UNIT 11 FUNCTIONS AND FILE HANDLING IN PYTHON

Structure

11.0 INTRODUCTION

Python provides a way of organizing the tasks into more manageable units called functions. Functions make the code modular which is one of the characteristics of an object oriented programming language (OOPs). Modularity is the process of decomposing a problem into set of sub-problems so as to reduce the complexity of a problem. It also makes the code reusable. File handling is another important aspect discussed in this unit. File handling includes- creation, deletion and manipulations of files using python programming.

11.1 OBJECTIVES

After completing this unit, you will be able to

- Perform functions definition and calling
- Understand scope of functions
- Define function objects
- Create lambda fnctions
- Understand basic file operations

11.2 FUCNTION DEFINITION AND CALLING

A function is block of code that performs a specific task. When the size of a program increases, its complexity also increases. Hence, it becomes important to organize the program into more manageable units. Further, it makes the code reusable.

There are three types of functions in python-

1. Built-in functions – these are the functions which are already defined in the language. Exampleprint( ), max( ), input( ), etc.
2. User Defined functions- these are the functions that can be created or defined by the users according to their needs.
3. Anonymous functions

11.2.1 Creating user defined functions

A function can be defined using *def* statement and giving suitable name to a function by following the rules of identifiers. The process of defining a function is called *function definition*.

Syntax of function definition

```
Deffunction_name( list_of_parameters ) :
Statement (s)
```

Where,

function_name() - is the name of the function. A function name must be followed by a set of parenthesis. Any name can be given to a function following the rules of identifiers.

List_of _parameters –is an optional field which is used to pass values or inputs to a function. It can be none or a comma separated list of variables.

Statements(s) – is a set of statements or commands within the function. Each time the function is called, all the statements will be executed. These statements together form the body of a function.

Even if the function is defined, it can never be executed till the time it is called. Hence, for using a function, it must be called using *function call statement*. A function can be called by its name with set of parenthesis and optional list of arguments.

Syntax of Function Calling

```
function_name(list_of_arguments)
```

Example1. A function to display HELLO WORLD

```
def first_function():          # function definition
    print("HELLO WORLD...!!!")

first_function()               # function calling
```

```
============== RESTART: C:/Users/test/Desktop/python_programs/abc.py ==============
HELLO WORLD...!!!
>>>
```

Function needs to be defined only once, but it can be called any number of times by using calling statements. They can not only be called in the same program, but they can also be called in different programs. This will further be discussed in the next chapter.

Another example shown below is the program having a function to calculate factorial of a number.

Example2. A function to display factorial of a number.

```
def factorial():          # function definition
    fact=1
    n=int(input("Enter a number :"))
    for i in range(1,n+1):
         fact=fact*i
    print("Factorial of ",n," is ",fact)

factorial()               # function calling
```

```
========== RESTART: C:/Users/test/Desktop/python_programs/factorial.py ==========
Enter a number :4
Factorial of  4  is  24
>>>
```

11.3 FUNCTION SCOPE

Part of a code in which a variable can be accessed is called *Scope of a variable*. Scope of a variable can be global or local.

Local variables are the variable created within a function's body or function's scope. They cannot be accessed outside the function. Their scope is only limited to the function in which they are created.

Global variables are the variable created outside any function. Their scope is global, hence can be used anywhere. Global variables can also be created within function using keyword *global*.

Example 3. Use of local and global variables.

```
x=1                                    # global variable
def test():
    y=2                                # local variable
    print("Global inside function x=",x) # both local and global can be used here
    print("Local inside function y=",y)

test()
print("Global outside function x=",x)
print("Local outside function y=",y) # shows error:local variable cannot be used her
```

```
============== RESTART: C:/Users/test/Desktop/python_programs/test.py ==============
Global inside function x= 1
Local inside function y= 2
Global outside function x= 1
Traceback (most recent call last):
  File "C:/Users/test/Desktop/python_programs/test.py", line 9, in <module>
    print("Local outside function y=",y)        # shows error: local variable canno
t be used here
NameError: name 'y' is not defined
>>>
```

In Example 3, variable created x is global, hence, can be used both inside and outside any function. Variable y is created inside function test(), therefore, its scope is only limited to this function and hence cannot be used outside.Global variables can be created in functions using *global* keyword as shown in example 4.

Example 4: Program to create global variable inside function.

```
x=1                                    # global variable
def test():
    global x                           # to access global variable for modification
    print("inside function before modification x=",x)
    x=x+2
    print("inside function after modification x=",x)

test()
print("outside function x=",x)
```

```
=========== RESTART: C:/Users/test/Desktop/python_programs/test.py ===========
inside function before modification x= 1
inside function after modification x= 3
outside function x= 3
```

If we create a variable inside function having same name as that of global variable, then a separate variable of same name gets created. Function in this case can access local variable and hence, cannot refer to global variable as shown in example 5.

Example 5: Program showing use of Local and global variables of same name.

```
x=1                                    # global variable
def test():
    x=2                                # local variable
    print("inside function x=",x)

test()
print("outside function x=",x)
```

```
============== RESTART: C:/Users/test/Desktop/python_programs/test.py ==============
inside function x= 2
outside function x= 1
>>>
```

In python, global variable can be accessed in function directly (as shown in Example 3) but they cannot be directly modified inside the function (shown in example 6). Then how can we modify global variables inside function? Answer to this question is through*global* keyword. It can not only be used for creating global variables inside functions, but it can also be used for modifying global variables inside functions. See example 7.

Example 6: Program to show modification of global variable is not allowed inside function

```
x=1              # global variable
def test():
    x=x+2        # shows error: modification of global variable is not allowed
    print("inside function x=",x)

test()
print("outside function x=",x)
```

```
============= RESTART: C:/Users/test/Desktop/python_programs/test.py =============
Traceback (most recent call last):
  File "C:/Users/test/Desktop/python_programs/test.py", line 6, in <module>
    test()
  File "C:/Users/test/Desktop/python_programs/test.py", line 3, in test
    x=x+2        # shows error: modification of global variable is not allowed
UnboundLocalError: local variable 'x' referenced before assignment
>>>
```

Example 7: Program to show modification of global variable using global keyword

```
x=1                                  # global variable
def test():
    global x                         # to access global variable for modification
    print("inside function before modification x=",x)
    x=x+2
    print("inside function after modification x=",x)

test()
print("outside function x=",x)
```

```
========== RESTART: C:/Users/test/Desktop/python_programs/test.py ==========
inside function before modification x= 1
inside function after modification x= 3
outside function x= 3
```

Check your Progress 1

Ex1. What are the benefits of using functions? Also differentiate between function definition and function calling.

Ex2. What is Scope of a variable? Explain local and global variables with example.

Ex3. Write a function named *pattern*to display the pattern given below-

1
1 2
1 2 3
1 2 3 4
1 2 3 4 5

11.4 FUNCTION ARGUMENTS

Arguments are a way of passing values or input to a function. Arguments are passed to a function during function calls. Their values replace the functions parameters in the function definition.

Python provides various mechanisms of passing arguments to a function.

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

1. Required Arguments or Positional Arguments

In required arguments, number of arguments should match number of function parameters and should be given in same order. If number of arguments is not the same, then error will be shown and if correct order is not followed then output can be incorrect.

Example 8: Passing arguments to a function.

```
def calculator(a,b):
    print("Addition      :",a+b)
    print("Subtraction   :",a-b)
    print("Multiplication:",a*b)
    print("Division      :",a/b)

calculator(5,2)
```

```
========== RESTART: C:/Users/test/Desktop/python_programs/arguments.py ==========
Addition      : 7
Subtraction   : 3
Multiplication: 10
Division      : 2.5
>>>
```

2. Default arguments

Default values of variables can be given in the parameters itself. So if any argument is not given at the calling time, it will be replaced by the default value. Default values can be given for any number of arguments. It must be taken care that default argument must follow non-default arguments.

Example 9: Use of Default Arguments

```
def simple_interest(p,r=2,t=5):
    interest=(p*r*t)/100
    print(" Simple Interest is:",interest)

simple_interest(1000)      # p=1000, r=2 (default) ,t=5(default)
simple_interest(1000,5)    # p=1000, r=5 ,t=5(default)
simple_interest(1000,5,10) # p=1000, r=2 ,t=5
```

```
========== RESTART: C:\Users\test\Desktop\python_programs\arguments.py =========
 Simple Interest is: 100.0
 Simple Interest is: 250.0
 Simple Interest is: 500.0
>>>
```

3. Keyword Arguments

We have seen above that arguments must be given in the order in which parameters are defined otherwise the result will be effected. Keyword arguments are one way by which we can give arguments in any order. They are given by specifying the parameter name with each argument during function call. In this case position does not matter but name matters hence they are also called named arguments.

Example 10: Use of Keyword Arguments

```
def compound_interest(p,r,t):
    amount = p*pow(1+ (r/100),t)
    interest = amount-p
    print("Compound Interest is:",interest)

compound_interest(r=13,p=100000,t=5)     # passing keyword arguments
compound_interest(100000,13,5)           # passing positional arguments
```

```
====== RESTART: C:/Users/test/Desktop/python_programs/compound_interest.py =====
Compound Interest is: 84243.51792999991
Compound Interest is: 84243.51792999991
>>>
```

4. Arbitrary Arguments

Sometimes it is required to pass different number of arguments to a same function or it is not known at function definition time that how many arguments could be used at calling time. Python provides the feature of arbitrary arguments to deal with this issue. Arbitrary arguments can be declared using * symbol.

Example11: Use of Arbitrary Arguments

```
def sum(*a):          # declaration of arbitrary argument
    sum=0
    for i in a:
        sum=sum+i
    print("Sum of Series",a," is:",sum)

sum(1,2,5,7,8,10,2)    # passing arbitrary arguments
sum(10,20,25)
```

```
========== RESTART: C:/Users/test/Desktop/python_programs/Arbitrary.py =========
Sum of Series (1, 2, 5, 7, 8, 10, 2)  is: 35
Sum of Series (10, 20, 25)  is: 55
>>>
```

11.5 RETURNING FROM A FUNCTION

As we can pass values as input to function parameters, similarly we can also return or extract values out of a function. This can be done using *return* statement. By default when we do not include return statement, value None is returned from the function.

Syntax of returning from function

```
def function_name( arg1,arg2.... ) :
        .....
        .....
return value
```

Example12: Function with return statement

```
def example(a,b,c):      # function with return statement
    avg=(a+b+c)/3
    return avg

result=example(2,3,4)        # returned value passed to result variable

print("AVG OF NUMBERS:",result)  # display return value by print statement
```

```
======= RESTART: C:/Users/test/Desktop/python_programs/return_example.py =======
AVG OF NUMBERS: 3.0
>>>
```

There can be only one return statement in a function. It doesn't mean that we can return only one value. It means that the return statement can return only one object and object in python can contain single (variable) or multiple values (List, tuple). Hence, we can return multiple values with a single return statement.See example13.

Example 13: Function returning more than one value

```
def example(a,b,c):
    sum=a+b+c
    avg=sum/3
    return sum,avg      # function returning two values with single statement

result1,result2=example(2,3,4)
print("SUM OF NUMBERS:",result1)
print("AVG OF NUMBERS:",result2)
```

```
======= RESTART: C:/Users/test/Desktop/python_programs/return_example.py =======
SUM OF NUMBERS: 9
AVG OF NUMBERS: 3.0
>>>
```

Various benefits of using return statement -

1. Valuesreturned by function can be used again in the rest of the program.
2. They can also be passed as argument to other functions providing better flow of control. See example 14.
3. They can also be used to break out of function in the middle.

Example 14: Function returning more than one value

```
def first(a,b):
    sum=a+b
    return sum

def second(a,b):
    avg=a/b
    return avg

value1=float(input("Enter first input:"))
value2=float(input("Enter second input:"))
result1=first(value1,value2) # result1 contains value return from first()functio
result2=second(result1,2)    # result1 is passed as argument to second()function
print("AVG OF NUMBERS:",result2)  # display return value by print statement

print("10 / 2 =:",second(10,2)) # function with return statement can also be
                                # directly called with print statement
```

```
======= RESTART: C:/Users/test/Desktop/python_programs/return_example.py =======
Enter first input:3
Enter second input:6
AVG OF NUMBERS: 4.5
10 / 2 =: 5.0
```

It is also possible to break out of a function in the middle of statements in function using return. The statements following the return statement will never be executed. In example 15, inside a function, there is loop ranging from 1 to 9, but it will only be executed 4 times, since the function will return when value of i reaches 5, hence remaining iterations will be skipped.

Example 15: Function returning in between the loop

```
def example():
    for i in range(1,10):
        if(i==5):
            return     # function returns when i=5, no further execution
        else:
            print(i)

example()
```

```
======= RESTART: C:/Users/test/Desktop/python_programs/return_example.py =======
1
2
3
4
>>>
```

Check your Progress 2

Ex 1. What are the various ways of passing arguments to a function?

Ex 2. Write a function which takes list of numbers as argument and returns a list of unique elements from it.
Ex 3. Write a function to display all prime numbers between a range which is passed as arguments.

11.6 FUNCTION OBJECTS

In python, data is represented as objects. Like Lists, Strings etc, functions are also treated as objects. Functions in python are first class objects. Since functions are objects,it provides various features with additional to the existing ones. These are-

1. functions can be assigned to a variable
2. functions can be used as elements of data structures
3. functions can be sent as arguments to another function
4. functions can be nested

1. Functions assigned to variable

As functions are object, they can be assigned to a variable (object). Hence, function can then be called using variable and set of parenthesis. Assigning function to a variable creates a reference to the same function or a function with two names. If one function or variable is deleted, function can still be referenced by another name. Following is the example of function assigned to a variable. (example 16)

Example 16: Function assigned to variable

```
def example():
    print("Testing function")


a = example     # function assigned to variable
a()             # function call using variable
```

```
======= RESTART: C:/Users/test/Desktop/python_programs/return_example.py =======
Testing function
>>>
```

In the following console window it can be clearly seen that after deleting a function, it cannot be accessed with the same name. But it can still be called using another name as shown in example 17.

Example 17: Deleting reference to a function

```
>>> del example
>>> example()
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    example()
NameError: name 'example' is not defined
>>> a()
Testing function
>>>
```

2. Functions as element of data structure

Functions can be passed as an element to any data structure. This is very useful at times when we want to apply different functions to same input. Below is the example 18 to show that. We are using various built-in functions to demonstrate that.

Example 18: Function assigned as elements to List

```
abc=[str.upper,str.lower,len]   # list of functions

for i in abc:
    print(i,i("function as element"))# applying different function to same input
```

```
======= RESTART: C:/Users/test/Desktop/python_programs/return_example.py =======
<method 'upper' of 'str' objects> FUNCTION AS ELEMENT
<method 'lower' of 'str' objects> function as element
<built-in function len> 19
>>>
```

3. Function as argument to another function

Objects can be passed as arguments to a function. Since functions are objects in python, they can also be passed as arguments to functions. This technique allows application of multiple operations on a particular set of inputs. Given below is an example of function calling another function as arguments.

Example 19: Function calling another function as argument

```
def add(n):
    sum=0
    for i in n:
        sum=sum+i
    return sum

def display(l,func):
    print("Addition of elements:",func(l))


display([1,2,3,4,5,6,7],add)    # function display() calling function add()
```

```
======= RESTART: C:/Users/test/Desktop/python_programs/return_example.py =======
Addition of elements: 28
>>>
```

4. Nested function

Functions created within another function are called nested function. This is one of the unique properties in python since functions are objects. The inner function which is created within some outer function has access to all the variables of enclosing scope. Inner functions can never be directly called outside outer function. Hence, it provides security feature called Encapsulation. It can only be called by the use of enclosing function.

Example 20: Nested function

```
def outer(text):            # outer function
    print(str.upper(text))

    def inner():            # nested or inner function
        print(str.lower(text))

    inner()     # nested function can only be called inside outer function

outer("Nested Function") # call to outer function makes indirect call to inner
inner("Nested Function") # shows error: nested function cannot be called outside
```

```
======= RESTART: C:/Users/test/Desktop/python_programs/return_example.py =======
NESTED FUNCTION
nested function
Traceback (most recent call last):
  File "C:/Users/test/Desktop/python_programs/return_example.py", line 12, in <m
odule>
    inner("Nested Function") # shows error: nested function cannot be called out
side
NameError: name 'inner' is not defined
>>>
```

11.7 LAMBDA / ANONYMOUS FUNCTIONS

Python allows to create functions without names called Anonymous functions. They are not declared with def keyword, instead *lambda* keyword is used to create these functions. Hence, these functions are also called **lambda functions.**Like other

functions, these functions can also contain arguments but there can be only one statement in the body of these functions. Statement is evaluated and the result is returned. There are situations when a function needs to be created for a short operation (similar to macros in C language) or usable for a short period of time, in that case lambda functions are best suited.

Syntax of Anonymous function

```
Variable = lambda arg : statement
```

Here,arg is one or more arguments passed to a function and statement consists of operation performed by a function.

Given below is example 21 in which two lambda functions are created. First function calculates cube of a given argument and second function gives addition to two given arguments. Lambda function is then assigned to a variable (function assigned to variable, discussed in previous section), which can then be used to call function with arguments and set of parenthesis.

Example 21: Use of Lambda function

```
cube = lambda x : x*x*x    # lambda function to calculate cube
add =  lambda x,y: x+y     # lambda function to calculate sum

print("cube of 3:",cube(3))  # first lambda function called
print("addition of 10 & 20:",add(10,20 # second lambda function called
```

```
========== RESTART: C:/Users/test/Desktop/python_programs/lambda.py ==========
cube of 3: 27
addition of 10 & 20: 30
>>>
```

Lambda functions are mostly used as argument to other high-order function like map () and filter () function.

map() function : map () is a built-in function which takes two arguments, first argument given is function (can be lambda function), second argument is given as a list. map () function returns a list of elements obtained by applyinggiven function to each element of alist.

filter() function : filter() is another built-in function which takes two arguments. First argument is a function and second is list of elements, just as map() function. Here, given function is applied to each of the elements of list and returns a list of items for which the function evaluates to True.

Syntax of map and filter function

```
map ( function ,  [ list ] )

filter ( function , [ list ] )
```

Example 22: Application of map() function with lambda function

```
def factorial(n):
    fact = 1
    for i in range(1,n+1):
        fact = fact*i
    return fact

a=[1,2,3,4,5]

b = list(map(lambda x : x*x*x,a)) # map() applied on lambda function
c = list(map(factorial,a))        # map() applied on user defined function

print("CUBE OF SERIES",b)
print("FACTORIAL OF SERIES",c)
```

```
========== RESTART: C:/Users/test/Desktop/python_programs/lambda.py ==========
CUBE OF SERIES [1, 8, 27, 64, 125]
FACTORIAL OF SERIES [1, 2, 6, 24, 120]
>>>
```

Example 22: Application of filter() function with lambda function

```
a=[1,2,3,4,5,6,7,8,9,10]

b = list(filter(lambda x : x%2==0,a)) # filter() applied on lambda function

print("EVEN NUMBER SERIES",b)
```

```
========== RESTART: C:/Users/test/Desktop/python_programs/lambda.py ==========
EVEN NUMBER SERIES [2, 4, 6, 8, 10]
>>>
```

Check your Progress 3

Ex 1. Write a program to find cube of numbers in a list using lambda function.

Ex 2. Write a program to add two given lists using map function.

Ex 3. Write a program to display vowels from a given list of characters using filter function.

11.8 FILE OPERATIONS

Main memory is volatile in nature, hence, nothing can be stored permanently. File handling is a very important functionality which must be provided by any language to deal with this problem. Inputs can be taken from files instead of users, output can be displayed and saved permanently in files which can further be accessed later and appended or updated. All these functionalities come under file handling. Like other languages, python also provides various built-in functions for file handling operations.

There are two types of files supported by python- Binary and Text.

Binary files – These are the files that can be represented as 0's and 1's. These files can be processed by applications knowing about the file's structure. Image files are example of binary files.

Text files – These files are organized as sequence of characters. Here, each line is separated by a special end of line character.

Any file handling operation can be performed in three steps-

1. Opening a file
2. Operating on file – read , write , append etc
3. Closing a file

1. Opening a file

A file can be opened in a Python using built-in function *open()*. By default the files are opened in read text ('r') file mode.

Syntax of open() function

```
file = open ( "file_name.ext ",''mode_of_operation'')
```

Where,

File_name.ext - is the name of the file to be opened and ext is the extension of file.

Mode_of_operation - is the mode in which file is needed to be opened.

file – is the object returned by the function. This object can be used for further operations on file

The possible mode of operations in python –

Mode of opening file	Functionality
'r'	Opens a file for reading. It is default mode
'w'	Opens a file for writing. Overwrites a file if already exists. Creates new file if does not exist.
'a'	Opens a file for appending.
'r+'	Opens a file for both reading and writing.
'w+'	Same as 'r+' but creates new file if does not exist and overwrites if exists
'a+'	Opens a file for appending and reading
'x'	Creates new file. Fails if already exists. Added in python 3.
'rb'	Opens a file for reading in binary mode.
'wb'	Same as 'w' except data is in binary
'ab'	Same as 'a' except data is in binary

2. Operating on file

There are various operations -

2.1 Reading from a file

There are many ways to read from a file. Given below are the functions available for reading from a file.
In the below table assume that *file* is the object returned from open() function.

Function	Syntax	Description
read()	file.read ()	Returns entire file contents as a string
	file.read (n)	Returns n characters from beginning of file as string
readline()	file.readline()	Returnssingle line of file at a time. First line in this case.
readlines()	file.readlines()	Returns a list of all lines

2.2 Writing to a file

Similar to read operations, functions are there in python to write data to file.

Function	Syntax	Description
write()	file.write("text")	Writes text or string to a file

2.3 Operations on file

Python allows many other operations to be done in with files. Listed below are some functions used for file handling.

Function	Syntax	Description
tell()	file.tell()	Returns the current cursor position in file
seek()	file.seek(pos)	Places the file cursor to the position pos
os.stat()	os.stat(filename)	This method is used to get display status of the file given as argument.

os.path.exists()	os.path.exists('arg')	Returns true if the file or directory of name 'arg' exist
os.path.isfile()	os.path.isfile('arg')	Returns true if the file of name 'arg' exists
os.path.isdir()	os.path.isdir('arg')	Returns true if the directory of name 'arg' exists
shutil.copy()	shutil.copy(src,dst)	Copies a file from src to des file.
os.path.getsize()	os.path.getsize(filename)	Returns the size of file

3. Closing a file

After completing all the operations on file, it must be closed properly. This step frees up the resources and allows graceful termination of file operations.

Syntax of closing a file

```
file.close()
```

11.8.1 Reading data from a file

There are various ways to read a file by following the steps explained in the above section. For reading data from the file, it must be existing. Below is given the program (Example 23) to read a file named "first.txt" in the directory files within the present directory.

Example 23: Program to display file contents

```
file = open ("files\\first.txt","r") #  step 1: opening file for reading
s = file.read()                      #  step 2: reading entire file
print(s)                             # displaying file contents
file.close()                         # step 3: closing file
```

```
============ RESTART: C:/Users/test/Desktop/python_programs/file.py ============
Python is an interpreted, high-level, general-purpose programming language. Crea
ted by Guido van Rossum and first released in 1991, Python's design philosophy e
mphasizes code readability with its notable use of significant whitespace. Its l
anguage constructs and object-oriented approach aim to help programmers write cl
ear, logical code for small and large-scale projects.[28]

Python is dynamically typed and garbage-collected. It supports multiple programm
ing paradigms, including structured (particularly, procedural,) object-oriented,
 and functional programming. Python is often described as a "batteries included"
```

It may be noted that the file that we want to read or write, may be present in some other directory or folder. In that case, we can give absolute or relatative path of that file along with its name, shown in the example below. The path separated by \ must

be proceeded by one more \ (backslash) to turn-off special feature of this character. If the file to be read or written is in the same folder, in which python program is present, we need not give its path.

11.8.2 Creating a file

For creation of new file, the file can be opened in 'w' mode. If the file already exists, the contents can be overwritten. Also, if we want to be sure that no existing file gets overwritten, then 'x' mode can be used to create new file. If the file already exists, it will show error message. Given below is the example to open a file and write contents to it.

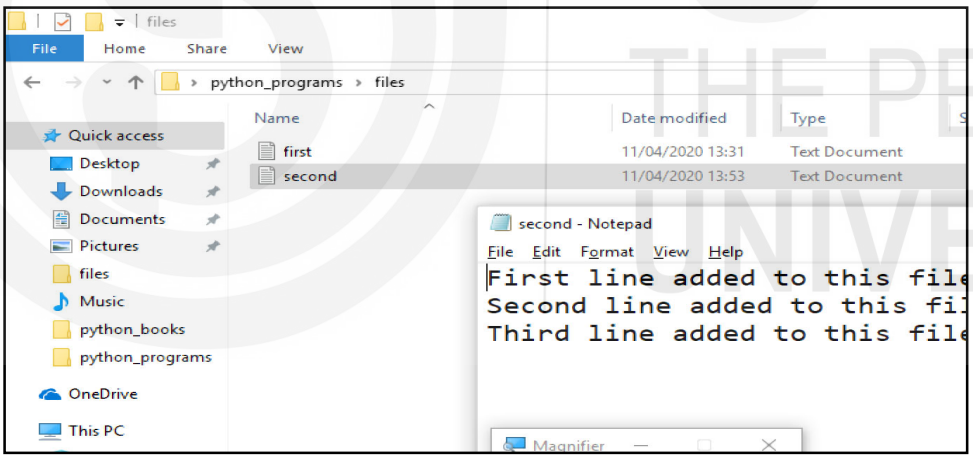
Example24 : Program to create file and add contents to it.

```
file = open ("files\\second.txt","w") #  step 1: opening file for reading

file.write("First line added to this file\n")# step 2: writing contents to file
file.write("Second line added to this file\n")
file.write("Third line added to this file\n")
file.close()                        # step 3: closing file
print("file created successfully")
```

```
============ RESTART: C:/Users/test/Desktop/python_programs/file.py ============
file created successfully
>>>
```

After executing this program a file name second.txt gets created in the mentioned drive with the added contents.



Reading, writing, appending with *with* statement

The methods of dealing with files used in the above section may not be safe sometimes. It may happen that an exception occurs as a result of operations on file and the code exits without closing the file. To make it more convenient python introduced another statement called *with* statement which ensures that file is closed when the code within *with* is exited. Call to close() function is not required explicitly. The syntax of how to use *with* statement for reading, writing and appending file is given below.

Syntax of read,write and append with *with* statement:

```
with open("file_name","r") as file:   # to read file
     file.read()

with open("file_name","w") as file:   # to write file
     file.write("contents to add")

with open("file_name","a") as file:   # to append file
     file.write("\n contents to append")
```

11.8.3 Copying a file

To copy a file from one to another various methods can be used. One such way of copying file is by importing a module named 'shutil' which contains a built-in function to create copy of a file.

Example 25 : Program to copy a file from 'second.txt' to 'third.txt'.

```
import shutil
shutil.copy("files\\second.txt","files\\third.txt")
```

11.8.4Deleting a file or folder

A file can be deleted using remove() function from os module. Before using this function you should move to the directory in which the file to be removed exists. Similarly, to delete a folder or directory, os.rmdir() function can be used. The directory to be removed must be empty or otherwise error will be shown. In the example given below, we want to delete a file name "delete_123.py". os.listdir() function is used to display the list of files present in a directory.

Example 26: Deleting a file.

```
>>> import os
>>> os.chdir("C:\\Users\\test\\Desktop\\python_programs")
>>> os.listdir()
['abc.py', 'Arbitrary.py', 'arguments.py', 'arg_test.py', 'compound_interest.py'
, 'delete_123.py', 'demo.py', 'demo2.py', 'factorial.py', 'file.py', 'file2.py',
 'files', 'lambda.py', 'modules_test.py', 'module_2.py', 'pack', 'path.py', 'ran
d_test.py', 'return_example.py', 'series.py', 'simple_interest.py', 'test.py', '
test_2.py', '__pycache__']
>>> os.remove("delete_123.py")
>>> os.listdir()
['abc.py', 'Arbitrary.py', 'arguments.py', 'arg_test.py', 'compound_interest.py'
, 'demo.py', 'demo2.py', 'factorial.py', 'file.py', 'file2.py', 'files', 'lambda
.py', 'modules_test.py', 'module_2.py', 'pack', 'path.py', 'rand_test.py', 'retu
rn_example.py', 'series.py', 'simple_interest.py', 'test.py', 'test_2.py', '__py
cache__']
>>>
```

Check your Progress 4

Ex 1. Write a program to display frequency of each word in a file.

Ex 2. Write a program to display first n lines from a file, where n is given by user.

Ex 3. Write a program to display size of a file in bytes.

11.9 SUMMARY

A **Function** is a block of code that performs a specific task. In this chapter, we have discussed various aspects of functions such as how to create functions, their scope, passing arguments to function, andLambda functions. In addition to these topics, file handling operations are also discussed in detailssuch as how to interact with file, copying, deleting, etc.

SOLUTIONS TO CHECK YOUR PROGRESS

Check your Progress 1

Ex 1. The various benefits of using functions are –

i) It decomposes a larger problem into more manageable units
ii) It allows reusability of code
iii) It reduces duplication
iv) It makes code easy to understand, use and debug

The process of defining a function is called *function definition*. It actually describes the working of a function. It includes – function name, list of arguments and function body.

```
Syntax  deffunction_name():
.....                    # body of function
......
```

Even if the function is defined, it can never be executed till the time it is called. Hence, for using a function, it must be called using *function call statement*. Calling of a function includes function name and list of arguments (no function body).

Syntax function_name()

Ex 2. Part of a code in which a variable can be accessed is called *Scope of a variable*.

Local variables are the variable created within a function's body or function's scope. They cannot be accessed outside the function. Their scope is only limited to the function in which they are created.
Global variables are the variable created outside any function. Their scope is global, hence can be used anywhere. Global variables can also be created within function using keyword *global*.

```
deff():
    s =1      # s is local variable
    print(s)
    r ="IGNOU" # r is global variable
f()
```

```
print(r)
```

Ex 3.

```
def pattern():
    for i in range(1,6):
        for j in range(1,i+1):
            print(j,end=' ')
        print()

pattern()
```

Check your Progress 2

Ex 1.Python provides various mechanisms of passing arguments to a function.

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

Ex 2.

```
def unique(numbers):
    out = []
    for i in numbers:
        if i not in out:
            out.append(i)
    return out

ans=unique([1,2,1,1,2,2,3,4,6,3])
print(ans)
```

Ex 3.

```
def prime(a,b):
    for i in range(a,b+1):
        if i==0 or i==1:
            continue
        test=0
        for j in range(2,i):
            if i%j==0:
                test=1
                break
        if test==0:
            print(i,end=' ')

prime(1,50)
```

Check your Progress 3

Ex 1.

```
nums = [1, 2, 3, 4, 5]
print("\nCube of numbers:")
cube = list(map(lambda x: x ** 3, nums))
print(cube)
```

Ex 2.

```
a = [1, 2, 3]
b = [10, 20, 6]

result = map(lambda x, y: x + y, a, b)
print("\nAddition of two list:")
print(list(result))
```

Ex 3.

```
def vowel(a):
     v=['a','e','i','o','u']
     if a in v:
          return a

l=['a','b','f','o','x','z','y']
print("LIST OF VOWELS:")
print(list(filter(vowel,l)))
```

Check your Progress 4

Ex 1.

```
from collections import Counter
def wordcount(fname):
        with open(fname) as f:
                return Counter(f.read().split())

print("Frequency of words:",wordcount("abc.txt"))
```

Ex 2.

```
n=int(input("enter number of lines:"))
c=1
file=open("test.txt","r")
for i in file:
     if c<=n:
          print(i)
          c+=1
file.close()
```

Ex 3.

```
def fsize(fname):
        import os
        status = os.stat(fname)
        return status.st_size

print("File size in bytes: ",fsize("test.txt"))
```