects all the functions where it is used.

• If a variable with the same name as the global variable is defined inside a function, then it is considered local to that function and hides the global variable.

• If the modified value of a global variable is to be used outside the function, then the keyword `global` should be prefixed to the variable name in the function.

Write a program to access any variable outside the function.

```
#To access any variable outside the function
num = 5
def myfunc1():
        #Prefixing global informs Python to use the updated global
        #variable num outside the function
        global num
        print("Accessing num =",num)
        num = 10
        print("num reassigned =",num)
        #function ends here

myfunc1()
print("Accessing num outside myfunc1",num)
```

Output:

```
Accessing num = 5
num reassigned = 10
Accessing num outside myfunc1 10
```

Global variable num is accessed as the ambiguity is resolved by prefixing global to it

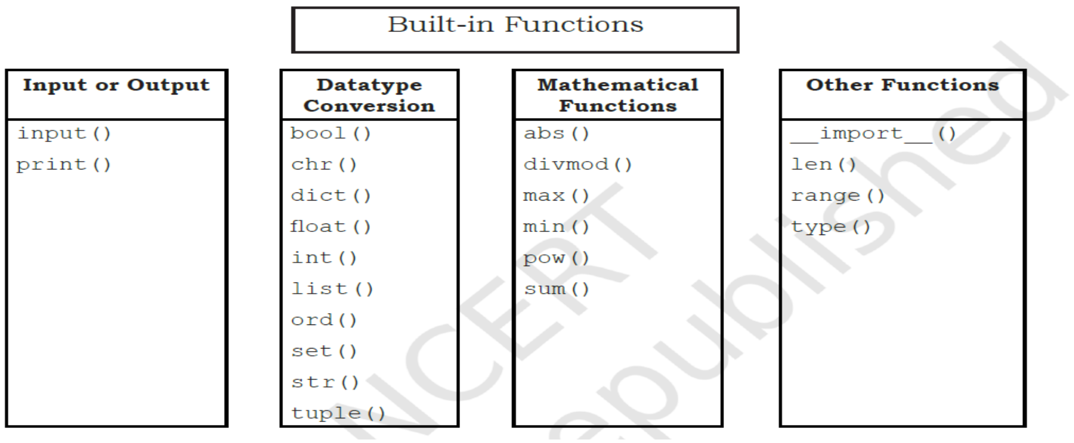
Built-in functions

Built-in functions are the ready-made functions in Python that are frequently used in programs. Consider the following Python program:

```
#Program to calculate square of a number
a = int(input("Enter a number: ")
b = a * a
print(" The square of ",a ,"is", b)
```

In the above program `input()`, `int()` and `print()` are the built-in functions. The set of instructions to be executed for these built-in functions are already defined in the python interpreter.

Some of the built-in functions are :



Built-in Functions

Input or Output	Datatype Conversion	Mathematical Functions	Other Functions
input()	bool()	abs()	__import__()
print()	chr()	divmod()	len()
	dict()	max()	range()
	float()	min()	type()
	int()	pow()	
	list()	sum()	
	ord()		
	set()		
	str()		
	tuple()		

Commonly used built-in functions

Function Syntax	Arguments	Returns	Example Output

abs(x)	x may be an integer or floating point number	Absolute value of x	`>>> abs(4)`
			`4`
			`>>> abs(-5.7)`
			`5.7`

divmod(x,y)	x and y are integers	A tuple: (quotient, remainder)	`>>> divmod(7,2)`
			`(3, 1)`
			`>>> divmod(7.5,2)`
			`(3.0, 1.5)`
			`>>> divmod(-7,2)`
			`(-4, 1)`

max(sequence) or max(x,y,z,...)	x,y,z,.. may be integer or floating point number	Largest number in the sequence/ largest of two or more arguments	`>>> max([1,2,3,4])`
			`4`
			`>>> max("Sincerity")`
			`'y' #Based on ASCII value`
			`>>> max(23,4,56)`
			`56`

min(sequence) or min(x,y,z,...)	x, y, z,.. may be integer or floating point number	Smallest number in the sequence/ smallest of two or more arguments	`>>> min([1,2,3,4])`
			`1`
			`>>> min("Sincerity")`
			`'S'`
			`#Uppercase letters have lower ASCII values than lowercase letters.`
			`>>> min(23,4,56)`
			`4`

pow(x,y[,z])	x, y, z may be integer or floating point number	x^y (x raised to the power y)
		if z is provided, then: (x^y) % z
			`>>> pow(5,2)`
			`25.0`
			`>>> pow(5.3,2.2)`
			`39.2`
			`>>> pow(5,2,4)`
			`1`

sum(x[,num])	x is a numeric sequence and num is an optional argument	Sum of all the elements in the sequence from left to right.
		if given parameter, num is added to the sum
			`>>> sum([2,4,7,3])`
			`16`
			`>>> sum([2,4,7,3],3)`
			`19`
			`>>> sum((52,8,4,2))`
			`66`

len(x)	x can be a sequence or a dictionary	Count of elements in x	`>>> len("Patience")`
			`8`
			`>>> len([12,34,98])`
			`3`
			`>>> len((9,45))`
			`2 >>>len({1:"Anuj",2:"Razia",3:"Gurpreet",4:"Sandra"})`
			`4`

Lambda Function in Python

There are three types of Python Functions. One of them is an anonymous function. Anonymous functions are the functions without a name. Now, to define a normal function, we use the keyword, 'def'. Similarly, to define an anonymous function, we use the keyword, 'lambda'. Since anonymous functions are defined using the lambda keyword, they are also sometimes referred to as lambda functions.

What is a Lambda Function in Python?
The lambda keyword is used to define anonymous functions, that is, functions without names. Python lambda functions are not much different from the regular functions that are defined using the def keyword. Syntax of a lambda function in Python:

Lambda arguments : expression

Here is an example of Lambda in Python.

(lambda a,b : a+ b) (4,6)

If we execute the above code line, the output will be 10.

In the above example, we simply performed addition operation using the lambda function in Python. If we compare this example to the syntax of the Python lambda function, a and b are the arguments and a+b is the expression that is being evaluated and returned, and the whole statement is the lambda function.

We have passed the values in arguments as soon as we defined the lambda function (values being 4 and 6, respectively). The same operation can be performed using a regular function as shown below:

```
def add(a,b):
        return a+b
add(4,6)

Output:

10
```

Module

Other than the built-in functions, the Python standard library also consists of a number of modules. While a function is a grouping of instructions, a module is a grouping of functions. As we know that when a program grows, function is used to simplify the code and to avoid repetition. For a complex problem, it may not be feasible to manage the code in one single file. Then, the program is divided into different parts under different levels, called modules. Also, suppose we have created some functions in a program and we want to reuse them in another program. In that case, we can save those functions under a module and reuse them. A module is created as a python (.py) file containing a collection of function definitions.

To use a module, we need to import the module. Once we import a module, we can directly use all the functions of that module. The syntax of import statement is as follows:

```
import modulename1 [,modulename2, …]
```

This gives us access to all the functions in the module(s). To call a function of a module, the function name should be preceded with the name of the module with a dot(.) as a separator. The syntax is as shown below:

```
modulename.functionname()
```

(A) Built-in Modules

Python library has many built-in modules that are really handy to programmers. Let us explore some commonly used modules and the frequently used functions that are found in those modules:

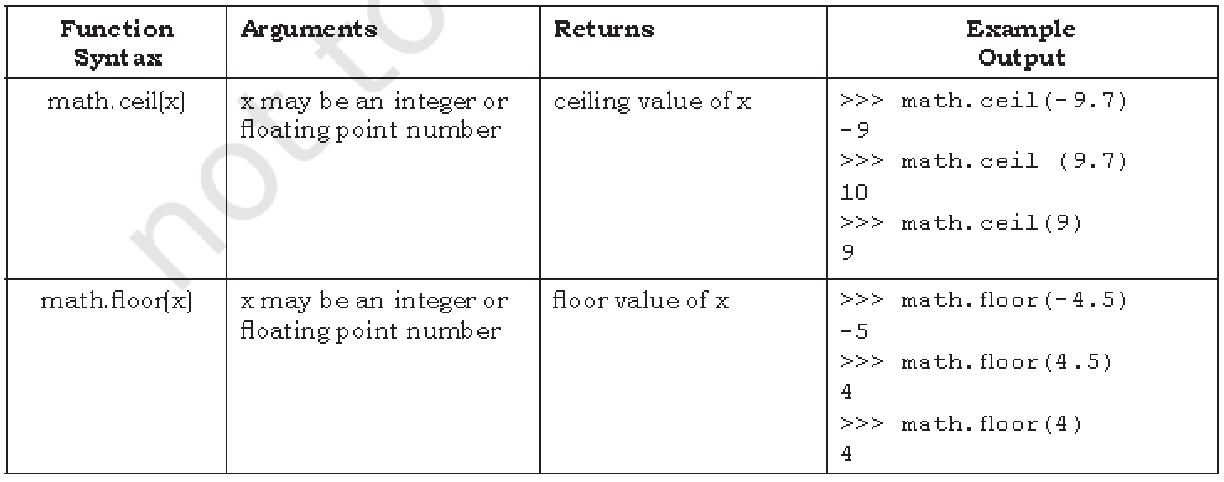
- math
- random
- statistics

1. Module name : math

It contains different types of mathematical functions. Most of the functions in this module return a float value. Some of the commonly used functions in math module are given in Table 7.2. In order to use the math module we need to import it using the following statement:

```
import math
```

Commonly used functions in math **module**



Function Syntax	Arguments	Returns	Example Output

math.ceil(x)	x may be an integer or floating point number	ceiling value of x	>>> math.ceil(-9.7)
			-9
			>>> math.ceil (9.7)
			10
			>>> math.ceil(9)
			9

math.floor(x)	x may be an integer or floating point number	floor value of x	>>> math.floor(-4.5)
			-5
			>>> math.floor(4.5)
			4
			>>> math.floor(4)
			4

math.fabs(x)	x may be an integer or floating point number	absolute value of x	`>>> math.fabs(6.7)`
			`6.7`
			`>>> math.fabs(-6.7)`
			`6.7`
			`>>> math.fabs(-4)`
			`4.0`

math.factorial(x)	x is a positive integer	factorial of x	`>>> math.factorial(5)`
			`120`

math.fmod(x,y)	x and y may be an integer or floating point number	x % y with sign of x	`>>> math.fmod(4,4.9)`
			`4.0`
			`>>> math.fmod(4.9,4.9)`
			`0.0`
			`>>> math.fmod(-4.9,2.5)`
			`-2.4`
			`>>> math.fmod(4.9,-4.9)`
			`0.0`

math.gcd(x,y)	x, y are positive integers	gcd (greatest common divisor) of x and y	`>>> math.gcd(10,2)`
			`2`

math.pow(x,y)	x, y may be an integer or floating point number	x^y (x raised to the power y)	`>>> math.pow(3,2)`
			`9.0`
			`>>> math.pow(4,2.5)`
			`32.0`
			`>>> math.pow(6.5,2)`
			`42.25`
			`>>> math.pow(5.5,3.2)`
			`233.97`

math.sqrt(x)	x may be a positive integer or floating point number	square root of x	`>>> math.sqrt(144)`
			`12.0`
			`>>> math.sqrt(.64)`
			`0.8`

math.sin(x)	x may be an integer or floating point number in radians	sine of x in radians	`>>> math.sin(0)`
			`0`
			`>>> math.sin(6)`
			`-0.279`

2. Module name : random

This module contains functions that are used for generating random numbers. Some of the commonly used functions in random module are given in Table. For using this module, we can import it using the following statement:

```
import random
```

Commonly used functions in random module

Function Syntax	Argument	Return	Example Output

random.random()	No argument (void)	Random Real Number (float) in the range 0.0 to 1.0	`>>> random.random()`
			`0.65333522`

random. randint(x,y)	x, y are integers such that x <= y	Random integer between x and y	`>>> random.randint(3,7)` 4 `>>> random.randint(-3,5)` 1 `>>> random.randint(-5,-3)` -5.0
random. randrange(y)	y is a positive integer signifying the stop value	Random integer between 0 and y	`>>> random.randrange(5)` 4
random. randrange(x,y)	x and y are positive integers signifying the start and stop value	Random integer between x and y	`>>> random.randrange(2,7)` 2

3. Module name : statistics

This module provides functions for calculating statistics of numeric (Real-valued) data. Some of the commonly used functions in statistics module are given in Table. It can be included in the program by using the following statements:

```
import statistics
```

Some of the function available through statistics module

Function Syntax	Argument	Return	Example Output
statistics.mean(x)	x is a numeric sequence	arithmetic mean	`>>> statistics. mean([11,24,32,45,51])` 32.6
statistics.median(x)	x is a numeric sequence	median (middle value) of x	`>>>statistics. median([11,24,32,45,51])` 32
statistics.mode(x)	x is a sequence	mode (the most repeated value)	`>>> statistics. mode([11,24,11,45,11])` 11 `>>> statistics. mode(("red","blue","red"))` 'red'

Note:

• import statement can be written anywhere in the program
• Module must be imported only once
• In order to get a list of modules available in Python, we can use the following statement:

```
>>> help("module")
```

• To view the content of a module say math, type the following:

```
>>> help("math")
```

• The modules in the standard library can be found in the Lib folder of Python.

(B) From Statement

Instead of loading all the functions into memory by importing a module, from statement can be used to access only the required functions from a module. It loads only the specified function(s) instead of all the functions in a module.
Its syntax is
>>> from modulename import functionname [,functionname,...]

To use the function when imported using "from statement" we do not need to precede it with the module name. Rather we can directly call the function as shown in the following examples:

Example

```
>>> from random import random
>>> random() #Function called without the module name
```

Output:

```
0.9796352504608387
```

Example

```
>>> from math import ceil,sqrt
>>> value = ceil(624.7)
>>> sqrt(value)
```

Output:

```
25.0
```

In example the ceil value of 624.7 is stored in the variable "value" and then `sqrt` function is applied on the variable "value". The above example can be rewritten as:

```
>>> sqrt(ceil(624.7))
```

The execution of the function `sqrt()` is dependent on the output of `ceil()` function.
If we want to extract the integer part of 624.7 (we will use `trunc()` function from math module), we can use the following statements.

```
#ceil and sqrt already been imported above
>>> from math import trunc
>>> sqrt(trunc(625.7))
```

Output:

```
25.0
```

A programming statement wherein the functions or expressions are dependent on each other's execution for achieving an output is termed as composition, here are some other examples of composition:

- a = int(input("First number: "))
- `print("Square root of ",a ," = ",math.sqrt(a))`
- print(floor(a+(b/c)))
- math.sin(float(h)/float(c))

Besides the available modules in Python standard library, we can also create our own module consisting of our own functions.