ਜਗਤ ਗੁਰੂ ਨਾਨਕ ਦੇਵ
ਪੰਜਾਬ ਸਟੇਟ ਓਪਨ ਯੂਨੀਵਰਸਿਟੀ
ਪਟਿਆਲਾ

JAGAT GURU NANAK DEV
PUNJAB STATE OPEN UNIVERSITY, PATIALA

(Established by Act No. 19 of 2019 of the Legislature of State of Punjab)

The Motto of the University

(SEWA)

SKILL ENHANCEMENT **EMPLOYABILITY** **WISDOM**
ACCESSIBILITY



DIPLOMA IN AI & DATA SCIENCE
SEMESTER-I
Course: Python Programming
Course Code: DAIDS-1-03T

ADDRESS: C/28, THE LOWER MALL, PATIALA-147001
WEBSITE: www.psou.ac.in

SELF-INSTRUCTIONAL STUDY MATERIAL FOR JGND PSOU, ALL COPYRIGHTS WITH JGND PSOU, PATIALA

DAIDS-1-03T: Python Programming

Total Marks: 100
External Marks: 70
Internal Marks: 30
Credits: 6
Pass Percentage: 40%

Course: Python Programming	
Course Code: DAIDS-1-03T	
Course Outcomes (COs) After the completion of this course, the students will be able to:	
CO1	Explain the basic syntax and structure of Python programs.
CO2	Understand variables, data types, and basic operations.
CO3	Understand and use common programming constructs like loops and conditionals.
CO4	Define and use functions in Python.
CO5	Understand the basics of object-oriented programming in Python.

Detailed Contents:

Module	Module Name	Module Contents
Module I	**Introduction to Python**	Python installation and setup, Command line Basics; Python Objects and Data Structures Basics: Introduction to Python data types, Variable assignments, Numbers, String, String methods, Lists, Python Comparison Operators: Chaining comparison operators with logical operators, Pass Break and continue.
Module II	**Program Flow Control in Python**	If, Elif and Else statements in python, for loops, While loops
Module III	**Functions in Python**	Introduction to functions, Def keyword, User defined functions, arguments and parameters, Parameter naming in python
Module IV	**Object Oriented Programming**	Introduction, Classes and objects, attributes and methods, Inheritance and polymorphism, Special methods; Modules and Packages: Pip install and PyPi.
Module V	**Use of Python Libraries**	Utilize common Python libraries for specific tasks (e.g., NumPy for numerical computing, Pandas for data manipulation). Use libraries for data manipulation, analysis, and visualization.
Module VI	**File handling in Python**	Files in python, importing own files, Read and writing text files, working with CSV, XML and JSON files.

Books

1. Timothy Budd, “Exploring Python”, TMH, 1st Ed, 2011
2. Allen Downey, Jeffrey Elkner, Chris Meyers, “How to think like a Computer Scientist: learning with Python”, Green Tea Pr, 2002
3. Paul Barry, “Head First Python: A Brain-Friendly Guide”, O′Reilly, 2nd ed. 2016
4. Udemy, https://www.udemy.com/course/complete-python-bootcamp/
5. Udemy,https://www.udemy.com/course/python-the-complete-python-developer-course

PYTHON PROGRAMMING

UNIT I: INTRODUCTION TO PYTHON

STRUCTURE

1.0 Objectives

1.1 Introduction

1.2 Introduction to Python

1.3 Python Usage

1.4 Development Environment

1.5 Python installation and Setup

- **1.5.1 Python installation on Windows**
- **1.5.2 Python Installation on Mac**
- **1.5.3 Python Installation on Linux**

1.6 Command Line Basics

1.7 Variables

- **1.7.1 Variables Reassignment**
- **1.7.2 Multi-variable Assignment**

1.8 Basic Data Types

- **1.8.1 Numeric**
- **1.8.2 String**
  - **1.8.2.1 String Methods**
- **1.8.3 Boolean**

1.9 Basic Data Structure in Python

1.9.1 Lists

1.9.2 Tupless

1.9.3 Dictionary

1.10 Self Check Question

1.11Summary

1.12 End Question

1.0 OBJECTIVES

- Learn the fundamentals of installing Python in different Operating Systems
- Work with different data types of Python
- Get deep insights about how to work with different data structures like lists, tuples, sets and dictionary.
- Students will be able to make programs related to the above-mentioned concepts.

1.1 INTRODUCTION

This module targets to inculcate python basics, data structures, and its operations in user minds so that they can build basic programs by using these concepts extensively. This module is developed by keeping the graduate learners in mind. Python is a very user-friendly language and is widely used by professional developers in different applications like artificial intelligence, web applications, and data analytics, etc. Python is recognized as one of the fastest-growing languages among different types of users. In the module, how to install python on different platforms is elaborated along with basic data types with appropriate programs so that learners will understand the concept. The main target of this module to cover data structures like Lists, tuples, dictionary, and sets with suitable programs.

1.2 PYTHON

Guide van Rossum (Dutch Programmer) is the creator of the Python programming language and it was invented in the year 1989. Its first public release was in 1991. Python is managed and distributed by Python Software Foundation. Python is very user-friendly, free to use and easy to learn if you are fresher in programming too [1]. This is popularly known as one of the powerful languages and open-source. Python is an interpreted and high-level language. Programmers have to focus on what to do in the tasks instead of how to do the tasks.

1.3 PYTHON USAGE

Python is used for the development of various applications in different fields. Some of these are mentioned below:

a) Web programming
b) Mobile Applications
c) Game development
d) Data Science and Data Visualization
e) Data Analytics
f) Frontend (GUI) development
g) Network programming
h) System Administrator
i) Machine learning and Artificial Intelligence
j) Web scrapping applications
k) Embedded applications

1.4 DEVELOPMENT ENVIRONMENT

There are three popular and common platforms. These are:

a) Terminal based or Shell-based
b) IDLE (Spyder IDE, PyCharm IDE)
c) Python notebook (Jupyter)

1.5 PYTHON INSTALLATION AND SETUP

→ If python is installed you can check the version of Python installed in your system by the following command.

C:\> python –version

→ To know the current location where your python is installed, you can use the below-mentioned command

C:\> where.exe python

If Python is not installed in your system then you have to install python according to your operating system. Below are the steps for installation according to your operating system.

1.5.1 Python installation on Windows

There are different ways of installing Python on windows and these are listed [2] as:

A) Package from Microsoft Store
B) Full Installer Package
C) Windows Subsystem for Linux

A) Package from Microsoft Store

One of the easiest, upfront and interactive ways of installing python on windows is with the help of app from the Microsoft store. If any of the beginners want to install Python, this is the only suggested way. Select the python version which you want to install on your machine and then click *GET* button on that suitable version, as it helps in downloading the python file. After that click on *INSTALL ON MY DEVICES* and select the device where you want to install the software. Now, select and click the button *INSTALLNOW*, after proper installation, you will see a message of installation congratulations.

B) Full Installer Package

If you are an advance or intermediate developer then this way of downloading from Python.org is a recommended step as it helps to control the things during installation set up. Go to the site and select the latest Python 3 release under Python Release for Windows and also select Windows x86-32 or Windows x86-64 executable installer and download the appropriate version. Now, double-click the downloaded file and look for the different options on the dialog box that appears after double-clicking the installer file. The first one is the default path which is set for

current Windows users. The second feature is to select for customization installation where users can select the features which they want to install like pip and IDLE also. The third one is to install a launcher for all users. This has a checkbox that is ticked default. It means all users can access py.exe whereas if you untick this checkbox, then only current can access py.exe. The fourth feature is to Add python to the path. This has a checkbox that is unticked by default and it is up to the user whether the user wants to add python in the environment path or not. After working on these, you can go to click on the button *INSTALL NOW* and after proper installation, you will see a message of installation congratulations.

C) Windows Subsystem for Linux (WSL)

You can run a virtual Linux in Windows system directly with the help of WSL. Python can be used with the help of the Linux platform here.

1.5.2 Python Installation on Mac

There are different ways of installing Python on macOS and these are listed [2] as:

A) Package from the official installer
B) Package Manager (Homebrew)

A) Package from the official installer

The most recommendable way of downloading is Python.org as it helps to control the things during installation set up. Go to the site and select the latest Python 3 release under Python Release for Mac OS X and also select macOS 64 bit executable installer and download the appropriate version. Now, double click the downloaded file and click *continue* button a few times until you reach software agreement and then click *Agree* button and you will see a dialog box where destination location with total space is shown and the user can change the destination location and then click on *INSTALL NOW* button and after proper installation, you will see a message of installation congratulations.

B) Package Manager (Homebrew)

First, install homebrew package manager unless ignore if it is already in the system. Go to browser and open it and type http://brew.sh/ and copy the command for install homebrew from there and open a Window terminal and paste that command and it will start homebrew installation process and suitably type your macOS password when it is asked in the installation process. This will take few minutes and after successful installation of this package manager, install python by using the command brew install python3.

1.5.3 Python Installation on Linux

There are different ways of installing Python on Linux and these are listed [2] as:

A) Package manager of machine OS

B) Building from source code

A) Package manager of machine OS

One of the easiest and popular methods for installing python on the Linux platform. This is done by running a command on the command line.

B) Building from source code

This is a typical harder method of installation when compared with the package installer method. This step involves a set of commands for installing python along with that take care of dependencies that are required for properly running the python code.

1.6 COMMAND LINE BASICS

The command-line interface is a text-based interpreter that helps in the interaction of the user with a running program. Executable commands are written on this and the appropriate function is done by the operating system according to the command. Python executable command is run by writing python program name in front of keyword python [3].

Example1: Message printing example and save the program as niceprog1.py (.py is the extension of python)

```
print("CSE-CA")

print("Mechatronics")
```

Now, how to run this program with the help of the command line and what will be the output of that command line.

Run the following command and press enter at end of the command:

```
C:\User\Python>python niceprog1.py
```

 # The output will be after running the above command is

```
CSE-CA

Mechatronics
```

If you want some data elements should be passed to the Python program using the command line then pass data elements values with python file name by putting up space as delimiter. These data elements that are used with space are known as command-line arguments.

Example 2: Command Line Arguments Program (Save this file as niceprog2.py)

```
import sys

print("The 1st argument in Command Line is",sys.argv[1])

print ("The 2nd argument in Command Line is",sys.argv[2])
```

Run the following command and press enter at end of the command:

C:\User\Python>python niceprog2.py “CSE-CA” “Mechatronics”

 # The output will be after running the above command is

The 1st argument in Command-Line is CSE-CA

The 2nd argument in Command-Line is Mechatronics

1.7 VARIABLES

Variables store the data and these are reserved memory spaces. There is no command available to declare variables unlike C/C++/Java has commands to declare variables. Here in python, only the „=„operator is used for assigning value to a variable.

Example 3:

```
str_BDay_Boy_Name="Vinay Stylish" #String variable

float_BDay_Boy_Age=21.6 #Float Variable

int_Bday_Boy_License_Num=12234562#Integer Variable

print(str_BDay_Boy_Name)

print(float_BDay_Boy_Age)

print(int_Bday_Boy_License_Num)
```

#Output

```
Vinay Stylish
21.6
12234562
```

NO7E: # is used to comment the line. # is used for single-line comment

1.7.1 Variables Reassignment

A variable can be assigned a value many times but the value that is assigned latest is considered for consideration.

Example 4: Variables Reassignment

```
float_BDay_Boy_Age=21.6 #Float Variable

float_BDay_Boy_Age=43.98

print(float_BDay_Boy_Age)
```

#Output

43.98

1.7.2 Multi-variable Assignment

When different variables are assigned with a same value that is the concept of multi-variable assignment.

Example 5: Multi-variable Assignment

```
int_var1=int_var2=int_var3=156 # Multivariable assignment

print(int_var1)

print(int_var2)

print(int_var3)

print(id(int_var1))  # id is used to get the memory address of the variable

print(id(int_var2))

print(id(int_var3))

int_var3=20

print(int_var3)

print(id(int_var3))
```

#Output

```
156
156
156
8790917690608
8790917690608
8790917690608
20
8790917686256
```

NOTE: Same value is pointing to the same reserved memory and variables int_var1, int_var2 and int_var3 having the same reserved memory as they point towards the same value. When the value is changed to 20 of int_var3, its memory address is also changed.

1.8 BASIC DATA TYPES

Data types are associated with every value. In object-oriented python, everything is treated as object. Objects are variables and classes are data types of these variables [4]. The basic data types are mentioned as:

1.8.1 Numeric
   i) Integer
   ii) Float
   iii) Complex
1.8.2 Strings
1.8.3 Boolean

1.8.4 Numeric

The numeric value is associated with numeric data type in python programming. The numeric data type can be integer, float or complex numbers. They are presented as int, float or complex respectively.

Integer numbers – These are whole numbers and can be positive or negative. No limit is restricted to how long can be an integer number. The memory is the only constraint for setting up the integer value.

Float numbers – These are real numbers with a fraction or decimal points. For scientific notation, e or E is used with integer numbers whether they are positive or negative.

Complex numbers – It consists of real and imaginary part.

Example 6: Numeric data type example

```
float_BDay_Boy_Age=21.6 #Float Variable

int_Bday_Boy_License_Num=12234562#Integer Variable

complex_Bday_Gift=2+5j # Complex number

print(type(float_BDay_Boy_Age))

print(type(int_Bday_Boy_License_Num))

print(type(complex_Bday_Gift))
```

#Output

```
<class 'float'>
<class 'int'>
<class 'complex'>
```

Example 7: Binary, Octal and Decimal Numbers Representation and their data type.

```
octal_var1=0o12

hexa_var2=0x16

binary_var3=0b1011
```

```
print(octal_var1)

print(type(octal_var1))

print(hexa_var2)

print(type(hexa_var2))

print(binary_var3)

print(type(binary_var3))
```

#Output

```
10
<class 'int'>
22
<class 'int'>
11
<class 'int'>
```

Example 8: Deep insights of floating points

```
float_var1=.56e7

float_var2=56.2e-3

print(float_var1)

print(float_var2)
```

#Output

```
5600000.0
0.0562
```

1.8.2 Strings

They are a group or sequence of characters and strings data types are represented as str. The use of single quotes and double quotes are used for strings [5].

Example 9: Usage of single quote and double quotes for printing strings

```
print('Take both vaccinations with a normal 4 to 6 weeks gap and say bye to COVID') # Usage of Single quotes

print("Do follow social distancing") # Usage of double quotes

print('Properly use mask')

print("Avoid unnecessary shopping and walking-out")
```

#Output

Take both vaccinations with a normal 4 to 6 weeks gap and say bye to COVID
Do follow social distancing
Properly use mask
Avoid unnecessary shopping and walking-out

Example 10: Another usage of single quotes and double quotes

```
print('Take both vaccination('')s with a normal 4 to 6 weeks gap and say bye to COVID') # Usage of Single quotes

print("Do follow(')s social distancing") # Usage of double quotes

print('Take both vaccination\''s with a normal 4 to 6 weeks gap and say bye to COVID') # Usage of Single quotes

print("Do follow\'s social distancing") # Usage of double quotes
```

#Output

Take both vaccination('')s with a normal 4 to 6 weeks gap and say bye to COVID
Do follow(')s social distancing
Take both vaccination''s with a normal 4 to 6 weeks gap and say bye to COVID
Do follow's social distancing

Example 11: Usage of triple single quotes for printing single and double quotes in one print statement.

```
print('''Take both vaccination(')s with a normal 4 to 6 weeks gap and say bye('') to COVID. Do follow social distancing''')
```

#Output

Take both vaccination(')s with a normal 4 to 6 weeks gap and say bye('') to COVID. Do follow s ocial distancing

Example 12: Usage of triple double quotes for printing single and double quotes in one print statement.

```
print("""Take both vaccination(')s with a normal 4 to 6 weeks gap and say bye('') to COVID. Do follow social distancing""")
```

#Output

Take both vaccination(')s with a normal 4 to 6 weeks gap and say bye('') to COVID. Do follow s ocial distancing

1.8.2.1 String Methods

i) String slicing

Accessing characters from the string is called string slicing. For slicing, the colon „:“ is used. For better understanding see example 12A.

ii) Updating strings

The whole string can be updated but the characters of strings cannot be updated. For better understanding see example 12A.

iii) Deleting strings

del keyword is used to delete the whole string as shown in example 12A.

Example 12A: Slicing, Updating and deleting operations in string

```
str_var='Take both vaccinations with a normal 4 to 6 weeks gap and say bye to COVID'

print(str_var)

print(str_var[5:14]) # Print elements from 5 to 13 and exclude 14

print(str_var[-12:-2])

# Print elements from -12 to -1 and exclude -1.

#At last, index is -1 value is D and -2 is I and -12 is b

str_var='Hey COVID, you will be completely over by 2022. I think!!!'

print(str_var)

str_var2='Hey COVID, you will be completely over by 2022. I think!!!'

print(str_var2)

del str_var2 # Deleted str_var2

str_var[0]='Z' # This cannot be done in string
```

#Output

```
Take both vaccinations with a normal 4 to 6 weeks gap and say bye to COVID
both vacc
bye to COV
Hey COVID, you will be completely over by 2022. I think!!!
Hey COVID, you will be completely over by 2022. I think!!!
TypeError: 'str' object does not support item assignment
```

iv) Formatting Method

format() method is used to do formatting in strings and {} braces are used for formatting and they act as placeholders for arguments. These can be positional as well as keyword arguments. See example 12B for the usage of the format() method.

Example 12B: Program to show deeper understanding of format method

```
str_var1="The first dose of {0} and second dose of {1} saves lives from attack of COVID {2}".format("injection1","injection2",2019)

print(str_var1)

print(type(str_var1))

str_var2="The first dose of {} and second dose of {} saves lives from attack of COVID {}".format("injection1","injection2",2019)

print(str_var2)

print(type(str_var2))

str_var3="The first dose of {val1} and second dose of {val2} saves lives from attack of COVID {val3}".format(val1="injection1",val2="injection2",val3=2019)

print(str_var3)

print(type(str_var3))

str_var4="{0:.3f}".format(2/7)  # Here.3f tell how many decimal points you need

print(str_var4)

print(type(str_var4))

str_var5="{0:o}".format(54) # Convert into octal format

print(str_var5)

print(type(str_var5))

str_var6="{0:b}".format(54) # Convert into binary format

print(str_var6)

print(type(str_var6))
```

#Output

```
The first dose of injection1 and second dose of injection2 saves lives from attack of COVID 201
9
<class 'str'>
```

The first dose of injection1 and second dose of injection2 saves lives from attack of COVID 201
9
<class 'str'>
The first dose of injection1 and second dose of injection2 saves lives from attack of COVID 201
9
<class 'str'>
0.286
<class 'str'>
66
<class 'str'>
110110
<class 'str'>

1.8.3) **Boolean**

This is one of the inbuilt data types in python programming and this data type can take only two values i.e. True or False. When checking of type of a variable is performed and it shows <"class bool"> then it is Boolean data type [6].

Example 13: Boolean Basic Understanding Program

```
var_bool1=True

var_bool2=False

print(type(var_bool1))

print(type(var_bool2))

print(type(var_bool1==var_bool2))
```

#Output

```
<class 'bool'>
<class 'bool'>
<class 'bool'>
```

NOTE: bool() function can be used to get the values in form of True or False

Example 14: bool() function usage

```
int_var1 = 19

print(bool(int_var1))

int_var2 = 0

print(bool(int_var2))
```

```
float_var3 = 17.89

print(bool(float_var3))

float_var4 = -17.89

print(bool(float_var4))
```

#Output

```
True
False
True
True
```

Explanation: Any value other than 0 is treated as True in the above example of bool() unless it is treated as False.

1.9 BASIC DATA STRUCTURES IN PYTHON

Data Structures help in managing, handling, dealing and storing the data. They help in traversing, modifying and updating the data. Different operations can be performed on the data with the help of data structures. This helps in controlling the functionalities involved in getting the proper data output.

The basic data structures of python are:

1.9.1 Lists
1.9.2 Tuples
1.9.3 Dictionary
1.9.4 Sets

1.9.1 Lists

This data structure helps in storing different data types. This is created using a square bracket []. The data types can be strings, other lists or integers. The list elements can be changed even after its creation. This is known as mutable property. Addresses are assigned to every element of the list, this is known as an index of the list. Two types of indexes are there one is positive and the other one is negative. While traversing in a list when the user starts from 0 to the last element of the list, it is called positive index. When the user traverse from the last (i.e. index=-1) to the start is called negative-index [7]. The different basic operations that can be performed on the list are mentioned as:

- The list can be created empty as well as initialized with different types of elements.
- Elements can be inserted in list using append(), extend() and insert() functions. These functions usage is explained in example 15.

- Elements can be deleted in the list using remove(), pop() and remove() functions. The del keyword is also used to delete elements in the list. The explanation is given in example 16.
- Accessing elements of the list is explained in example 17.

Example 15: Program for List creation (empty, initialization with elements) and Insertion in list (Usage of insert, extend and append functions) [7]

```
mixed_list1=[] # Empty list creation

print(type(mixed_list1)) # Check the data type

print(mixed_list1)  #  Printing the  empty list

mixed_list2 = [2,"injection1",23.56,"injection1",23.56]

# Adding heterogeneous elements to the mixed_list2

print(type(mixed_list2)) #Check the data type

print(mixed_list2) # Printing the mixed_list2 elements

mixed_list1.extend([2,"injection1"])

#Extend function is used to add elements one by one in the empty list (mixed_list1)

print(mixed_list1)

mixed_list1.append([23.56,"Wear Mask Properly","Rules for Social Distancing"])

#Append function is used to add all elements as single element in mixed_list1 and added at end of list

print(mixed_list1)

# INDEXING IN LIST STARTS FROM 0

# Adding element at INDEX 2 i.e. after value of injection1

mixed_list1.insert(2,"injection2") # injection2 is inserted at index 2

print(mixed_list1)
```

#Output

```
<class 'list'>
[]
<class 'list'>
[2, 'injection1', 23.56, 'injection1', 23.56]
[2, 'injection1']
```

[2, 'injection1', [23.56, 'Wear Mask Properly', 'Rules for Social Distancing']]
[2, 'injection1', 'injection2', [23.56, 'Wear Mask Properly', 'Rules for Social Distancing']]

Example 16: Program for deleting elements in the list using different ways [7]

```
mixed_list1=[2, 'injection1', 'injection2', 23.56, 'Wear Mask Properly', 'Rules for Social Distancing']

print(mixed_list1)

# Remove function is used to delete the elements using value and it deletes the first occurrence of value

mixed_list1.remove('Wear Mask Properly') # 'Wear Mask Properly' element is removed from the list

print(mixed_list1)

# Pop is used to remove the element from the list by passing index as the argument in pop function.

# If  index is not given then last element of the list is deleted

pop_element=mixed_list1.pop(3) # Element is removed at index 3 using pop function

print("popped element is",pop_element)

print(mixed_list1)


#del keyword is used to delete the element using the index value

del mixed_list1[2] # Element is deleted at index 1

print(mixed_list1)

# Clear function is used to remove the whole list

mixed_list1.clear()

print(mixed_list1)
```

#Output

[2, 'injection1', 'injection2', 23.56, 'Wear Mask Properly', 'Rules for Social Distancing']
[2, 'injection1', 'injection2', 23.56, 'Rules for Social Distancing']
popped element is 23.56
[2, 'injection1', 'injection2', 'Rules for Social Distancing']

[2, 'injection1', 'Rules for Social Distancing']
[]

Example 17: Program to show ways of accessing list elements

```
mixed_list1=[2, 'injection1', 'injection2', 23.56, 'Wear Mask Properly', 'Rules for Social Distancing']

print(mixed_list1) # Accessing all list elements

print(mixed_list1[5])

print(mixed_list1[3:5])

# Slicing Method is used here, [starting element index : end element range-1].

# [3:5] here denotes go from index 3 to 5-1 i.e. 4

# Access elements from index 3 to 4 and exclude 5 (Last element of range is excluded)

print(mixed_list1[-4:-2])

# When access from last, index values from last are -1,-2,-3 and so on

# When index is -4,by calculating from last it is "injection2"

# Index is -2,by calculating from last it is "Wear Mask Properly"

# Range is -4 to -2 i.e Access elements from -4 to -3 and exclude -2 (Last element of range is excluded)

# See below access elements of list using loop

for mixed_var1 in mixed_list1: # one by one elements of list are accessing

    print(mixed_var1)
```

#Output

```
[2, 'injection1', 'injection2', 23.56, 'Wear Mask Properly', 'Rules for Social Distancing']
Rules for Social Distancing
[23.56, 'Wear Mask Properly']
['injection2', 23.56]
2
injection1
injection2
23.56
Wear Mask Properly
Rules for Social Distancing
```

1.9.2 Tuples

Tuples exactly work like lists of python but they are immutable i.e. it elements cannot be changed. They are created using () or tuple() function. They also store heterogeneous elements. While creating () are optional as can be seen in example 18. The different basic operations that can be performed on the tuple are mentioned as [7]:

- Tuple can be created empty as well as initialized with different types of elements. It can be created with () brackets, without () brackets and tuple() function. Tuple() function has one optional argument that consists of an iterator like lists, tuples, dictionary, etc. See example 18 for a better understanding.
- Tuple concatenation using „+“ operator. An example is shown in 18.
- Tuple elements cannot be removed or deleted due to their immutable property. Immutable property is explained in example 19. But, the whole tuple can be deleted using „del“ keyword as shown in example 20.
- Slicing method used in tuples as mentioned in example 21.

Example 18: Tuple elements creation and accessing program

```
mixed_tuple1= () # Empty tuple creation

print(type(mixed_tuple1)) # Check the data type

print(mixed_tuple1) # Printing the empty tuple

# Adding heterogeneous elements to the mixed_tuple2

mixed_tuple2 = (2,"injection1",23.56,"injection1",23.56)

print(type(mixed_tuple2)) #Check the data type

print(mixed_tuple2) # Printing the mixed_tuple2 elements

# Without bracket creating tuple elements

mixed_tuple3 = 2,"injection1",23.56,"injection1",23.56

print(type(mixed_tuple3)) #Check the data type

print(mixed_tuple3) # Printing the mixed_tuple3 elements

# For concatenating elements in tuple '+' is used

mixed_tuple3=mixed_tuple3 + ("bye_covid",576,"huge effort required")

print(mixed_tuple3) # Printing the mixed_list2 elements

# Creation of tuple elements using tuple function

mixed_tuple4= tuple()  # Empty tuple creation with no argument in tuple() function
```

```
print(type(mixed_tuple4)) # Check the data type

print(mixed_tuple4) # Printing the empty tuple

# tuple() is used when list is passed as argument

mixed_list =[2,3,17,67]

mixed_tuple4= tuple(mixed_list)  # In tuple()function, list is passed as argument

print(type(mixed_tuple4)) # Check the data type

print(mixed_tuple4) # Printing the empty tuple
```

#Output

```
<class 'tuple'>
()
<class 'tuple'>
(2, 'injection1', 23.56, 'injection1', 23.56)
<class 'tuple'>
(2, 'injection1', 23.56, 'injection1', 23.56)
(2, 'injection1', 23.56, 'injection1', 23.56, 'bye_covid', 576, 'huge effort required')
<class 'tuple'>
()
<class 'tuple'>
(2, 3, 17, 67)
```

Example 19: Program to show immutable property of tuples

```
mixed_tuple2 = (2,"injection1",23.56,"injection1",23.56)

mixed_tuple2[1]="vaccination1"  # This cannot be done as it tries to change element value of tuple.

print(mixed_tuple2)
```

#Output

TypeError: 'tuple' object does not support item assignment

Example 20: Program to delete the whole tuple using „del“ keyword

```
mixed_tuple2 = (2,"injection1",23.56,"injection1",23.56)

print(mixed_tuple2)

del mixed_tuple2

print(mixed_tuple2)
```

#Output

(2, 'injection1', 23.56, 'injection1', 23.56)

NameError: name 'mixed_tuple2' is not defined

Example 21: Program for slicing and accessing elements of tuples

```
mixed_tuple2 = (2,"injection1",23.56,"injection1",23.56)
print(mixed_tuple2[1:3])
print(mixed_tuple2[-4:-1])
print(mixed_tuple2[:])
print(mixed_tuple2[:3])
print(mixed_tuple2[::-1])
for mixed_var1 in mixed_tuple2:
    print(mixed_var1)
```

#Output

```
('injection1', 23.56)
('injection1', 23.56, 'injection1')
(2, 'injection1', 23.56, 'injection1', 23.56)
(2, 'injection1', 23.56)
(23.56, 'injection1', 23.56, 'injection1', 2)
2
injection1
23.56
injection1
23.56
```

1.9.3 Dictionary

In python programming, the dictionary can be created using {} brackets and it consists of keys and values where two keys cannot be the same in a dictionary but values can be repeated and of different datatype in a dictionary, Keys are immutable. Many keys and values can be there in the dictionary but they are separated using a comma operator as shown in syntax. Nesting of dictionaries is also possible like lists.

Syntax:

Dictionary_name = { key1:value, key2:value ……. Key n:value}

The different basic operations that can be performed on the dictionary are mentioned as [7]:

- Dictionary creation and accessing elements as mentioned in example 22. Empty dictionary and elements in the dictionary are created using {} and dict() function is also used to create a dictionary. Accessing elements of dictionary can be used keys and get() function.
- Updation and insertion elements in the dictionary as shown in example 23.
- Deletion of elements in a dictionary using del keyword, clear() function, pop() function as shown in example 24.
- Nesting of dictionaries as shown in example 25.

Example 22: Creation of dictionary

```
base_dnary={} # Empty dictionary creation
print(base_dnary)
mixed_dnary = {'BdayBoyName':'XYZee', 2:'injections', 2019:'Year'}
print(mixed_dnary) # Whole dictionary is printed
print(mixed_dnary[2])# key is 2nd and its value is printed
print(mixed_dnary['BdayBoyName'])#Key value of 'BdayBoyName' is printed
print(mixed_dnary.get(2)) # get method is used to access value
mixed_dnary2=dict({'BdayBoyName':'XYZee', 2:'injections', 2019:'Year'})
print(mixed_dnary2)
print("Using keys() method")
for key_elements in mixed_dnary2.keys():  # Accessing elements using keys() method
  print (key_elements, mixed_dnary2[key_elements])
print("Using items() method")
for key_elements, val_elements in mixed_dnary2.items(): # items() method
  print (key_elements, val_elements)
print(mixed_dnary[0])
#Generate error as key 0 is not present and it does not take index 0 into consideration
```

#Output

```
{}
{'BdayBoyName': 'XYZee', 2: 'injections', 2019: 'Year'}
```

injections
XYZee
injections
{'BdayBoyName': 'XYZee', 2: 'injections', 2019: 'Year'}
Using keys() method
BdayBoyName XYZee
2 injections
2019 Year
Using items() method
BdayBoyName XYZee
2 injections
2019 Year
KeyError: 0

Example 23: Program for inserting and updating elements in dictionary [8]

```
mixed_dnary = {'BdayBoyName':'XYZee', 2:'injections', 2019:'Year'}

print(mixed_dnary)

mixed_dnary[2]='doses of injections' # Updating elements using key

print(mixed_dnary)

mixed_dnary['Mask']='Wear properly' # Insert new key and value in dictionary

print(mixed_dnary)
```

#Output

{'BdayBoyName': 'XYZee', 2: 'injections', 2019: 'Year'}
{'BdayBoyName': 'XYZee', 2: 'doses of injections', 2019: 'Year'}
{'BdayBoyName': 'XYZee', 2: 'doses of injections', 2019: 'Year', 'Mask': 'Wear properly'}

Example 24: Program to show deletion of elements in a dictionary

```
mixed_dnary = {'BdayBoyName': 'XYZee', 2: 'doses of injections', 2019: 'Year', 'Mask': 'Wear properly'}

mixed_dnary2= {'BdayBoyName': 'XYZee', 2: 'doses of injections', 2019: 'Year', 'Mask': 'Wear properly'}

print(mixed_dnary)

print(mixed_dnary2)

del mixed_dnary['Mask'] # deleting a particular key and value using 'key'
```

```
print(mixed_dnary)

mixed_dnary2.pop(2019) # pop(key) put key value as argument to delete key and value from dictionary

print(mixed_dnary2)

mixed_dnary2.popitem() # popitem() randomly delete any key and value from dictionary

print(mixed_dnary2)

mixed_dnary2.clear()# Removing all elements of 2nd dictionary using clear function

print(mixed_dnary2)

del mixed_dnary # Whole dictionary is deleted

print(mixed_dnary)
```

#Output

```
{'BdayBoyName': 'XYZee', 2: 'doses of injections', 2019: 'Year', 'Mask': 'Wear properly'}
{'BdayBoyName': 'XYZee', 2: 'doses of injections', 2019: 'Year', 'Mask': 'Wear properly'}
{'BdayBoyName': 'XYZee', 2: 'doses of injections', 2019: 'Year'}
{'BdayBoyName': 'XYZee', 2: 'doses of injections', 'Mask': 'Wear properly'}
{'BdayBoyName': 'XYZee', 2: 'doses of injections'}
{}
NameError: name 'mixed_dnary' is not defined
```

Example 25: Program to show the concept of nesting of dictionary

```
mixed_new_dnary1={'Trade':'CSE-CA',2:123,'E2':234,4:{'1st':'basic','2nd':888,3:'style'}}

print(mixed_new_dnary1)

print(mixed_new_dnary1[2]) # Accesing value when key is 2

print(mixed_new_dnary1[4]['1st']) # Accesing value when key is '4' and again its key is 1st

print(mixed_new_dnary1[4]) # Accessing value even if it is dictionary while using key '4'
```

#Output

```
{'Trade': 'CSE-CA', 2: 123, 'E2': 234, 4: {'1st': 'basic', '2nd': 888, 3: 'style'}}
123
basic
{'1st': 'basic', '2nd': 888, 3: 'style'}
```

1.9.4 Sets

Sets are a collection data structure and these are unordered (i.e. elements cannot be accessed using the index), mutable. Sets do not contain duplicate elements i.e. they are unique.

Syntax:

mixed_set = set(iterable_object)

where iterable_object can be strings, list, tuple and any other iterable object.

Example 26: Program to understand basic concept of sets

```
mixed_set=set(["injection1","Wear Mask",2, "Social Distancing"])
print(mixed_set) # Look at its output, it is unordered
print(type(mixed_set))
mixed_set.add("injection2") # Add new element in set
print(mixed_set) # Look at its output, it is unordered
print(len(mixed_set)) # len function is used to know the length of set
```

#Output

```
{'Wear Mask', 2, 'Social Distancing', 'injection1'}
<class 'set'>
{2, 'Social Distancing', 'injection1', 'injection2', 'Wear Mask'}
5
```

1.10 SELF-CHECK QUESTIONS

A) Which of the options are true for the following statements:

Statement 1: List is mutable

Statement 2: Tuple is mutable

Statement 3: List is immutable

Statement 4: Tuple is immutable

a) Statement 1 and Statement 2 are correct
b) Statement 1 and Statement 4 are correct
c) Statement 2 and Statement 3 are correct
d) All are correct

B) What will be the output of the following code?

```
print(("CSE-CA ")+("Mechatronics"))
```

a) CSE-CA Mechatronics
b) CSE-CAMechatronics
c) Mechatronics
d) CSE-CA

C) Choose the correct option where indexing is not valid for doing opertaions.

a) Lists
b) Dictionary
c) Strings
d) Tuples

D) What will be the output?

```
str_var4="{0:.3f}".format(8/11)
print(str_var4)
```

a) 0.727
b) 0.728
c) 0.726
d) 0.730

E) Print the appropriate output of the following code

```
mixed_list1=[2, 'injection1', 'injection2', 23.56, 'Wear Mask Properly', 'Rules for Social Distancin
g']
print(mixed_list1[-3])
```

a) 23.56
b) injection2
c) Rules for Social Distancing
d) Wear Mask Properly

1.11 SUMMARY

This module helps the students in understanding how different types of variables are used and what type of operations can be performed on these different variables along with appropriate examples. Deep discussions about lists, tuples, dictionary, and sets have been done. Different operations related to these data structures have been explained along with the output of different examples. This module helps the students in building up the basic blocks of python that are required for doing hard problems or competitive problems at later stages.

1.12 UNIT END QUESTIONS

1) Construct a program to add two lists of the same size having integer numbers.

2) Which is the correct option for the following code (Note: Read carefully)

```
mixed_tuple=[55,34,89,165]
```

mixed_tuple.pop(2)

print(mixed_tuple)

a) Error

b) [55, 34, 165]
c) [55, 34, 89]
d) [55, 89, 165]

3) Correct way of selecting 67 as output from the code is:

mixed_tuple = ("COVID_19", [13, 25, 36, 67], (15, 675, 543))

a) print(mixed_tuple[1][3])
b) print(mixed_tuple[2][3])
c) print(mixed_tuple[2][2])
d) print(mixed_tuple[1][2])

4) List down the operation that can be performed on dictionary with appropriate examples.

5) A tuple in python can be created without () brackets (True/False)

6) What will be the output of the below-mentioned code

mixed_tuple = 13, 17, 23, 78, 141

mixed_tuple[3] = 90

print(mixed_tuple)

a) (13, 17, 23, 78, 141)
b) (13, 17, 23, 90, 141)
c) (13, 17, 90, 78, 141)
d) Error

7) What will be the output of the below-mentioned code

mixed_list1 = 13, 17, 23, 78, 141

mixed_list1[3] = 90

print(mixed_list1)

a) [13, 17, 23, 78, 141]
b) [13, 17, 23, 90, 141]
c) [13, 17, 90, 78, 141]
d) Error

REFERENCES

[1] https://beginnersbook.com/2018/01/introduction-to-python-programming/

[2] https://realpython.com/installing-python/

[3] https://www.tutorialspoint.com/How-do-we-access-command-line-arguments-in-Python

[4] https://www.programiz.com/python-programming/variables-datatypes

[5] https://realpython.com/python-data-types/

[6] http://localhost:8888/notebooks/Untitled30.ipynb?kernel_name=python3

[7] https://www.edureka.co/blog/data-structures-in-python/

[8] https://www.tutorialspoint.com/python/python_dictionary.htm

UNIT II: PYTHON COMPARISON OPERATORS

STRUCTURE

2.0 Objectives

2.1 Introduction

2.2 Logical Operators

- **2.2.1 Types of Logical Operators**
- **2.2.1.1 Logical AND**
- **2.2.1.2 Logical OR**
- **2.2.1.3 Logical NOT**

2.3 Precedence of Logical Operators

2.5 Order of Evaluation of Logical Operators

2.5 Comparison Operators

- **2.5.1 Chaining of Comparison Operators**

2.6 Usage of break, continue and pass statements

- **2.6.1 Break Statement**
- **2.6.2 Continue Statement**
- **2.6.3 Pass Statement**

2.7 Practice Questions

2.8 Summary

2.0 OBJECTIVES

- Having knowledge about logical operators
- Students will be able to understand chaining comparison operators.
- Students will be inculcated concepts like break, continue and pass statements deeply and clearly.
- Students can build programs using relational, logical, identity and membership operators.

2.1 INTRODUCTION

This module helps the students in better understanding of logical operators. The different types of logical operators (and, or, not) are elaborated with many programming examples. Truth tables along with flowcharts are also mentioned for clearly understand the logical operators. The next topic is followed with precedence of logical operators and appropriate programs are mentioned to explain this topic briefly. This unit also targets to explain chaining comparison operators. Before deeply understanding this concept, firstly comparison operators, identity operators and membership operators are explained and then these are explained with chaining programming examples also. The difference between equal to operator and is operator is clearly explained followed with not equal to operator and is not operator. For explaining both differences clearly, python programming examples are given. Statements like break, continue and pass are discussed with programming examples. Distinction between pass and comments ae also assessed and mentioned. These whole things are necessary for doing complex problems and projects. This unit is followed with self-checked questions and summary of the whole topic. At last, practice questions are mentioned.

2.2 LOGICAL OPERATORS

It is used when decision making has to be performed on multiple conditions. Condition is the operand and it is assessed with a True of False. Example: have_injection1 and have_injection2

2.2.1 Types of Logical Operators

In Python, three logical operators are used for conditional statements. The operators with its symbols are mentioned in Table 2.1.

Table 2.1: Logical operators with symbols

Operator Symbol	Name	Narrative (Comparison between two conditions)
and	Logical AND	1. Returns True if both operands are True 2. Returns False if one of the operands is False
Or	Logical OR	1. Returns True if one of the operands is True 2. Returns False if both operands are False
Not	Logical NOT	1. Returns True if operand is False 2. Returns False if operand is True

2.2.1.1 Logical AND

Logical AND operator returns True if both operands are True unless it returns False. The truth table is mentioned in table 2.2 and its flow diagram is shown in figure 2.1.

Table 2.2: Truth Table for Two Conditions using Logical AND

Condition1	Condition2	Output (Condition1 or Condition2)
True	True	True
True	False	False
False	True	False
False	False	False

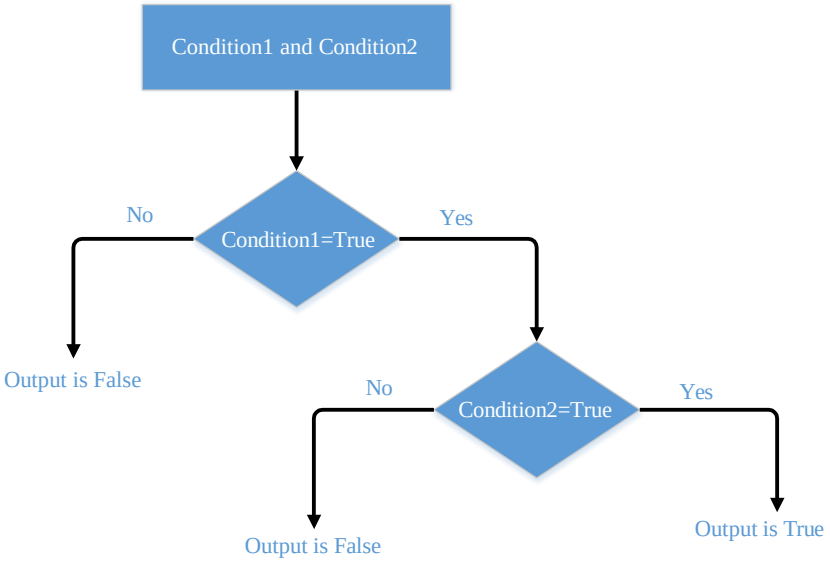


Figure 2.1: Flowchart: Logical AND

Example 1: Use of Logical AND

#variables with initial values

have_injection1=True

have_injection2=True

have_fever=False

have_headache=False

 # Condition with use of LOGICAL AND

have_injection1 and have_injection2

 # Output

True

 # Condition with use of LOGICAL AND

have_injection1 and have_fever

#Output

False

 # Condition with use of LOGICAL AND

have_fever and have_injection2

#Output

False

 # Condition with use of LOGICAL AND

have_fever and have_headache

#Output

False

Example 2: Make a python program in which get the input from user as CGPA of student, if CGPA is between 8 to 10 (including both 8 and 10) then print is message ‘Outstanding’ and if it is between 6 to 8 (including 6 only) then print a message ‘Average’ and if it lies between 4 to 6 (including 4 only) then print a message ‘Hard Work Needed’ and if it lies between 0 to 4 then print a message ‘Fail’.

Solution:

```
CGPA = float(input("Enter CGPA "))  #Input for CGPA from the user
if CGPA >=8 and CGPA <=10:    # Check CGPA between 8 to 10, (including 8 and 10 both)
   print("Outstanding")
if CGPA >=6 and CGPA <8:       # Check CGPA between 6 to 8, (including 6 only)
   print("Good Job")
if CGPA >=4 and CGPA <6:       # Check CGPA between 4 to 6, (including 4 only)
   print("Hard Work Needed")
if CGPA >=0 and CGPA <4:       # Check CGPA between 0 to 4, (including 0 only)
   print("Fail")
if CGPA < 0:                   # Check CGPA should not be less than 0 then print Wrong Input
   print("Wrong Input")
if CGPA > 10:                  # Check CGPA should not be more than 10 then print Wrong
Input
   print("Wrong Input")
```

Test Case 1:
Enter CGPA 8.1
#Output
Outstanding
Test Case 2:
Enter CGPA -1
#Output
Wrong Input
Test case 3:
Enter CGPA 11.3
#Output
Wrong Input

2.2.1.2 Logical OR

Logical OR operator returns True if any one of the operand is True unless it returns False. The truth table is mentioned in table 2.3 and its flow diagram is shown in figure 2.2.

Table 2.3: Truth Table for Two Conditions Using Logical OR

Condition1	Condition2	Output (Condition1 or Condition2)
True	True	True
True	False	True
False	True	True

False	False	False

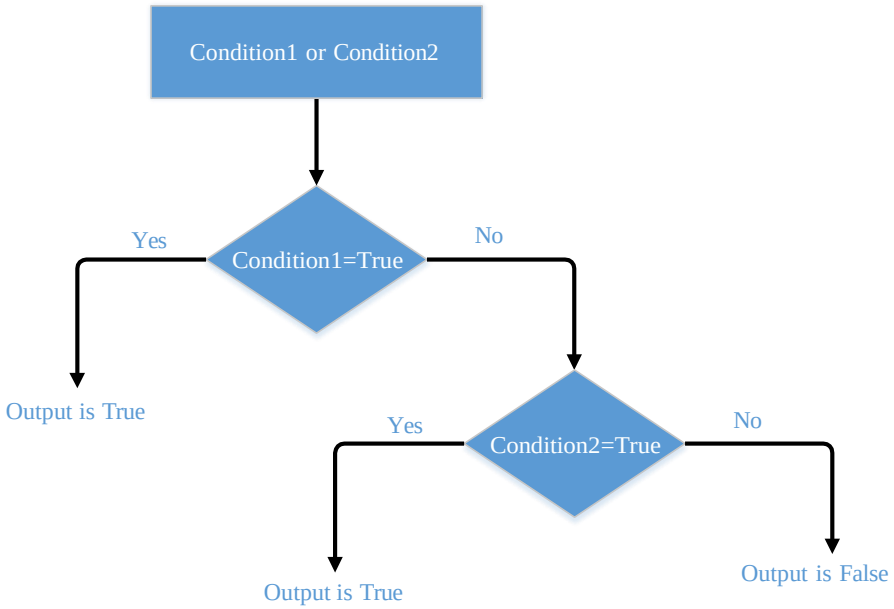


Figure 2.2: Flowchart: Logical OR

Example 3: Use of Logical OR

#variables with initial values

have_injection1=True

have_injection2=True

have_fever=False

have_headache=False

 # Condition with use of LOGICAL OR

have_injection1 or have_injection2

 # Output

True

 # Condition with use of LOGICAL OR

have_injection1 or have_fever

#Output

True

 # Condition with use of LOGICAL OR

have_fever or have_injection2

#Output

True

 # Condition with use of LOGICAL OR

have_fever or have_headache

#Output

False

Example 4:

Make a python program and input three integer numbers from the user if either one of the number is positive then print 'Atleast one of them is positive' else print 'All three numbers are negative'

Solution:

```
number1 = int(input())          #Input first number from the user
number2 = int(input())          #Input second number from the user
number3 = int(input())          #Input third number from the user
if number1>=0 or number2 >= 0 or number3>=0:     # Check if any number is positive
   print("Atleast one of them is positive")
else:
   print("All three numbers are negative")
```

Test Case 1:

100
-300
-900

#Output

Atleast one of them is positive

Test Case 2:

-700
-12300
-18000

#Output

All three numbers are negative

Test Case 3:

0
-12
-23

#Output

Atleast one of them is positive

2.2.1.3 Logical NOT

Logical NOT operator returns True if the operand is False and returns False if the operand is True. The truth table is mentioned in table 2.4 and its flow diagram is shown in figure 2.3.

Table 2.4: Truth Table for Logical NOT

Condition	Output (not(Condition))
True	False
False	True

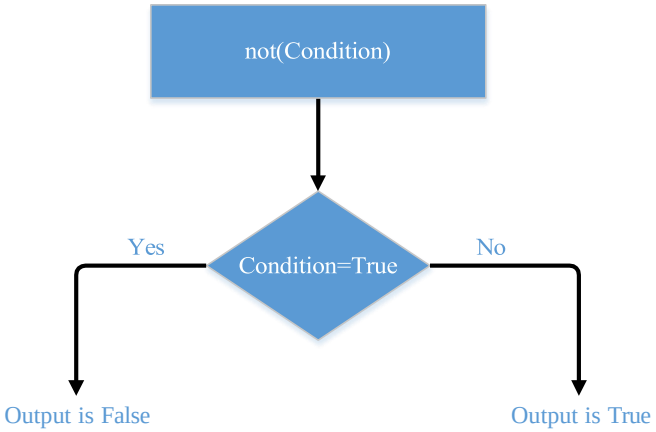


Figure 2.3: Flowchart: Logical NOT

Example 5: Use of Logical NOT

#variables with initial values

have_injection1 = True

have_fever = False

 # Condition with use of LOGICAL NOT

not(have_injection1)

 # Output

False

 # Condition with use of LOGICAL NOT

not(have_fever)

 # Output

True

2.3 PRECEDENCE OF LOGICAL OPERATORS

Table 2.5 shows the precedence of logical operators.

Table 2.5: Precedence of Logical Operators

Operator Name	Symbol	Precedence Level
Logical NOT	Not	1
Logical AND	And	2
Logical OR	Or	3

Example 6:

#variables with initial values

have_injection1=True

have_injection2=True

have_fever=False

have_headache=False

#Condition with Multiple Logical Operators

```
if have_fever == True or have_injection1==True and not(have_headache):
    print("CovidChances")
else:
```

```
    print("NoCovidChances")
```
#Output
CovidChances
Explanation of Example 6:
In above example 6, first 'not' operator is executed for not(have_headache) and the value becomes True, then 'and' operator is executed for (have_injection1==True and True) and the value becomes True and at last 'or' operator is executed for have_fever or True and the value becomes True, the final output is CovidChances.

Example 7:
#variables with initial values
```
have_injection1=True
have_injection2=True
have_fever=False
have_headache=False
```
#Condition with Multiple Logical Operators
```
if have_fever == True and have_injection1==True or have_headache==True:
    print("CovidChances")
else:
    print("NoCovidChances")
```
#Output
NoCovidChances
Explanation of Example 7:
In above example 7, first 'and' operator is executed for (have_fever == true and have_injection1==True) and the value becomes False, then 'or' operator is executed for (False or have_headache==True) and the value becomes False and the final output is NoCovidChances.

2.4 ORDER OF EVALUATION OF LOGICAL OPERATORS

Python evaluates the expression from left to right if there are multiple operators in the expression.

Example 8:
```
#variables with initial values
have_injection1=True
have_injection2=True
have_fever=False
have_headache=False
#Condition with Multiple Logical Operators
if have_headache == True or have_fever==True and have_injection1==True and
have_injection2==True:
    print("CovidChances")
else:
    print("NoCovidChances")
#Output
```

NoCovidChances

Explanation: In example 8, firstly ‘and’ operator is implemented as it has higher precedence then all other operators. There are two ‘and’ operators in above expression, so order of evaluation of logical operators is from left to right, so firstly (have_fever==True and have_injection1) is evaluated and its output is False and then this output is evaluated as (False and have_injection2==True) and the output is again False and then last operation is performed i.e. (have_headache == True or False) and for the final output else part will work and it is NoCovidChances.

Example 9:

```
#variables with initial values
have_injection1=True
have_injection2=True
have_fever=False
have_headache=False
#Condition with Multiple Logical Operators
if have_headache == True or not(have_fever==True) and have_injection1==True and
have_injection2==True:
   print("CovidChances")
else:
   print("NoCovidChances")
#Output
CovidChances
```

Explanation: In example 9, firstly ‘not’ operator is executed as it has higher precedence then all other operators in the expression and the output of not(have_fever==True) becomes True and then ‘and’ operator is implemented as (True and have_injection1==True) and the output is true and then this output is used as (True and have_injection2==True) and the output is True and then it is used as (have_headache==True or True) and the final output is True so else part is executed and it results in CovidChances.

Example 10: Make a python program to find out a year is leap year or not using Logical operators.

Basic Understanding: First of understand the basics of leap year, a year always divided by 4 is not a leap year. Surprised, yes but it is true. A year when divided by 4 it should not be divided by 100 then it is a leap year. If it is divided by 100 then it should be divided by 400 for being a leap year unless it is not a leap year

Solution:

```
# Python Program to check whether a year is leap year or not using Logical Operators
normal_year = int(input())
if normal_year%400 == 0 or normal_year%4 == 0 and normal_year%100 != 0:
   print("Leap Year")
else:
   print("Not a Leap Year")
```

Hints: firstly ‘and’ operators is executed and then ‘or’ operator is executed
Test case 1:
1008
#Output
Leap Year
Test case 2:
2000
#Output
Leap Year
Test case 3:
1900
#Output
Not a Leap Year

2.5 COMPARISON OPERATORS

These operators compare the operand values and on the basis of these results, it returns True or False. The list of comparison operators used in python are:

a) < (less than operator)

It checks condition whether left hand side value is less than right hand side value. After comparing these values, it returns True or False. If condition is True then it returns True else False.

Example:

i) 17 < 23
Output: True

ii) 28 < 23
Output: False

b) > (greater than operator)

It checks condition whether left hand side value is greater than right hand side value. After comparing these values, it returns True or False. If condition is True then it returns True else False.

Example:

i) 17 > 23
Output: False

ii) 28 > 23
Output: True

c) <= (less than and equal to operator)

It checks condition whether left hand side value is less than and equal to right hand side value. After comparing these values, it returns True or False. If condition is True then it returns True else False.

Example:

i) 17 <= 23
Output: True

ii) 28 <= 23
Output: False
iii) 17 <= 17
Output: True

d) >= (greater than and equal to operator)

It checks condition whether left hand side value is greater than and equal to right hand side value. After comparing these values, it returns True or False. If condition is True then it returns True else False.

Example:

i) 17 >= 23
Output: True
ii) 28 >= 23
Output: False
iii) 17 >= 17
Output: True

e) == (equal to operator)

It checks condition whether left hand side value is equal to right hand side value. After comparing these values, it returns True or False. If condition is True then it returns True else False.

Example:

i) 17 == 23
Output: False
ii) 180== 180
Output: True

f) != (Not equal to operator)

It checks condition whether left hand side value is not equal to right hand side value. After comparing these values, it returns True or False. If condition is True then it returns True else False.

Example:

i) 17 != 23
Output: True
ii) 180!= 180
Output: False

g) is (identity operator)

is operator is used to check whether both operands point to the same object or not. This is different from ==. In equal to operator, it is checked whether both operand have same value but here is operator checks whether both operands point to the same object or not [1].

Example :

```
#List1 is initialised with four values
mixed_list1 = [34,45,89,123]
```

```
# List2 is initialised with four values
mixed_list2 = [34,45,89,123]
# List 3 is assigned values of List1 (Note: Both will have same address)
mixed_list3 = mixed_list1
#Printed the elements of List1
print(mixed_list1)
#Printed the elements of List2
print(mixed_list2)
#Printed the elements of List3
print(mixed_list3)
# Here address of List1 is printed
print(id(mixed_list1))
# Here address of List1 is printed
print(id(mixed_list2))
# Here address of List1 is printed
print(id(mixed_list3))
# Here, values are checked of both operands
if mixed_list1 == mixed_list2:
   print("Both mixed-list1 and mixed_list2 have equal value")
else:
   print("Both mixed-list1 and mixed_list2 have not equal value")

if mixed_list1 == mixed_list3:
   print("Both mixed-list1 and mixed_list3 have equal value")
else:
   print("Both mixed-list1 and mixed_list3 have not equal value")

if mixed_list2 == mixed_list3:
   print("Both mixed-list2 and mixed_list3 have equal value")
else:
   print("Both mixed-list2 and mixed_list3 have not equal value")

# Here checking of operands point to the same object or not
if mixed_list1 is mixed_list2:
   print("Both mixed-list1 and mixed_list2 point to the same object")
else:
   print("Both mixed-list1 and mixed_list2 do not point to the same object ")

if mixed_list1 is mixed_list3:
   print("Both mixed-list1 and mixed_list3 point to the same object")
else:
   print("Both mixed-list1 and mixed_list3 do not point to the same object ")

if mixed_list2 is mixed_list3:
```

```
        print("Both mixed-list2 and mixed_list3 point to the same object")
    else:
        print("Both mixed-list2 and mixed_list3 do not point to the same object ")
```

#Output

```
[34, 45, 89, 123]
[34, 45, 89, 123]
[34, 45, 89, 123]
89447112
89448072
89447112
Both mixed-list1 and mixed_list2 have equal value
Both mixed-list1 and mixed_list3 have equal value
Both mixed-list2 and mixed_list3 have equal value
Both mixed-list1 and mixed_list2 do not point to the same object
Both mixed-list1 and mixed_list3 point to the same object
Both mixed-list2 and mixed_list3 do not point to the same object
```

h) is not (identity operator)

is not operator is used to check whether both operands do not point to the same object . This is different from !=. In not equal to operator, it is checked whether both operand have not same value but here is operator checks whether both operands do not point to the same object.

Example:

```
#List1 is initialised with four values
mixed_list1 = [34,45,89,123]

# List2 is initialised with three values
mixed_list2 = [34,89,123]

# List 3 is assigned values of List1 (Note: Both will have same address)
mixed_list3 = mixed_list1
#Printed the elements of List1
print(mixed_list1)
#Printed the elements of List2
print(mixed_list2)
#Printed the elements of List3
print(mixed_list3)
# Here address of List1 is printed
print(id(mixed_list1))
# Here address of List1 is printed
print(id(mixed_list2))
# Here address of List1 is printed
print(id(mixed_list3))
# Here, values are checked of both operands
```

```
if mixed_list1 != mixed_list2:
   print("Both mixed-list1 and mixed_list2 have not equal value")
else:
   print("Both mixed-list1 and mixed_list2 have equal value")

if mixed_list1 != mixed_list3:
   print("Both mixed-list1 and mixed_list3 have not equal value")
else:
   print("Both mixed-list1 and mixed_list3 have equal value")

if mixed_list2 != mixed_list3:
   print("Both mixed-list2 and mixed_list3 have not equal value")
else:
   print("Both mixed-list2 and mixed_list3 have equal value")

# Here checking of operands point to the same object or not
if mixed_list1 is not mixed_list2:
   print("Both mixed-list1 and mixed_list2 do not point to the same object")
else:
   print("Both mixed-list1 and mixed_list2 point to the same object ")

if mixed_list1 is not mixed_list3:
   print("Both mixed-list1 and mixed_list3 do not point to the same object")
else:
   print("Both mixed-list1 and mixed_list3 point to the same object ")

if mixed_list2 is not mixed_list3:
   print("Both mixed-list2 and mixed_list3 do not point to the same object")
else:
print("Both mixed-list2 and mixed_list3 point to the same object ")
```

#Output

```
[34, 45, 89, 123]
[34, 89, 123]
[34, 45, 89, 123]
89101000
89101512
89101000
Both mixed-list1 and mixed_list2 have not equal value
Both mixed-list1 and mixed_list3 have equal value
Both mixed-list2 and mixed_list3 have not equal value
Both mixed-list1 and mixed_list2 do not point to the same object
Both mixed-list1 and mixed_list3 point to the same object
Both mixed-list2 and mixed_list3 do not point to the same object
```

i) in (membership operator)

This operator checks whether the given value is present in the sequence or not. It evaluates to True if it is present else False.

Examples:

i) var_int1=459

```
mixed_list1 = [23,459,478,65]
if var_int1 in mixed_list1:
   print("Variable value is present in the given list")
else:
   print("Variable value is absent in the given list")
```

Output: Variable value is present in the given list

ii) var_int1=455

```
mixed_list1 = [23,459,478,65]
if var_int1 in mixed_list1:
   print("Variable value is present in the given list")
else:
   print("Variable value is absent in the given list")
```

Output: Variable value is absent in the given list

j) not in(membership operator)

This operator checks whether the given value is not present in the sequence or not. It evaluates to True if it is not present else False.

Examples:

i) var_int1=459

```
mixed_list1 = [23,459,478,65]
if var_int1 not in mixed_list1:
   print("Variable value is not present in the given list")
else:
   print("Variable value is present in the given list")
```

Output: Variable value is present in the given list

ii) var_int1=455

```
mixed_list1 = [23,459,478,65]
if var_int1 not in mixed_list1:
   print("Variable value is not present in the given list")
else:
   print("Variable value is present in the given list")
```

Output: Variable value is not present in the given list

2.5.1 Chaining of Comparison Operators

All the above-mentioned comparison operators gives the output in the form of True or False. While doing chaining using comparison operators, it is done arbitrarily [2].
Chaining can be better understood by the following example:
int_var1 > int_var2
int_var2 > int_var3

For above, chaining can be done like
int_var1 > int_var2 > int_var3

Syntax:
Operand1 operator1 Operand2 operator2 Operand3 operator3 Operand4 ….

Always process chaining from left to right as precedence of comparison operators is same.

Example 11: Program 1 of Chaining to understand the concept

```
int_var1 = 189
int_var2 = 174
int_var3 = 67

# Firstly check int_var1 >int_var2
print(int_var1 > int_var2)

# Now, Check int_var2 > int_var3
print(int_var2 > int_var3)

# Do chaining and check the output
print(int_var1 > int_var2 > int_var3)

# Above chaining is equivalent to
if (int_var1 > int_var2 and int_var2 > int_var3):
   print(True)
else:
   print(False)
```

#Output

```
True
True
True
True
```

Example 12: Program 2 for deeper understanding of chaining operator

```
int_var1 = 189
int_var2 = 474
int_var3 = 267

# Firstly check int_var1 >int_var2
print(int_var1 < int_var2)

# Now, Check int_var2 > int_var3
print(int_var2 > int_var3)

# Do chaining and check the output
print(int_var1 < int_var2 > int_var3)  # Perfectly Legal

# Above chaining is equivalent to
if (int_var1 < int_var2 and int_var2 > int_var3):
    print(True)
else:
    print(False)
```

#Output

```
True
True
True
True
```

Example 13: Program 3 of Chaining

```
int_var4=34
print(27<int_var4<35)
print(37<int_var4<45)

print(27<int_var4>14)
print(27<int_var4>67)

print(34==int_var4>17)

print(34==int_var4>67)
```

#Output

True
False
True
False
True
False

Example 14: Chaining operators complex programs

```
int_var1=28
int_var2=28
int_var3=37

# Here checking int_var1 and int_var2 has same address and int_var3 is having different address
print(int_var1 is int_var2 is not int_var3)

# Addresses can be checked
print(id(int_var1))
print(id(int_var2))
print(id(int_var3))

print(int_var1 is not int_var2 is int_var3)

int_var4=12
print(int_var1 is int_var2 is not int_var3 > int_var4)
# Above left to right will work
int_var5=9
print(int_var1 is int_var2 is not int_var3 > int_var4 >int_var5)

int_var6=89
print(int_var1 is int_var2 is not int_var3 > int_var4 >int_var5< int_var6)

int_var7=76
print(int_var1 is int_var2 is not int_var3 > int_var4 >int_var5< int_var6 >int_var7)
```

#Output

```
True
8790643746032
8790643746032
8790643746320
```

False
True
True
True
True

Example 15: Another program for chaining

```
# Variables with initial values
int_var1=28
int_var2=28
int_var3=37

print(int_var1==int_var2)

int_var4=0
print(int_var4 < int_var1==int_var2)

int_var5=0
print(int_var4 < int_var1==int_var2 <int_var5)

int_var6=45
print(int_var4 < int_var1==int_var2 >int_var6)
```

#Output

True
True
False
False

Example 16: Chaining with usage of ==,!=,is, is not with lists

```
#List1 is initialised with four values
mixed_list1 = [34,45,89,123]

# List2 is initialised with three values
mixed_list2 = [34,89,123]

# List 3 is assigned values of List1 (Note: Both will have same address)
mixed_list3 = mixed_list1
```

```
#Printed the elements of List1
print(mixed_list1)
#Printed the elements of List2
print(mixed_list2)
#Printed the elements of List3
print(mixed_list3)

# Here address of List1 is printed
print(id(mixed_list1))
# Here address of List1 is printed
print(id(mixed_list2))
# Here address of List1 is printed
print(id(mixed_list3))

# First if-else
if mixed_list1 != mixed_list2 is not mixed_list3:
   print("The first if condition is True")
else:
   print("The first else part is worked")

#Second if-else
if mixed_list1 != mixed_list3 is not mixed_list2:
   print("The second if condition is True")
else:
   print("The second else part is worked")

# Third if-else
if mixed_list2 != mixed_list3 is not mixed_list1:
   print("The third if condition is True")
else:
   print("The third else part is worked")

# Fourth if-else
if mixed_list1 != mixed_list2 is mixed_list3:
   print("The fourth if condition is True")
else:
   print("The fourth else part is worked")

# Fifth if-else
```

```
if mixed_list1 != mixed_list3 is mixed_list2:
   print("The fifth if condition is True")
else:
   print("The fifth else part is worked")

# sixth if-else
if mixed_list2 != mixed_list3 is mixed_list1:
   print("The sixth if condition is True")
else:
   print("The sixth else part is worked")

# seventh if-else
if mixed_list1 is not mixed_list2 == mixed_list3:
   print("The seventh if condition is True")
else:
   print("The seventh else part is worked")

# eight if-else
if mixed_list1 is not mixed_list3 == mixed_lis2:
   print("The eight if condition is True")
else:
   print("The eight else part is worked")

# ninth if-else
if mixed_list2 is not mixed_list3==mixed_list1:
   print("The ninth if condition is True")
else:
   print("The ninth else part is worked")

# tenth if-else
if mixed_list1 is mixed_list2 == mixed_list3:
   print("The tenth if condition is True")
else:
   print("The tenth else part is worked")

# eleventh if-else
if mixed_list1 is mixed_list3 == mixed_list2:
   print("The eleventh if condition is True")
else:
   print("The eleventh else part is worked")
```

```
# twelveth  if-else
if mixed_list2 is mixed_list3==mixed_list1:
   print("The twelfth if condition is True")
else:
   print("The twelfth else part is worked")
```
#Output
```
[34, 45, 89, 123]
[34, 89, 123]
[34, 45, 89, 123]
90139784
90139720
90139784
The first if condition is True
The second else part is worked
The third else part is worked
The fourth else part is worked
The fifth else part is worked
The sixth if condition is True
The seventh else part is worked
The eight else part is worked
The ninth if condition is True
The tenth else part is worked
The eleventh else part is worked
The twelfth else part is worked
```

2.6 USAGE OF BREAK, CONTINUE AND PASS STATEMENTS

2.6.1 Break Statement

The keyword break is used to interrupt the running loop. Basically, this is intentional. When break statement is encountered, the flow goes to immediate step after the loop [3].

Syntax:

```
break
```

Example 17: Usage of break with while loop

```
int_var1 = 5
while int_var1<15:
  if int_var1 == 9:
  break
  print(int_var1)
  int_var1 = int_var1+1
print("Outside loop now")
```

```
#Output
5
6
7
8
Outside loop now
```

Example 18: Usage of break with for loop

```
for int_var1 in range(5,15):
  if int_var1!=9:
    print(int_var1)
  else:
    break
print("Outside loop now")

#Output
5
6
7
8
Outside loop now
```

Example 19: Search an element in the list

```
mixed_list1 = [12, 3, 24, 45, 89, 108, 23]
item_find = int(input("Enter the item you want to find out from the list"))
num_elements = len(mixed_list1)
flag=0
for i in range(0, num_elements):
        if (mixed_list1[i] == item_find):
            flag=1
            break
if flag==0:
    print("Element is not present in array")
else:
    print("Element is present in array")
```

Test Case 1:

```
#Input
Enter the item you want to find out from the list88
#Output
```

Element is not present in array

Test Case 2:
#Input
Enter the item you want to find out from the list12
#Output
Element is present in array
Test Case 3:
#Input
Enter the item you want to find out from the list108
#Output
Element is present in array

2.6.2 Continue Statement

The keyword continue is used to end the current iteration of the loop and iterate the next iteration of the loop [4].

Syntax:
continue

Example 20: Usage of continue with while loop

```
int_var1 = 5
while int_var1<15:
   int_var1 = int_var1+1;
   if int_var1 == 9:
      continue
   print(int_var1)

print("Outside loop now")
```

#Output
6
7
8
10
11
12
13
14
15
Outside loop now

Example 21: Usage of continue with for loop

```
for int_var1 in range(5,15):
  if int_var1!=9:
    continue
  else:
    print(int_var1)
print("Outside loop now")
```

#Output

9

Outside loop now

2.6.3 Pass Statement

pass statement is simply does not do anything. It can be used in loops, functions, classes etc. It is different from comments. As interpreter ignore comments whereas interpreter does not ignore pass statement.

Syntax

```
pass
```

Example 22: Program to show Without pass statement, an error will occur

```
for int_var1 in [12,23,89,56,235]:
```

#Output

SyntaxError: unexpected EOF while parsing

Example 23: Program with relation to example 22 but with using pass

```
for int_var1 in [12,23,89,56,235]:
   pass
```

After executing the above statements, No error will be generated

Example 24: pass with function and loop

```
str_var1 = "JagatOpenUniversity"

for new_var1 in str_var1:
   if new_var1 =='O' or new_var1=='U':
      print("Pass statement is exceuted when value is O and U")
      pass
```

```
    print(new_var1)


def fun_pass():
    pass

print("Now fun_pass() is called")
fun_pass()
print("After fun_pass() calling, nothing is printed as pass statement is in fun_pass()")
```

#Output

```
J
a
g
a
t
Pass statement is exceuted when value is O and U
O
p
e
n
Pass statement is exceuted when value is O and U
U
n
i
v
e
r
s
i
t
y
Now fun_pass() is called
After fun_pass() calling, nothing is printed as pass statement is in fun_pass()
```

2.7 SELF-CHECKED QUESTIONS

A) What will be the output of the following code:

```
for int_var1 in range(3):
    for int_var2 in range(3):
        if int_var2==2:
```

```
        break
    print(int_var1,end=" ")
```

a) 0 0 1 1 2 2 b) 0 1 0 1 0 1 c) 0 1 2 0 1 2 d) 0 1 0 2 0 3

B) Select the best option for the following code:

```
for int_var1 in range(3):
    for int_var2 in range(3):
        if int_var2==2:
            break
        print(int_var2,end=" ")
```

a) 0 0 1 1 2 2 b) 0 1 0 1 0 1 c) 0 1 2 0 1 2 d) 0 1 0 2 0 3

C) Write the output of the following code

```
for int_var1 in range(6):
    for int_var2 in range(6):
        if int_var2==2:
            continue
        print(int_var2,end=" ")
```

D) Write the output of the following code

```
for int_var1 in range(6):
    for int_var2 in range(6):
        if int_var2==2:
            continue
        print(int_var1,end=" ")
```

E) Mention the output of the code

```
for int_var1 in range(3):
    for int_var2 in range(3):
        if int_var2==2:
            pass
        print(int_var2,end=" ")
```

2.8 SUMMARY

After reading this unit, students will be able to differentiate between all logical operators and they will know where they have to use which operator. When students are working on break and continue statements. There is a lot of confusion, how these statements will work. Appropriate programming examples are given to provide deeper understanding of these concepts. The unit gives programming examples related with logical operators, chaining comparison operators, pass statement. Overall, this unit helps in building complex programs. Students can differentiate pass

statement and comments. This unit targets to build logical concepts so that they can do MCQ, predict the output of the code along with programs construction.

2.9 PRACTICE QUESTIONS

A) Which of the following will not be printed when the Python3 cod3 is run

```
for str_var1 in 'JagatOpenUniversity':
    if str_var1 == 'O' or str_var1=='U':
        continue
    print('The string alphabet is :' + str_var1)
```

a) The string alphabet is :a
b) The string alphabet is :U
c) The string alphabet is :e
d) The string alphabet is :n

B) How many times '?' will be printed when the following code is executed on Python3 platform.

```
for int_var1 in [5, 8, 10]:
    for int_var2 in [5,6,7,8,9,10]:
        if int_var1!=int_var2:
            continue
        print('?')
```

a) 1
b) 3
c) 4
d) 6

C) How many times '?' will be printed when the following code is executed on Python3 platform.

```
for int_var1 in [5, 8, 10]:
    for int_var2 in [5,6,7,8,9,10]:
        if int_var1!=int_var2:
            break
        print('?')
```

a) 1
b) 3
c) 4
d) 6

D) Mention the output of the following code

```
int_var1=12
int_var2 = 37
int_var3 =23

print(int_var2> int_var1==int_var3)
```

E) Mention the output

```
int_var1=12
int_var2 = 37
int_var3 =12

print(int_var2> int_var1==int_var3 < int_var2)
```

F) Predict the output

```
int_var1 = 1
while True:
    if int_var1%9 == 0:
        break
    print("The output when this statement is executed",int_var1)
    int_var1=int_var1+4
print("At the end, you are out of the loop and this line is printed")
```

G) Predict the output

```
int_var1 = 1
while int_var1<19:
    if int_var1%9 == 0:
        int_var1=int_var1+1
        continue
    print("The output when this statement is executed",int_var1)
    int_var1=int_var1+4
print("At the end, you are out of the loop and this line is printed")
```

H) Choose the correct option after running the following code:

```
int_var1 = 1
while int_var1<19:
    if int_var1%2 == 0:
        int_var1=int_var1+1
        continue
    if int_var1%9 == 0:
```

```
        int_var1=int_var1+1
        pass
    print(int_var1,end=" ")
    int_var1=int_var1+4
```

a) 1 5 10 15 b) 1 1 3 3 5 5 7 7 9 9 11 11 13 13 15 15 17 17
c) 1 5 9 13 17 d) 1 3 5 7 9 11 13 15 17

I) Select the best option after running the following code

```
int_var1 = 1
while int_var1<19:
    if int_var1%2 == 0:
        int_var1=int_var1+1
        continue
    if int_var1%11 == 0:
        int_var1=int_var1+1
        break
    print(int_var1,end=" ")
    int_var1=int_var1+4
```

a) 1 5 9 13 17
b) 1 3 5 7 9 11 13 15 17
c) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
d) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18

J) What will be the correct output of the following code

```
bool_var1 = False
while True:
    print(True)
    break
```

K) What will be the correct output of the following code

```
bool_var1 = True
while True:
    print(True)
    break
```

L) Differentiate pass and comments statement in python programming
M) Construct a program to find out whether 179 is a prime number or not.
N) With the help of logical operators, find out the largest of four numbers.
O) List out the differences between break and continue.

P) Mention the precedence level of logical operators and explain with a suitable example which shows precedence level usage.

Q) List the usage of ‘is operator’ and ‘is not operator’ with suitable example.

REFERENCES

[1] https://www.geeksforgeeks.org/difference-operator-python/

[2] https://www.tutorialspoint.com/chaining-comparison-operators-in-python

[3] https://www.programiz.com/python-programming/break-continue

[4] https://www.w3schools.com/python/ref_keyword_continue.asp

PYTHON PROGRAMMING

UNIT III: PROGRAM FLOW CONTROL

STRUCTURE

3.0 Objectives

3.1Introduction

3.2 Flow Control

- **3.2.1 Conditional Statements**
  - **3.2.1.1 Simple if statement**
  - **3.2.1.2 if-else statement**
  - **3.2.1.3 if elif else statement Or Chained Conditional statements**
  - **3.2.1.4 Nested Conditions**
- **3.2.2 Shorthand Notations**
  - **3.2.2.1 Shorthand Notations for if**
  - **3.2.2.2 Shorthand Notations for if else**
  - **3.2.2.3 Shorthand Notations for 3 if elif else**

3.3 Loops

- **3.3.1 While Loop**
  - **3.3.1.1 Use of else with while loop**
- **3.3.2 for loop**
  - **3.3.2.1 Iterating over Sequences using Index**
  - **3.3.2.2 Use of else with for Loop**
- **3.3.3 Nested Loop**

3.4 Self-Check Questions

3.5 Summary

3.6 Practice Questions

3.0 OBJECTIVES

- Learn in detail about conditional statements followed by syntax and coding programs.
- Shorthand notations learning with syntax and examples
- Familiarize with different types of loops.
- Detailed learning about nested loops along with coding examples.

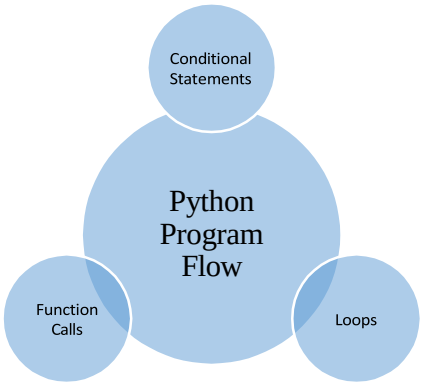
3.1 INTRODUCTION

This module targets to give deep insights into conditional statements like if, if-else, if-elif-else, nested if-else. Syntax along with examples are also mentioned. Program examples are like leap year, even-odd number, whole number or not, greatest among three numbers and four numbers, etc. Conditional statements help in knowing the execution order of code statements. The statements like if, if-else, etc. help in making the decision and repeat certain blocks to do a particular task. For repeating the tasks, loops are used extensively. For and while loops are used to iteratively repeat tasks. Syntax and examples related to loops are mentioned in detail. How else part works with for and while loop is explained. Followed by loops working on lists, iterable objects, etc. This module helps the users in getting knowledge about different conditional statements and loops. Functions calls will be explained in the next module.

3.2 FLOW CONTROL

Any program written in any programming language has a certain flow and the flow tells the order in which the programming language code flows. Generally, the programming language code control flow is controlled by the following:

1. Conditional Statements
2. Loops
3. Function Calls



3.1 Python Control Flow

3.2.1 Conditional Statements

Conditional statements are also known as conditional constructs or conditional expressions. They perform different actions depending upon the evaluation of the conditions either True or False. The conditions use different operators like arithmetic, relational, comparison, etc. The conditional statements consist of the following types:

1. Simple if statement
2. if-else statement
3. if elif else statement or Chained Conditional Statements
4. Nested Conditions

3.2.1.1 Simple if Statement

Simple if statement is one of the commonly used conditional statements in mostly all programming languages for decision making. The keyword used for this conditional statement is "*if*". The "*if"* keyword is followed with a condition and this condition is „True" then a certain indented block of code is executed that is inside "if" block otherwise not. The syntax is mentioned below:

Syntax:

```
If (condition):
   block of statement(s)
Rest of the statements
```

In the above syntax, *if condition is True then both block of statement(s) and rest of the statements will be executed and if condition is False then only rest of the statements will be executed.*

Example 1: # Check a number is even

```
number1=16                  # Variable with initialization
if(number1%2==0):           # if condition
   print("Even Number")
print("This statement of the code is always printed")     #Rest of the statements
#Output
Even Number
This statement of the code is always printed
```

Example 2: # Check a number is even

```
number1=15                  # Variable with initialization
if(number1%2==0):           # if condition
   print("Even Number")
print("This statement of the code is always printed")     #Rest of the statements
#Output
This statement of the code is always printed
```

Example 3: # Check chances of COVID

```
have_fever = True               # Variable with initialization
if(have_fever):                 # if condition
   print("Chances of COVID")
print("This statement of the code is always printed")     #Rest of the statements
#Output
Chances of COVID
```

This statement of the code is always printed

Example 4: Check Chances of COVID Variant

```
have_fever = False                    # Variable with initialization
if(have_fever):                       # if condition
   print("Chances of COVID")
print("This statement of the code is always printed")     #Rest of the statements
#Output
This statement of the code is always printed
```

3.2.1.2 if-else Statement

This conditional statement uses the keyword *if-else*. *if* condition is True then block of if statement(s) is executed and *if* condition is False then else part i.e. a block of else statement(s) is executed. The syntax is mentioned below:

Syntax:

```
if (condition():
   block of if statement(s)
else:
   block of else statement(s)
```

Example 5: # Check Immunization Improves for COVID or Not

```
have_injection1 = True
have_injection2 = True
if(have_injection1 == True and have_injection2==True):
   print("Immunization improves for COVID")
else:
   print("Higher chances of prone to COVID")
#Output
Immunization improves for COVID
```

Example 6: Check Number is even or odd

```
number1=15
if(number1%2==0):
   print("Number is Even")
else:
   print("Number is Odd")
#Output
Number is Odd
```

Example 7: Check whether a number is a whole number or not

```
x = 0.87
if (x - int(x) == 0):
```

```
    print("Whole Number")
else:
    print("Has decimals")
#Output
Has decimals
```

Example 8: Find the greater element between two integer elements (Assumption is that both elements cannot be equal)

```
number1=int(input())
number2=int(input())
if number1>number2:
    print("Number1 is greater")
else:
    print("Number2 is greater")
```

Test case 1:

#Input Elements
122
213
#Output
Number2 is greater

Test case 2:

#Input Elements
422
113
#Output
Number1 is greater

3.2.1.3 if elif else Statement Or Chained Conditional Statements

This statement is used to control multiple conditions in the program. The *if* condition is False then *elif* is used to control multiple conditions.

Syntax:

```
if(conidtion1):
    block of statement(s) of if
elif(condition2):
    block of statement(s) of elif
elif(condition3):
    block of statement(s) of elif
.................
.................
.................
else:
    block of statement(s) of else
```

#NOTE: *else* part is optional, the program will work if you do not use *else* part in the code too.

Example 9: # Compare two integer numbers

```
number1=int(input())
number2=int(input())
```

```
if number1>number2:
   print("Number1 is greater")
elif number1<number2:
   print("Number 2 is greater")
else:
   print("Both numbers are equal")
```

Test case 1:

```
#Inputs
122
213
#Output
Number2 is greater
```

Test case 2:

```
#Inputs
422
113
#Output
Number1 is greater
```

Test case 3:

```
#Inputs
333
333
#Output
Both numbers are equal
```

Example 10: Check whether a positive number is two digit, three digit, four digit or greater that four digit number and print an appropriate message.

```
user_number=int(input())#input from the user
if user_number>=0 and user_number <=9:
   print("Single Digit Number")
elif user_number >=10 and user_number <=99:
   print("Two Digit Number")
elif user_number >=100 and user_number <=999:
   print("Three Digit Number")
elif user_number >=1000 and user_number <=9999:
   print("Four Digit Number")
else:
   print("Number is greater than Four Digit Number")
```

Test Case 1:

```
#Input
10
```

#Output
Two Digit Number
Test Case 2:
#Input
9989
#Output
Four Digit Number
Test Case 3:
#Input
6748309
#Output
Number is greater than Four Digit Number

Example 11: Check whether a year is a leap year or not
NOTES for Basic Understanding: Leap year is a year if it is divided by 400. If it is not divided by 400 then check is it divided by 100, if it is divided by 100 then it is not a leap year. If a number is neither divisible by 100 nor 400 then only check whether a given number is divided by 4, if it is divided by 4 then it is a leap year unless it is not a leap year.

```
normal_year  =  int(input())
if (normal_year%400 == 0):
        print("Leap Year")
elif (normal_year%100 == 0):
        print("Not a Leap Year")
elif (normal_year%4 == 0):
        print("Leap Year")
else:
        print("Not a Leap Year")
```

Test Case 1:
#Input
2000
#Output
Leap Year
Test Case 2:
#Input
1900
#Output
Not a Leap Year

Test Case 3:
#Input
2020
#Output
Leap Year
Test Case 4:
#Input
2021
#Output
Not a Leap Year

3.2.1.4 Nested Conditions

Nested conditions consist of nested if as well as nested if-else. In the nested if, we have multiple if in outer if block. In the nested if-else block, we have if-else statements inside other *if* or *else* or both blocks. The important thing is here to take care of indentation and it is the way to know the level of nesting.

Syntax of nested if:

```
if condition1():
   if condition2():
      if condition3():
         …….
         …….
         block3 of statement(s)
      block 2 of statement(s)
   block 1 of statement(s)
else:
   block of statement(s)
```

Syntax of nested if-else:

```
if condition1():
   if condition2():          #optional if-else
      block of if statement(s)
   else:
      block of else statement(s)
else:
   if condition3():          #optional if-else
      block of if statement(s)
   else:
      block of else statement(s)
```

Example 12: Find the greatest of three integer numbers (Assume that all three numbers are distinct)

```
number1=int(input())
number2=int(input())
number3=int(input())
if number1>number2:
   if number1>number3:
      print("number1 is greatest")
   else:
      print("number3 is greatest")
else:
   if number2>number3:
```

```
        print("number2 is greatest")
    else:
        print("number3 is greatest")
```

Test case1:

```
#Inputs
23
36
47
#Output
number3 is greatest
```

Test case2:

```
#Inputs
67
50
56
#Output
number1 is greatest
```

Test case3:

```
#Inputs
145
567
444
#Output
number2 is greatest
```

Example 13: Check whether a year is a leap year or not

```
normal_year = int(input())
if normal_year%4==0:
    if normal_year%100==0:
        if normal_year%400==0:
            print("Leap Year")
        else:
            print("Not a Leap Year")
    else:
        print("Leap Year")
else:
    print("Not a Leap Year")
```

Test Case 1:

```
#Input
2000
#Output
Leap Year
```

Test Case 2:

#Input
1900
#Output
Not a Leap Year
Test Case 3:
#Input
2020
#Output
Leap Year
Test Case 4:
#Input
2021
#Output
Not a Leap Year

Example 14: Find the greatest of four integer numbers (Assume that all four numbers are distinct)

```
number1=int(input())
number2=int(input())
number3=int(input())
number4=int(input())
if number1>number2:
   if number1>number3:
      if number1>number4:
         print("number1 is greatest")
      else:
         print("numbe4 is greatest")
   else:
      if number3> number4:
         print("number3 is greatest")
      else:
         print("number4 is greatest")
else:
   if number2>number3:
      if number2>number4:
         print("number2 is greatest")
      else:
         print("numbe4 is greatest")
   else:
      if number3> number4:
         print("number3 is greatest")
      else:
         print("number4 is greatest")
```

Test Case1:
$inputs
11
21
25
45
#output
number4 is greatest
Test Case2:
$inputs

23
45
34
12
#output
number2 is greatest
Test Case3:
$inputs
104
210
250
145
#output
number3 is greatest
Test Case4:
$inputs
678
246
298
484
#output
number1 is greatest

3.2.2 Shorthand Notations:

3.2.2.1 Shorthand Notation for if

Syntax:

if (condition): if-statement

Example 15: Check a number is Even Number

```
number1=16
if number1%2==0: print("Even Number")
#Output
Even Number
```

3.3.2.2 Shorthand Notation for if else

Syntax:

if-statement(s) if (condition) else else-statement(s)

Example 16: Check whether a number is even or odd

```
number1=int(input())
print("Number is Even") if number1%2==0 else print("Number is Odd")
```

Test Case 1

#input

15

#output

Number is Odd

Test Case 2

#input

28

#Output

Number is Even

Example 17: Check whether a number is a whole number or not

```
x = 0.87
```

```
print("Whole Number") if(x - int(x) == 0) else print("Has decimals")

#Output
Has decimals
```

Example 18: Find greater of two elements

```
number1=int(input())
number2=int(input())
print("number1 is greater") if(number1>number2) else print("number2 is greater")
```

Test Case1:

```
#Inputs
25
67
#Output
number2 is greater
```

Test Case2:

```
#Inputs
475
143
#Output
number1 is greater
```

3.2.2.3 Shorthand Notation for if elif else

Example 19: Compare two integer numbers

```
number1=int(input())
number2=int(input())
print("Number1 is greater") if number1>number2 else print("Both numbers are equal") if
number1==number2 else print("Number2 is greater")
```

Test case 1:

```
#Inputs
122
213
#Output
Number2 is greater
```

Test case 2:

```
#Inputs
422
113
#Output
Number1 is greater
```

Test case 3:

```
#Inputs
```

333
333
#Output
Both numbers are equal

3.3 LOOPS

Loops execute a specific block of code that contains many statements and this block is executing repetitively. Two types of loops are there in python:

a) while loop
b) for loop

In loops, three things to be taken care and these are mentioned below:

a) Initialization
b) Condition
c) Increment/Decrement

In initialization, variables are initialized to certain values. In the condition part, conditions are specified that tells how many times the loop will be executed repetitively. The third part is increment or decrement of the values of variables that are used in the condition part so that loop will be stopped after performing a particular task.

3.3.1 While Loop

This loop iterate over a block of code repetitively or repeatedly until the condition is satisfied or True and when it becomes dissatisfied or False, the program jumps after the immediate line of the loop. The important thing is how to group many statements in a block. This is done by indentation.

Syntax:

```
initialization
while(condition):
        statement 1
        statement 2  #optional
        …….
        …….
        …….
        statement n #optional
        increment/decrement  # Order of statements and increment/decrement is not specific
```

Example 20: Print first 4 natural numbers

```
var=1          # var is a counter that will execute from 1 to 4
num=4          # Till this loop should go
while var<=num:
   print(var,end=" ")  # end=" " is used to print each element with a space
```

```
    var=var+1
#Output
1 2 3 4
```

Explanation:
Step 1. var is initialized with value 1.

Step 2. num is initialized with value 4 as the condition will go till this value.

Step 3. Condition is checked (1 <= 4) as var=1 and num=4 and it is true.

3a) 1 is printed 1st time with space.

3b) Value of var is updated to 2.

Step 4. Condition is checked (2 <= 4) as var=2 and num=4 and it is true.

4a) 2 is printed 2nd time with space.

4b) Value of var is updated to 3.

Step 5. Condition is checked (3 <= 4) as var=3 and num=4 and it is true.

4a) 3 is printed 3rd time with space.

4b) Value of var is updated to 4.

Step 6. Condition is checked (4 <= 4) as var=4 and num=4 and it is true.

4a) 4 is printed 4th time with space.

4b) Value of var is updated to 5.

Step 7. Condition is checked (5 <= 4) as var=5 and num=4 and it is false.

Step 8. The flow goes outside the while loop

Example 21: Make a program to find sum of first 15 natural numbers.

```
sum_numbers = 0 #initialise variable sum_numbers=0, it will store sum of first 15 numbers
num=15              # Till this loop should go
var=1               # var is a counter that will execute from 1 to 15
while var<=num:
    sum_numbers=sum_numbers+var #everytime loop runs var value is added to sum_numbers
    var=var+1
print("Sum is ",sum_numbers)
#output
Sum is 120
```

Example 21: Print sum of numbers from 10 to -5 using while loop.

```
var=10          # var is a counter that will execute from -5 to 10
num=-5          # Till this loop should go
sum_numbers = 0 #initialise variable sum_numbers=0
while var>=num:
   sum_numbers=sum_numbers+var #everytime loop runs var value is added to sum_numbers
   var=var-1 # decrement the var
print("Sum is ",sum_numbers)
#Output
Sum is 40
```

3.3.1.1 Use of else with While Loop

As discussed, the loop iterates over a block of code repetitively or repeatedly until the condition is satisfied or True and when it becomes dissatisfied or False, the program jumps after the immediate line of the loop. The else part is executed when the condition becomes False or dissatisfied. Only break and exceptions can hold the execution of else part.

Syntax:

```
initialization
while(condition):
        statement 1
        statement 2  #optional
        …….
        …….
        …….
        statement n #optional
        increment/decrement  # Order of statements and increment/decrement is not specific
else:                   #optional
   statement(s)
```

Example 22: Make a program to find sum of first 15 natural numbers by using else with while loop.

```
sum_numbers = 0 #initialise variable sum_numbers=0, it will store sum of first 15 numbers
num=15              # Till this loop should go
var=1               # var is a counter that will execute from 1 to 15
while var<=num:
   sum_numbers=sum_numbers+var #everytime loop runs var value is added to sum_numbers
   var=var+1
else:
   print("Sum is ",sum_numbers)
#output
Sum is 120
```

3.3.2 For Loop

In python, „for loop‟ is used to iterate over sequences like tuple, list, string, etc. and iterable objects. It will iterate till the last value of the sequence

Syntax:

```
for variable in sequence:                    # Sequence can be list, tuple, string
                    Body of for loop
```

Example 23: # Program to find the sum of first 15 natural numbers

```
sum_numbers = 0 #initialise variable sum_numbers=0, it will store sum of first 15 numbers
num=range(1,16)
# iterate over the list
for var in num:  # take one value from 1 till 15 after every iteration
   sum_numbers = sum_numbers+var
print("Sum is", sum_numbers)
```

#Output is

Sum is 120

Explanation:

First of all, understand the inbuilt range() function. range function syntax is range(start, stop, step_size). By default, step_size is 1.

Here range(1,16) or range(1,16,1) are the same and this will generate numbers from 1 to 15, remember 16 will not be generated. It will go to 15.

Example 24: # Program to find the sum of the first 15 natural numbers stored in a list

```
num = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15] # List
sum_numbers = 0 #initialise variable sum_numbers=0, it will store sum of first 15 numbers
# iterate over the list
for var in num:
   sum_numbers = sum_numbers+var
print("Sum is", sum_numbers)
```

#Output

Sum is 120

Example 25: Program to iterate over sequences

```
# Iterating over a list
print("Iteration over List Sequence")
grad_l = ["CSE","ME","Civil","ECE"]
for var in grad_l:
   print(var,end=" ")
# Iterating over a tuple
print("\nIteration over Tuple Sequence")
```

```
grad_t = ["CSE","ME","Civil","ECE"]
for var in grad_t:
    print(var,end=" ")
 # Iterating over a String
print("\nIteration over String")
grad_s = ["Btech"]
for var in grad_s:
    print(var,end=" ")
# Iterating over Dictionary
print("\nIteration over Dictionary")
grad_d = dict()
grad_d['CSE']=1
grad_d['Civil'] =2
grad_d['ECE']=3
grad_d['ME']=4
for var in grad_d :
    print("%s  %d" %(var, grad_d[var]))
#OUTPUT
Iteration over List Sequence
CSE ME Civil ECE
Iteration over Tuple Sequence
CSE ME Civil ECE
Iteration over String
Btech
Iteration over Dictionary
CSE  1
Civil  2
ECE  3
ME  4
```

3.3.2.1 Iterating over Sequences using Index

In the sequence, the index of the elements is used for iteration. The main idea here is to first calculate the length of the sequence and then use the range function that helps in iteration till the length of the sequence.

Example 26: Iteration over a sequence using index using range and len function

```
grad_l = ["CSE","ME","Civil","ECE"]
for index in range(len(grad_l)):
    print(index, grad_l[index])
#OUTPUT
0 CSE
```

```
1 ME
2 Civil
3 ECE
```

Example 27: Iteration over a sequence using enumerate function

```
grad_l = ["CSE","ME","Civil","ECE"]
for index,value in enumerate(grad_l):
   print(index,value)
#Output
0 CSE
1 ME
2 Civil
3 ECE
```

Example 28: Iteration over a sequence using zip function

```
grad_l = ["CSE","ME","Civil","ECE"]
for index in zip(range(len(grad_l)),grad_l):
        print(index)
#Output
(0, 'CSE')
(1, 'ME')
(2, 'Civil')
(3, 'ECE')
```

3.3.2.2 Use of else with for Loop

As discussed, for loop iterate till the last element of the sequence and sequence can be list, string, tuple, and other iterable objects. The else part is executed when the block finishes the execution.

Syntax:

```
for variable in sequence:                    # Sequence can be list, tuple, string
                    Body of for loop
else:
   Body of else block
```

Example 29: Print 0 to 6 numbers

```
num=int(input())
for var in range(num): # take one value from 0 till 6 after every iteration, as range goes till num1
   print(var,end=" ")
else:
   print(“Inside else block”)
#Input
7
#Output
```

0 1 2 3 4 5 6
Inside else block

3.3.3 Nested Loops

In python, one loop can be used inside other loops. *while* loop can be used within other *while* loops, *for* loop can be used within other *for* loop, *for* loop can be used within *while* loop and vice-versa.

Syntax for nested *while* loop

```
while(condition1): # Outer Loop
        while(condition2): # Inner Loop
                        statement(s) # Inside Inner Loop
        statement(s) #
```

Syntax for nested *for* loop:

```
for var1 in sequence: # Outer loop
        for var2 in sequence: # Inner Loop
                        statement(s)  # Inside Inner Loops
        statement(s) #
```

NOTE: Many levels of nesting can be done for loops

Example 30 Print the following sequence using while loop

```
*
*       *
*       *       *
*       *       *       *
```

Note: There is tab space between elements in a line.

Explanation: First look outer loop is running, how many times. The vertically here shows the outer loop execution. It shows four times. Now look at inner loop, try to understand the following:

when outer loop runs 1st time, inner loop runs 1 time (horizontal values), print *

when outer loop runs 2nd time, inner loop runs 2 times (horizontal values), print * two times with tab space

when outer loop runs 3rd time, inner loop runs 3 times (horizontal values), print * three times with tab space

when outer loop runs 4th time, inner loop runs 4 times (horizontal values), print * four times with tab space

Solution:

```
outer_var=1
while(outer_var<=4):
   inner_var=1
```

```
    while(inner_var<=outer_var):
        print('*',end="\t")#end="\t" for tab space between elements in a line
        inner_var=inner_var+1
    print("\n") #Next line
    outer_var=outer_var+1
```

Example 31: Print the following sequence using while loop

```
1
1       2
1       2       3
1       2       3       4
```

Note: There is tab space between elements in a line.

Explanation: First look outer loop is running, how many times. The vertically here shows the outer loop execution. It shows four times. Now, look at the inner loop, try to understand the following:

when outer loop runs 1st time, inner loop runs 1 time (horizontal values), print 1

when outer loop runs 2nd time, inner loop runs 2 times (horizontal values), print 1,2 with tab space

when outer loop runs 3rd time, inner loop runs 3 times (horizontal values), print 1,2,3 with tab space

when outer loop runs 4th time, inner loop runs 4 times (horizontal values), print 1,2,3,4 with tab space

Solution

```
outer_var=1
outer_cond=int(input())
while(outer_var<=outer_cond):
    inner_var=1
    while(inner_var<=outer_var):
        print(inner_var,end="\t")
        inner_var=inner_var+1
    print()#New line
    outer_var=outer_var+1
#Input
4
#Output
1
1       2
1       2       3
1       2       3       4
```

Example 32: Print the following sequence using for loop

```
1
2       3
4       5       6
7       8       9       10
```

Solution (Step wise procedure is explained in Table 4.1)

```
var=1 # It is used to print 1 2 3 4 5 6 7 8 9 10
for outer_var in range(1, 5): # The loop will work for fout times i.e. 5-1=4
    for inner_var in range(outer_var):
        print(var, end=' ')
        var=var+1
    print()
```

Table 3.1: Example 32 step wise procedure explained

Step Numbers	outer_var	Var	inner_var	Printing Pattern
Step 1	1	1	1	1
Step 2		2	2	
Step 3	2	2	1	1 2
Step 4		3	2	1 2 3
Step 5		4	3	
Step 6	3	4	1	1 2 3 4
Step 7		5	2	1 2 3 4 5
Step 8		6	3	1 2 3 4 5 6
Step 9		7	4	
Step 10	4	7	1	1 2 3 4 5 6 7
Step 11		8	2	1 2 3 4 5 6 7 8

Step 12		9	3	1 2 3 4 5 6 7 8 9
Step 13		10	4	1 2 3 4 5 6 7 8 9 10
Step 14		11	5	
Step 15	5			

Example 33: Print the following sequence

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

Solution (Step wise procedure is explained in Table 3.2)

```
outer_var=1
while(outer_var<=5):
    inner_var=6-outer_var
    while(inner_var>=1):
        print(inner_var, end='\t')
        inner_var-=1
    outer_var+=1
    print()
```

Explanation: In the print sequence column, due to the space limitation of a page, tab space is shown as space.

Table 3.2: Step wise procedure of example 33

Step Numbers	outer_var	outer_var<=5	inner_var	inner_var>=1	Print the sequence
Step 1	1	1<=5	5	5>=1	5
Step 2			4	4>=1	5 4
Step 3			3	3>=1	5 4 3
Step 4			2	2>=1	5 4 3 2
Step 5			1	1>=1	5 4 3 2 1
Step 6			0	0>=1	
Step 7	2	2<=5	4	4>=1	5 4 3 2 1 4

Step 8			3	3>=1	5 4 3 2 1 4 3
Step 9			2	2>=1	5 4 3 2 1 4 3 2
Step 10			1	1>=1	5 4 3 2 1 4 3 2 1
Step 11			0	0>=1	
Step 12	3	3<=5	3	3>=1	5 4 3 2 1 4 3 2 1 3
Step 13			2	2>=1	5 4 3 2 1 4 3 2 1 3 2
Step 14			1	1>=1	5 4 3 2 1 4 3 2 1 3 2 1
Step 15			0	0>=1	
Step 16	4	4<=5	2	2>=1	5 4 3 2 1 4 3 2 1 3 2 1 2
Step 17			1	1>=1	5 4 3 2 1 4 3 2 1 3 2 1 2 1
Step 18			0	0>=1	
Step 19	5	5<=5	1	1>=1	5 4 3 2 1 4 3 2 1 3 2 1 2 1 1
Step 20			0	0>=1	
Step 21	6	6<=5			

Example 34: Print Armstrong Numbers between 50 to N (Nested for and while loop)

```
start_range = 50
end_range = int(input())
for chk_num in range(start_range, end_range + 1):
    order_num = len(str(chk_num))#number of digits in a number
    sum = 0
```

```
    temp_num = chk_num
    while temp_num > 0:
        digit = temp_num % 10
        sum += digit ** order_num
        temp_num //= 10
    if chk_num == sum:
        print(chk_num)# Armstrong Number printing
```

Test Case 1:
#Input
500
#output
153
370
371
407

Test Case 2:
#Input
200
#Output
153

Explanation: A number is said to be an Armstrong number of order n if pqr… = $p^n + q^n + r^n$ + ……. where n is number of digits in the number
Example: $370 = 3^3 + 7^3 + 0^3 = 370$

Practice Questions

Example 35: Write a program to generate the pattern
1
10
101
1010
10101
101010
……..
…….
Sample Input 1:
4
Sample Output1:
1
10

101
1010
Sample Input 2:
3
Sample Output2:
1
10
101
Sample Input3:
6
Sample Output3
1
10
101
1010
10101
101010

Solution (Step wise procedure is explained in Table 3.3):

```
int_number=int(input())
outer_var=1
while outer_var<=int_number:
    count=1
    inner_var=1
    while inner_var<=outer_var:
        if count%2!=0:
            print("1",end="");
        else:
            print("0",end="");
        count=count+1
        inner_var=inner_var+1
    outer_var=outer_var+1
    print("")
```

Table 3.3: Step wise procedure of example 35 with int_numbers=5

Step Numbers	outer_var	Outer Condition	count	inner_var	Inner Condition	Printing Sequence
Step 1	1	1<=5	1	1	1<=1	1
Step 2			2	2	2<=1	
Step 3	2	2<=5	1	1	1<=2	1

						1
Step 4			2	2	2<=2	1 10
Step 5			3	3	3<=2	
Step 6	3	3<=5	1	1	1<=3	1 10 1
Step 7			2	2	2<=3	1 10 10
Step 8			3	3	3<=3	1 10 101
Step 9			4	4	4<=3	
Step 10	4	4<=5	1	1	1<=4	1 10 101 1
Step 11			2	2	2<=4	1 10 101 10
Step 12			3	3	3<=4	1 10 101 101
Step 13			4	4	4<=4	1 10 101 1010
Step 14			5	5	5<=4	
Step 15	5	5<=5	1	1	1<=5	1 10 101 1010 1
Step 16			2	2	2<=5	1 10 101 1010

						10
Step 17			3	3	3<=5	1 10 101 1010 101
Step 18			4	4	4<=5	1 10 101 1010 1010
Step 19			5	5	5<=5	1 10 101 1010 10101
Step 20			6	6	6<=5	
Step 21	6	6<=5				

Example 36: Make a program that tells whether the input positive integer is the sum of four consecutive numbers. Print „Thumbs Up" when it is the sum of four consecutive numbers else print „Thumbs Down".
Input: A positive Integer
Output: Thumbs Up or Thumbs Down

Test Case1:
#Input
6
#Output
Thumbs Up #0+1+2+3

Test Case2:
#Input
16
#Output
Thumbs Down

Test Case3:
#Input
86
#Output

Thumbs Up # 20+21+22+23
Solution:

```
int_number=int(input())
temp_res=int_number//4
if(((temp_res-1 + temp_res-2 + temp_res-3 + temp_res)==int_number) or ((temp_res-1 + temp_res-2 + temp_res + temp_res+1)==int_number) or ((temp_res-1 + temp_res + temp_res+1 + temp_res+2)==int_number) or ((temp_res + temp_res+1 + temp_res+2 + temp_res+3)==int_number)):
   print("Thumbs Up")
else:
   print("Thumbs Down")
```

Example 37: Make a program for finding out whether a number is a prime number or not.
Test Case 1:
#Input
7
#Output
Prime number

Test Case 2:
#Input
23
#Output
Prime number

Test Case 3:
#Input
72
#Output
Not a Prime number

Test Case 4:
#Input
700
#Output
Not a Prime number

Solution:
Method 1:

```
int_number =int(input())
flag_prime = True # flag variable
if int_number > 1:
```

```
    for var in range(2, int_number):
        if (int_number % var) == 0:
            flag_prime = False
            break
if flag_prime==True:
    print("Prime number")
else:
    print("Not a Prime number")
```

Method 2:

```
int_number =int(input())
if int_number > 1:
    for var in range(2, int_number):
        if (int_number % var) == 0:
            break
if (var==(int_number-1) or (var == int_number)):
    print("Prime number")
else:
    print("Not a Prime number")
```

Method 3:

```
int_number =int(input())
if int_number > 1:
   for var in range(2,int_number):
       if (int_number % var) == 0:
           print("Not a Prime number")
           break
   else:
       print("Prime number")
else:
   print("Not a Prime number")
```

3.4 SELF-CHECK QUESTIONS

1. Is else part is mandatory while using for and while loop? True/False
2. Is for loop can be nested in while loop? True/False
3. Is while loop can be nested in for loop? True/false
4. What is the output of the following code?

```
if 10+7==17:
    print("Yes")
else:
    print("No")
```

```
print("Last Line")
```

a) Yes
   Last Line
b) No
   Last Line
c) Yes
   No
   Last Line
d) Last Line

5. What is the output of the following code?

```
var = 6
if (var > 5):
   var = var * 3;
if (var > 10):
   var = 0;
print(var)
```

a) 0
b) 18
c) 18
   0
d) None of these

3.5 SUMMARY

Conditional statements help the students in learning the flow of the program and these statements can be executed in python when certain conditions are met. Different examples are elaborated for if, if-else, if-elif-else along with nested conditional statements. The students will be able to understand different types of loops like for and while. They will be able to make programs and dry run their code. Step by step procedure has been explained with many examples of looping constructs. This module targets to inculcate the basics of loops and conditional statements in students.

3.6 Practice Questions

1. Which of the following statements will be executed in python version 3?
   a) if (7,6): print("CSE-CA")
   b) if (7,6):
         print("CSE-CA")
   c) if (7,6):

         print("CSE-CA")

d) if (7,6):
    print("CSE-CA")

2. Let two variables have been initialized as:
   var1=23
   var2=7
   Write a python program to find the remainder when you divide var1 by var2 and assign the result to a variable result_var?
3. Is the following code has valid syntax or not?
   var1=12
   var2=6
   if var1 > var2: if var1 > 10: print('CSE-CA')
4. What will be the output of the following code?
   x=12
   y=6
   if x < y: print('CSE')
   elif y < x: print('CA')
   else: print('CSE-CA')

   a) CSE
   b) CA
   c) CSE-CA
   d) Error
5. Correctly choose the options that will tell how many times the loop will work?
   var1=-5
   while var1>0:
      print("CSE")
      print("CA")
      print("CSE-CA")
      var1=var1+1
   a) 0 times
   b) -5 times
   c) 5 times
   d) Infinite times
6. What will be the output of the following code?
   x = 2.00
   if (x - int(x) == 0):
      print("Whole Number")
   else:
      print("Has decimals")

a) Whole Number
b) Has decimals
c) 2.00
d) Error

7. What will the output of the following code?

```
sum_numbers = 0
num=range(7)
for var in num:
    sum_numbers = sum_numbers+var
print(sum_numbers)
```

a) 28
b) 21
c) 15
d) 0

8. Choose the correct output

```
grad_l = ["CSE","ME","Civil","ECE"]
for index in range(len(grad_l)):
    print(len(grad_l))
```

a) 4
   4
   4
   4
b) 3
   3
   3
   3
c) CSE
   ME
   Civil
   ECE
d) None of the option is correct

9. Construct a program for doing a reverse of an N-digit number using for and while loop both.
10. Elaborate differences between while and for loop.

PYTHON PROGRAMMING

UNIT IV: METHODS AND FUNCTIONS

STRUCTURE

4.0 Objectives

4.1 Introduction

4.2 Functions

4.2.1 Inbuilt Functions

4.2.2 User-Defined Functions

4.2.2.1 Function Definition Arguments

4.2.2.2 Special Case of Keyword Arguments

4.2.2.3 Special Case of Positional Arguments

4.2.2.4 Pass by Reference or Pass by Value

4.2.3 Anonymous Function or Lambda Function

4.2.3.1 Lambda with Filter Function

4.2.3.2 Lambda Function with Map Function

4.2.3.3 Lambda with Reduce Function

4.3 Self-check Questions

4.4 Summary

4.5 Practice Question

4.0 OBJECTIVES

- Familiarize how to build, define and write functions
- Learn how to use different types of arguments in the function definition
- Learn and examine how lambda functions are accessed and used.
- Usage of filter, map and reduce functions

4.1 INTRODUCTION

Functions are mainly used to perform particular tasks for programmers, users and associated stock holders. They basically used to cope with the input and output of the computer programs. The most important element in any real time projects is data and functions are the best and effective way to deal with projects that can be small, medium, large as well as complex. When we work on large and complex problems or projects, data duplication is a problem. To avoid this drawback, codes are reusable with the help of functions. This module targets to give better understanding of inbuilt functions, user-defined functions along with lambda functions. Different examples of each type of functions are mentioned along with proper output. Many inbuilt functions are explained like max(), abs(), type() etc., with proper syntax and programs. Various programs of user defined functions are also elaborated. Different types of arguments in functions passing are discussed with appropriate programs. Arguments like positional, keyword, default and special cases of variable-length arguments. At the end, lambda functions explanation along with filter function, map function and reduce function are described and discussed extensively followed with self-check questions, summary and unit end questions.

4.2 FUNCTIONS

A function in python is a block of statements that do a particular work or task, this work or task can be related to any logical, any computational or any evaluation task. The idea is to combine the common statements such that code is reusable and helps in avoiding writing the same code again and again. Functions help in dividing the larger programs into smaller blocks that help in managing the whole program easily.

Functions are divided into two types:

1. Inbuilt functions
2. User-defined functions
3. Anonymous functions

4.2.1 Inbuilt functions

The python interpreter has many functions built into it and these functions have their pre-defined functionalities. Some of these inbuilt functions are listed below:

a) abs()
b) print()
c) input()

d) chr()
e) max()
f) min()
g) int()
h) float()
i) type()
j) round()

Inbuilt Functions with Examples

a) abs() – This inbuilt function returns the absolute value of a number and this number can be an integer, complex number or float. This function takes only one argument and it returns the absolute value if the passed number is an integer or float number and it returns the magnitude of the number if the passed number is complex.

Example 1: # Python code for abs() built-in function

```
int_var     = -101 #Initialising variable with an integer value
float_var  = -46.87 #Initialising variable with an float value
complex_var = (5 - 12j) #Initialising variable with an complex value
print("Absolute Value",abs(int_var))
print("Absolute Value",abs(float_var))
print("Absolute Value",abs(complex_var))
#Output
Absolute Value 101
Absolute Value 46.87
Absolute Value 13.0
```

b) print() – It generates the output from a passed value or many values. The output can be in the form of a screen/standard output device or text stream file. In the print() function, you can have more zero or more expressions and all these expressions are separated by using a comma operator. print() functions have five arguments as expressions and these are mentioned in syntax along with its explanation. All these arguments are optional and keyword arguments.

Syntax:

```
print(*object, sep=' ', end='\n', file=sys.stdout, flush=False)
```

where

*object – object denotes the screen output and * indicates the number of objects as screen output.

sep – object denotes the screen output and is separated using the sep value. The default value of sep=‟ „

end – It is used to print at the last

file – By default, its value is sys.stdout and must use import sys in the program if you use sys.stdout and it helps in printing objects on the screen. You can also use any object having a write(string) method instead of the default value.
flush – The internal buffer is forcibly flushed if its value is True. By default its value is False.

Example 2: Usage of print() with many objects

```
print("CSE","CA")
print("Btech")
print()
print("ECE","ME")
#Output
CSE CA
Btech

ECE ME
```

NOTE: At the end of the print() line, the new line is inserted as end="\n" by default. One more thing to be noted in the first and fourth print() function, multiple objects are mentioned and when they are printed there is space between them as sep=" „ by default.

Example 3: Usage of print() function with sep argument

```
print("CSE","CA",sep="\t")
print("Btech")
print()
print("ECE","ME",sep="**")

#Output
CSE	CA
Btech

ECE**ME
```

Example 4: Usage of print() function with sep and end arguments

```
print("CSE","CA",sep="\t",end='\n\n')
print("Btech")
print()
print("ECE","ME",sep="**",end='&&')
#Output
CSE	CA

Btech
```

ECE**ME&&

Example 4: Usage of print() function with file argument

```
Python_src_file =open(„python_basics.txt“, “w“)
print(“Jagat Guru Nanak Dev PSOU”, file= Python_src_file)
Python_src_file.close()
```

c) input() – In this inbuilt function, the user enters the input and then this inbuilt function assesses this input expression whether it is correct or not. If it is correct then it is assigned to the variable unless it generates a syntax error or an exception is raised.

Syntax:

```
input([prompt])  # here, prompt is optional
```

Example 5:

```
str_var = input("Enter Official name")
print(str_var)
int_var=int(input(" Enter Emp-Id")) #typecasting to integer, here int() is another inbuilt function
print(int_var)
float_var=float(input("Enter Gross Salary"))
# typecasting to float, here float() is another inbuilt function
print(float_var)
#Output
Enter Official nameVinay
Vinay
Enter Emp-Id178
178
Enter Gross Salary26734.56
26734.56
```

d) chr() – This inbuilt function takes the parameter of integer type (valid Unicode) and it returns a character corresponding to that passed integer.

Example 6: Print the characters using chr()

```
print(chr(79))
print(chr(107))
print(chr(1178))
print(chr(45))
#Output
O
k
```

Ķ
-

e) max() – It takes a python object or many objects (optional) as argument(s) with another argument „key‟ and it is optional. Key is the function that compares the objects. One more argument is default and it is also optional. If an object is empty then the default value is used. This function returns the maximum value if it is an integer or float object. If it is a string then it returns a lexographic value. If the objects are iterable like lists, tuples, or dictionary then it returns the largest item of the iterable.

Syntax:
max(object1,object 2, … object n, key, default)

Example 7: Usage of max() function with many objects as arguments

```
int_var1=45
float_var2=78.23
float_var3=34.893
max_var=max(int_var1,float_var2,float_var3)
print(max_var)
#Output
78.23
```

Example 8: Usage of max() function with list as an argument

```
list_var=[45,78.23,34.893]
max_var=max(list_var)
print(max_var)
#Output
78.23
```

Example 9: Usage of max() function with key=len as an argument

```
str_var1 = "CSE-CA"
str_var2 = "Btech"
str_var3 = "CSE-ME-ECE-CA"
max_var = max(str_var1, str_var2, str_var3,key = len)
print(max_var)
#Output
CSE-ME-ECE-CA
```

f) min() – This function returns the minimum value if integer or float values are passed as argument(s). And it returns lexographic smallest value if the strings are passed as argument(s).

Syntax:
min(object1, object2 …… object n, key, default)
where object 1 is integer, float, string, list, tuple, dictionary
object 2, object 3….. object n are optional
key is optional and this function takes all the passed objects and comparison is performed
the default value is allotted if the given objects are empty

Example 10: Usage of min() function with many objects as arguments
```
int_var1=45
float_var2=78.23
float_var3=34.893
min_var=min(int_var1,float_var2,float_var3)
print(min_var)
#Output
34.893
```

Example 11: Usage of min() function with list as an argument
```
list_var=[45,78.23,34.893]
min_var=min(list_var)
print(min_var)
#Output
34.893
```

Example 12: Usage of min() function with key=len as an argument
```
str_var1 = "CSE-CA"
str_var2 = "Btech"
str_var3 = "CSE-ME-ECE-CA"
min_var = min(str_var1, str_var2, str_var3,key = len)
print(min_var)
#Output
Btech
```

g) int() – This functions converts the specified value as argument into an integer number [1].
Syntax:
int(string, any_base)
where the string is a combination of elements of 1‟s and 0‟s
any_base indicates any base of the number

Example 13: Usage of int()
```
binary_var="110"
octal_var="110"
hexa_var="A0A"
```

```
decimal_convertfrom_binary=int(binary_var,2)
print(decimal_convertfrom_binary)
decimal_convertfrom_octal=int(octal_var,8)
print(decimal_convertfrom_octal)
decimal_convertfrom_hexa=int(hexa_var,16)
print(decimal_convertfrom_hexa)
$Output
6
72
2570
```

Example 14: Converting string to int()

```
str_var=input("Enter the string variable value")
print(str_var)
print(type(str_var))
int_var=int(input("Enter the string variable value"))
print(int_var)
print(101+int_var)
print(type(int_var))
#Output
Enter the string variable value123
123
<class 'str'>
Enter the string variable value123
123
224
<class 'int'>
```

h) float() – This inbuilt function converts the argument value of number or string into float value [2], [3].

Syntax:

```
float([string or number])
```

Example 15: float() function usage

```
float_var1= (float(101))
print(float_var1)
print(type(float_var1))
float_var2= (float(101.67))
print(float_var2)
```

```
print(type(float_var2))

float_var3= (float("101.87"))
print(float_var3)
print(type(float_var3))
float_var4= (float(5e003))
print(float_var4)
print(type(float_var4))
float_var5= (float(5e-003))
print(float_var5)
print(type(float_var5))
float_var6= (float(False))
print(float_var6)
print(type(float_var6))
float_var7= (float(True))
print(float_var7)
print(type(float_var7))
float_var8= (float('abc'))
print(float_var8)
print(type(float_var8))
#output
101.0
<class 'float'>
101.67
<class 'float'>
101.87
<class 'float'>
5000.0
<class 'float'>
0.005
<class 'float'>
0.0
<class 'float'>
1.0
<class 'float'>
ValueError
```

i) type() – It is one method that is widely used for debugging and it returns the class type of the object that Is passed as an argument to the type() method.

Example 16: Usage of type() inbuilt function

```
int_var=10
print(type(int_var) is int)
float_var = 10.45
print(type(float_var) is float)
list_var=[1,3,5]
print(type(list_var) is list)
tuple_var=(1,3,5)
print(type(tuple_var) is list)
float_var = 10.45
print(type(float_var) is not float)
tuple_var=(1,3,5)
print(type(tuple_var) is not list)
#Output
True
True
True
False
False
True
```

j) round() - This is one of the inbuilt functions in Python and it rounds off the number to the parameter value (i.e. ndigits decimal) and if no parameter value is mentioned then it rounds off to the nearest integer [5].

Syntax:
round(number, [ndigits decimal])
where
number is to whom which rounded to be done
ndigits decimal tell up to what decimals rounding of is required and it is optional

Example 17: round() function basic example without optional parameter

```
# Use Case for integers
print(round(123))

# Use Case for floating-point numbers
print(round(123.7))

# Use Case for floating-point numbers
print(round(123.2))

# Use Case for floating-point numbers
```

```
print(round(123.5))

#Output
123
124
123
124
```

Example 18: round() function basic example with optional parameter

```
print(round(15.455,2))
print(round(15.453,2))
print(round(15.457,2))
#Output
15.46
15.45
15.46
```

Example 19

```
print(round("CSE",2))

#Output
TypeError
```

NOTE: If any input other than the number is given in parameter value „number‟ then it generates an error i.e. Type Error. If any input other than a number is given in parameter value „ndigits decimal‟ then it generates and error i.e. TypeError.

4.2.2 User-Defined Functions

The functions that are defined by the user are known as user-defined functions. The user-defined functions can have any name excluding space, pre-defined keywords, and any special character.

Syntax:

```
def function-name(parameters):
        statement(1)
        statement(2) #optional
        …..... #optional
        …..... #optional
        statement(n) #optional
        “””optional documentation string”””
        return #Optional return statement
```

where

def is the keyword and it tells the start of the function header

function-name differentiate the functions used in the program and it follows naming conventions of identifiers and it is unique in nature.
parameters- they are also known as arguments, the values to the function are passed using parameters or arguments and they are optional in the user-defined functions.
: colon tells the end of the function header
Optional documentation string- It is used to tell what the functions do in the program and they are optional to define in the functions
statement(s) – the python user-defined function consists of one or more than one valid statements
return – This helps in returning the value from the function and it is optional in a user-defined function.

Example 20: First Simple example of a function

```
# Construct a user-defined function to print COVID message when it is called
def bye_COVID():
    print("Take both vaccinations with a normal 4 to 6 weeks gap and say bye to COVID")
    print("Do follow social distancing")
    print("Properly use mask")
    print("Avoid unnecessary shopping and walking-out")

#Call the function to print COVID message
bye_COVID()
```

#Output

```
Take both vaccinations with a normal 4 to 6 weeks gap and say bye to COVID
Do follow social distancing
Properly use mask
Avoid unnecessary shopping and walking-out
```

Example 21: Simple example with a parameter

```
# Construct a user-defined function to print COVID message when it is called with a parameter
def bye_COVID(name):
    print("Take both vaccinations with a normal 4 to 6 weeks gap and say bye to COVID")
    print("Do follow social distancing")
    print("Properly use mask")
    print("Avoid unnecessary shopping and walking-out")

# Take input from the user
year_name=input("Enter the year in which it came into existence\n")

#Call the function to print COVID message
bye_COVID(year_name)
```

```
#Output
Enter the year in which it came into existence
19
Take both vaccinations with a normal 4 to 6 weeks gap and say bye to COVID
Do follow social distancing
Properly use mask
Avoid unnecessary shopping and walking-out
```

Example 21: Simple example without passing an appropriate parameter

```
# Construct a user-defined function to print COVID message when it is called with a parameter
def bye_COVID(name):
    print("Take both vaccinations with a normal 4 to 6 weeks gap and say bye to COVID")
    print("Do follow social distancing")
    print("Properly use mask")
    print("Avoid unnecessary shopping and walking-out")

# Take input from the user
year_name=input("Enter the year in which it came into existence\n")

#Call the function to print COVID message
bye_COVID()

#Output
Enter the year in which it came into existence
19
TypeError: bye_COVID() missing 1 required positional argument: 'name'
```

Example 22: Use of return statement in function

```
def sum_of_numbers(num_var1,num_var2,num_var3):
    variables_sum=num_var1+num_var2+num_var3
    return(variables_sum)

variables_summation=sum_of_numbers(13,45,78)
print("Sum is depicted as", variables_summation)

#Output
Sum is depicted as 136
```

Example 23: Use of return when strings are passed as arguments

```
# Construct a user-defined function to print COVID message when it is called with a parameter
def bye_COVID(name,new_variant):
```

```
    print("Take both vaccinations with a normal 4 to 6 weeks gap and say bye to COVID")
    print("Do follow social distancing")
    print("Properly use mask")
    print("Avoid unnecessary shopping and walking-out")
    total=name+new_variant
    return(total)

# Take input from the user
year_name=input("Enter the year in which it came into existence\n")
variant_number=input("Enter the latest variant that is coming\n")
#Call the function to print COVID message
result=bye_COVID(year_name,variant_number)
print(result)

#Output
Enter the year in which it came into existence
2019
Enter the latest variant that is coming
2nd-variant
Take both vaccinations with a normal 4 to 6 weeks gap and say bye to COVID
Do follow social distancing
Properly use mask
Avoid unnecessary shopping and walking-out
20192nd-variant
```

Example 25: Use of return without returning any value

```
def sum_of_numbers(num_var1,num_var2,num_var3):
      variables_sum=num_var1+num_var2+num_var3
    return

variables_summation=sum_of_numbers(13,45,78)
print(variables_summation)

#Output
None
```

NOTE: The arguments/parameters which are specified in the function definition are called formal arguments whereas the arguments/parameters which are specified in the function call are called actual arguments.

4.2.2.1 Function Definition Arguments or Function Formal Arguments

Four types of formal arguments are specified in user-defined functions and these are mentioned below:

A) Positional or Required Arguments
B) Keyword Arguments
C) Default Arguments
D) Variable-length Arguments

A) Positional or Required Arguments

The correct position and an exact number of the arguments are passed to a function or in other words function call and function definition number of arguments must be the same along with the correct position of these arguments is required.

Example 26: Positional Arguments Program

```
def bye_COVID(injection1,injection2):
# Look carefully, first argument have value of "True" and Second have "Yes" values
   if injection1=="True" and injection2=="Yes":
      print("Take both vaccinations with a normal 4 to 6 weeks gap and say bye to COVID")
      print("Do follow social distancing")
      print("Properly use mask")
      print("Avoid unnecessary shopping and walking-out")

# Take input from the user
var_vaccination1=input("Enter Yes or No if vaccination1 has been done or not\n")
var_vaccination2=input("Enter True or False if vaccination2 has been done or not\n")
#Call the function to print COVID message
bye_COVID(var_vaccination2,var_vaccination1)  # Look arguments calling
```

```
#Output
Enter Yes or No if vaccination1 has been done or not
Yes
Enter True or False if vaccination2 has been done or not
True
Take both vaccinations with a normal 4 to 6 weeks gap and say bye to COVID
Do follow social distancing
Properly use mask
Avoid unnecessary shopping and walking-out
```

Example 27: No Positional Arguments in Function Definition but Two Values Are Specified in Function Calling

```
def bye_COVID(): # Look carefully, No formal argument
```

```
    if injection1=="True" and injection2=="Yes":
        print("Take both vaccinations with a normal 4 to 6 weeks gap and say bye to COVID")
        print("Do follow social distancing")
        print("Properly use mask")
        print("Avoid unnecessary shopping and walking-out")

# Take input from the user
var_vaccination1=input("Enter Yes or No if vaccination1 has been done or not\n")
var_vaccination2=input("Enter True or False if vaccination2 has been done or not\n")
#Call the function to print COVID message
bye_COVID(var_vaccination2,var_vaccination1)  # Look arguments calling

#Output
Enter Yes or No if vaccination1 has been done or not
Yes
Enter True or False if vaccination2 has been done or not
True
TypeError: bye_COVID() takes 0 positional arguments but 2 were given
```

Example 28: Positional Arguments in Function Definition but no values are specified in Function Calling

```
def bye_COVID(injection1,injection2): # Look carefully, Two Positional Arguments
    if injection1=="True" and injection2=="Yes":
        print("Take both vaccinations with a normal 4 to 6 weeks gap and say bye to COVID")
        print("Do follow social distancing")
        print("Properly use mask")
        print("Avoid unnecessary shopping and walking-out")

# Take input from the user
var_vaccination1=input("Enter Yes or No if vaccination1 has been done or not\n")
var_vaccination2=input("Enter True or False if vaccination2 has been done or not\n")
#Call the function to print COVID message
bye_COVID()  # Look arguments calling

#Output
Enter Yes or No if vaccination1 has been done or not
Yes
Enter True or False if vaccination2 has been done or not
True
TypeError: bye_COVID() missing 2 required positional arguments: 'injection1' and 'injection2'
```

B) Keyword Arguments

When arguments are passed in the function call, they can or cannot be in the order as formal arguments defined in the function definition. These things are achieved through keyword arguments. Remember that keyword argument must match the arguments of formal arguments [6].

Example 29: Keyword Arguments basic example 1

```
def BYE_COVID(var_string):
    print(var_string)
    return

# function calling
BYE_COVID(var_string = "Take both vaccinations with a normal 4 to 6 weeks gap and say bye to COVID")
```

#Output

Take both vaccinations with a normal 4 to 6 weeks gap and say bye to COVID

Example 30: Keyword Arguments basic example 2

```
def BYE_COVID(var_string1,var_string2):
    print(var_string1)
    print(var_string2)
    return

# function calling
BYE_COVID(var_string2 = "Take both vaccinations with a normal 4 to 6 weeks gap and say bye to COVID", var_string1="Properly use mask")
```

#Output

Properly use mask

Take both vaccinations with a normal 4 to 6 weeks gap and say bye to COVID

C) Default Arguments

If in a function call, no explicit values are given then the formal arguments take default values [8].

Example 31: Keyword Arguments usage in calling function

```
def BYE_COVID(precaution1, precaution2="injection2", precaution3="Mask", precaution4="Social Distance",disease="COVID"):
    print("Take", precaution1, "followed with", precaution2, ",Wear", precaution3, "and Follow rules of", precaution4,"that helps in avoiding", disease)
```

```
# Keyword argument
BYE_COVID(precaution1="injection1")

# Keyword arguments
BYE_COVID(precaution1="injection1",precaution3="N95 or Fully Covered Mouth/Nose
Mask")

# Way of changing arguments
BYE_COVID(precaution3="N95 or Fully Covered Mouth/Nose
Mask",precaution1="injection1")

#Output
Take injection1 followed with injection2 ,Wear Mask and Follow rules of Social Distance that he
lps in avoiding COVID
Take injection1 followed with injection2 ,Wear N95 or Fully Covered Mouth/Nose Mask and Fo
llow rules of Social Distance that helps in avoiding COVID
Take injection1 followed with injection2 ,Wear N95 or Fully Covered Mouth/Nose Mask and Fo
llow rules of Social Distance that helps in avoiding COVID
```

Example 32: Deeper understanding of keyword arguments

```
def BYE_COVID(precaution1, precaution2="injection2", precaution3="Mask",
precaution4="Social Distance",disease="COVID"):
    print("Take", precaution1, "followed with", precaution2, ",Wear", precaution3, "and Follow
rules of", precaution4,"that helps in avoiding", disease)

# Invalid keyword
BYE_COVID(do_precaution1="injection1")      # Run this line individually

# No argument passes
BYE_COVID() # Run this line individually

BYE_COVID(precaution2="inject","injection1") # Run this line individually

# Output
TypeError: BYE_COVID() got an unexpected keyword argument 'do_precaution1'
TypeError: BYE_COVID() missing 1 required positional argument: 'precaution1'
SyntaxError: positional argument follows keyword argument
```

4.2.2.2 Special Case of Keyword Arguments

In this **kwargs as an argument in the function definition is used and it means you can pass any number of keyword arguments along with variable length [7]. Remember the keyword name that is passed in function call must match the keyword name of the function definition.

Example 33: Basic Example of variable length arguments

```
def usage_of_kwargs(**kwargs):
    print(kwargs)

usage_of_kwargs(injection1=True, Mask="Yes", injection2=True, age=67)

#Output
{'injection1': True, 'Mask': 'Yes', 'injection2': True, 'age': 67}
```

Example 34: Another way of printing elements of method defined in previous example 31

```
def usage_of_kwargs(**kwargs):
    for var_key, var_value in kwargs.items():
        print ("%s == %s" %(var_key, var_value))

usage_of_kwargs(injection1=True, Mask="Yes", injection2=True, age=67)

#Output
injection1 == True
Mask == Yes
injection2 == True
age == 67
```

Example 35: Another way of printing elements in keyword arguments

```
def usage_of_kwargs(**kwargs):
    for var_key, var_value in kwargs.items():
        print("The key is {} and its associated value is {}".format(var_key, var_value))

usage_of_kwargs(injection1=True, Mask="Yes", injection2=True, age=67)

#Output
The key is injection1 and its associated value is True
The key is Mask and its associated value is Yes
The key is injection2 and its associated value is True
The key is age and its associated value is 67
```

4.2.2.3 Special Case of Positional Arguments

In this *var_arguments are used in function definition and the var_argumentss are non-keyword variable length arguments [6]. The * denotes to have any number of arguments.

Example 36: Usage of *var_arguments

```
def BYE_COVID(*var_arguments):
    for var_arg in var_arguments:
        print (var_arg)

BYE_COVID('Take injection1', 'Followed with injection2', 'Wear Mask Properly', 'Follow the rules of Social Distance')

#Output
Take injection1
Followed with injection2
Wear Mask Properly
Follow the rules of Social Distance
```

Example 37: Program to show *var_arguments having extra arguments.

```
def BYE_COVID(var_argument1,var_argument2,*var_arguments):
    print("The first argument value is",var_argument1)
    print("The second argument value is",var_argument2)
    for var_arg in var_arguments:
        print (var_arg)

BYE_COVID('COVID GUIDELINES','READ PROPERLY','Take injection1', 'Followed with injection2', 'Wear Mask Properly', 'Follow the rules of Social Distance')
#Output
The first argument value is COVID GUIDELINES
The second argument value is READ PROPERLY
Take injection1
Followed with injection2
Wear Mask Properly
Follow the rules of Social Distance
```

4.2.2.4 Pass by Reference or Pass by value

In python language, every variable is a reference. It means when we pass a variable to the function, every time a new object is created with a new reference.

Example 38: Largest of three numbers

```
def max_three_numbers(var_number1,var_number2,var_number3):
```

```
    if var_number1>var_number2:
        if var_number1>var_number3:
            var_largest=var_number1
        else:
            var_largest=var_number3
    else:
        if var_number2>var_number3:
            var_largest=var_number2
        else:
            var_largest=var_number3
    return(var_largest)
var_number1=int(input())
var_number2=int(input())
var_number3=int(input())
print(max_three_numbers(var_number1,var_number2,var_number3))

#Output
Test case1:
#Inputs
23
36
47
#Output
47
Test case1:
#Inputs
67
50
56
#Output
67
Test case1:
#Inputs
145
567
444
#Output
567
```

Example 39: Program to know whether a year is leap year or not a leap year

```
def leap_or_notleap(normal_year):
    if normal_year%4==0:
        if normal_year%100==0:
            if normal_year%400==0:
                flag=1
            else:
                flag=0
        else:
            flag=1
    else:
        flag=0

    return(flag)

normal_year = int(input())
flag_value=leap_or_notleap(normal_year)
if flag_value==1:
    print("Leap Year")
else:
    print("Not a Leap Year")

#Output
Test Case 1:
#Input
2000
#Output
Leap Year
Test Case 2:
#Input
1900
#Output
Not a Leap Year
Test Case 3:
#Input
2020
#Output
Leap Year
Test Case 4:
#Input
2021
```

#Output
Not a Leap Year

Example 40: Find factorial of a given number using user defined functions

```
def num_factorial(formal_number):
    if formal_number == 0:
        return 1
    else:
        return formal_number * num_factorial(formal_number-1)

var_number=int(input("Enter the number for factorial computation\n"))
print("The factorial is", num_factorial(var_number))
```

#Output
Enter the number for factorial computation
7
The factorial is 5040

Example 41: Write a program to generate the pattern using functions
A
AB
ABA
ABAB
ABABA
ABABAB
……..
…….
Sample Input 1:
4
Sample Output1:
A
AB
ABA
ABAB
Sample Input 2:
3
Sample Output2:
A
AB
ABA
Sample Input3:

6
Sample Output3
A
AB
ABA
ABAB
ABABA
ABABAB

Solution (Step wise procedure is explained in table 4.1):

```
def pattern_making(int_number):
    outer_var=1
    while outer_var<=int_number:
        count=1
        inner_var=1
        while inner_var<=outer_var:
            if count%2!=0:
                print("A",end="");
            else:
                print("B",end="");
            count=count+1
            inner_var=inner_var+1
        outer_var=outer_var+1
        print("")

int_number=int(input())
pattern_making(int_number)
```

Table 4.1: Pattern making by assuming int_number=5

Step Numbers	outer_var	Outer Condition	count	inner_var	Inner Condition	Printing Sequence
Step 1	1	1<=5	1	1	1<=1	A
Step 2			2	2	2<=1	
Step 3	2	2<=5	1	1	1<=2	A A
Step 4			2	2	2<=2	A AB
Step 5			3	3	3<=2	
Step 6	3	3<=5	1	1	1<=3	A AB

						A
Step 7			2	2	2<=3	A AB AB
Step 8			3	3	3<=3	A AB ABA
Step 9			4	4	4<=3	
Step 10	4	4<=5	1	1	1<=4	A AB ABA A
Step 11			2	2	2<=4	A AB ABA AB
Step 12			3	3	3<=4	A AB ABA ABA
Step 13			4	4	4<=4	A AB ABA ABAB
Step 14			5	5	5<=4	
Step 15	5	5<=5	1	1	1<=5	A AB ABA ABAB A
Step 16			2	2	2<=5	A AB ABA ABAB AB
Step 17			3	3	3<=5	A AB ABA ABAB ABA

Step 18			4	4	4<=5	1 A AB ABA ABAB ABAB
Step 19			5	5	5<=5	A AB ABA ABAB ABABA
Step 20			6	6	6<=5	
Step 21	6	6<=5				

4.2.3 Anonymous function or Lambda Function

The keyword used in the creation of lambda function is „lambda" and they are popularly known as single line function. They are known as anonymous functions (a function without having a name). They are different from normal functions as this function does not use keywords like „return" and „def" [9].

Syntax:
Lambda multiple_arguments: expression
where expression returns an object and it is only one in the whole function
multiple_arguments used a comma to separate multiple arguments

Example 42: Example to differentiate between normal function and lambda function

```
def multiply_25(var_int1): # Normal function definition
   return var_int1*25

lambda_25 = lambda var_int1:var_int1*25 #lambda function definition

var_int1=int(input("User is entering the number which he/she is required to be multiply by 25 is
"))
print(multiply_25(var_int1))#Normal function calling
print(lambda_25(var_int1))#lambda function calling
```

#Input
User is entering the number which he/she is required to be multiply by 25 is 8
#Output
200
200

Example 43: Sum of three numbers for differentiating between def and lambda functions.

```
def summation_of_3num(var_int1,var_int2,var_int3):
    return var_int1+var_int2+var_int3

lambda_3num = lambda var_int1,var_int2,var_int3:var_int1+var_int2+var_int3

var_int1=int(input("User is entering the first integer number"))
var_int2=int(input("User is entering the second integer number"))
var_int3=int(input("User is entering the third integer number"))
print(summation_of_3num(var_int1,var_int2,var_int3))
print(lambda_3num(var_int1,var_int2,var_int3))

#Input
User is entering the first integer number23
User is entering the second integer number67
User is entering the third integer number40

#Output
130
130
```

4.2.3.1 Lambda with filter function

filter function is having two arguments, one argument is a function that helps in filtering and the other argument is iterator like list, tuples, sets, etc. [10]. It returns values that are passing the filtering condition. Only one iterator is passed in the filter function.

Syntax:

```
filter(lambda_function,iterator)
```

Example 44: Program to find out elements greater than 25 in a list

```
int_list1=[24,26,32,18,10,75]
result_greater_25 = filter(lambda var_int: var_int>25, int_list1)
print(type(result_greater_25))
print(list(result_greater_25))

#Output
<class 'filter'>
[26, 32, 75]
```

Explanation

1. In the first line, a list is defined with integer numbers.

2. In the second line, the result_greater_25 variable will store the values returned by the filter function.
3. In the second line, list each element is run by lambda function, and when filtering criteria is True (i.e. when list element > 25) then it returns its value.
4. In the third line, the type of the returned values are printed.
5. In the last line, the results are printed that are returned by the filter function.

Example 45: Program to multiple every element of list using filter function (look carefully at the output)

```
int_list1=[24,26,32,18,10,75]
result_multiply_3 = filter(lambda var_int: var_int*3, int_list1)
print(list(result_multiply_3))

#Output
[24, 26, 32, 18, 10, 75]# Here value is not modified
```

4.2.3.2 Lambda function with Map Function

map function is having two arguments, one argument is a function and the other arguments are iterator like list, tuples, sets, etc. The map function has one or more iterators. It returns the modified values to the resultant variable.

Syntax:

```
map(lambda_function,iterator)
```

Example 46: Program to multiple every element of list using map function

```
int_list1=[24,26,32,18,10,75]
result_multiply_3 = map(lambda var_int: var_int*3, int_list1)
print(type(result_multiply_3))
print(list(result_multiply_3))

#Output
<class 'map'>
[72, 78, 96, 54, 30, 225]
```

Example 47: Map function with lists (swapcase converts lower to upper and vice-versa)

```
str_list1=['take InjecTion1','Followed with Injection2','wEAr MasK','FoLLOW RULES of social distancing']

result_swapcase_alphabets = map(lambda var_chr: str.swapcase(var_chr), str_list1)
print(list(result_swapcase_alphabets))
```

#Output
['TAKE iNJECtION1', 'fOLLOWED WITH iNJECTION2', 'WeaR mASk', 'fOllow rules OF SO CIAL DISTANCING']

4.2.3.3 Lambda with Reduce Function

The reduce function belongs to the functools module and this function is having two arguments, one argument is lambda function and the other argument is iterator like list, tuple, sets, etc. This function performs repetitive operation over the iterable elements in pairs and the new reduced result is returned.

Syntax:
reduce(lambda_function, iterator)

Example 48: Factorial of a number using reduce function

```
from functools import reduce
int_list1 = [1,2,3,4,5,6]
print(reduce(lambda var1,var2:var1*var2, int_list1))
```

#Output
720

Explanation: Here, firstly 1 is multiplied with 2, then this result 2 is multiplied with 3, then this result 6 is multiplied with 4, then this result 24 is multiplied with 5, then this result 120 is multiplied with 6, and the final result of 720 is returned.

Example 49: Find smallest element in the list using reduce function

```
from functools import reduce
int_list1 = [-56,2,89,234,-78,-452]
print(reduce(lambda var1,var2:var1 if var1<var2 else var2, int_list1))
```

#Output
-452

4.3 SELF-CHECK QUESTIONS

1. Predict the following code output:

```
def output_func(var_int1,var_int2 = -17):
   print(var_int1,var_int2)
output_func(-89)
```

   a) -89 -17 b) -17 c) -89 d) -17 -89

2. Look at the below code and what line says?

```
def output_func(var_int1,var_int2 = -17):
   print(var_int1,var_int2)
output_func(-89)
```

a) Function definition
b) Function calling
c) Function header
d) Function tail

3. If Function does not have a return statement, it automatically returns None (True/False)
4. In functions, positional arguments must follow keyword arguments (True/False)
5. Tell the output of the following code

```
var_int1=10
def summation_with_five(var_int1):
  var_int1=var_int1+5
  return var_int1
summation_with_five(5)
print("Value is=",var_int1)
```

a) Value is= 10
b) Value is= 15
c) Value is= 20
d) Error

4.4 SUMMARY

The students will be able to make different programs with the help of functions. This module helps the students in understanding the concepts of inbuilt, user-defined and anonymous functions. Different types of arguments that are used in functions are well explained with suitable examples. Different inbuilt functions are mentioned, lambda function usage with map, filter and reduce functions are elaborated. This chapter helps the students to know about the reusability concept is important and functions play an important role in the reusability of code. The students can now proceed with advanced concepts of python that will help them in making projects.

4.5 PRACTICE QUESTIONS

1. The output of the following inbuilt function is
   A. print(round(-14.2378,2))
   B. print(type(type(type(float))))
   C. print(max([-34,78,-89,-12,12]))
2. If you are not aware about the number of arguments to be passed in function definition, then arbitrary arguments are used. (True / False)
3. Differentiate lambda function with map and reduce function
4. How *var_arguments and **kwargs are different and elaborate with an suitable program?
5. What is the code output?

```
var_int1 = 3
var_int3 = lambda var_int2: var_int2*var_int1**var_int2
print(var_int3(4))
```

a) 324 b) 36 c) 24 d) 496

6. A number of statements are included in anonymous function like lambda
   a) True b) False
7. The correct output of the code is

```
print(float('1e-003'))
```

a) 0.001 b) 0.003 c) 0.01 d) 0.03

8. What will be printed?

```
def bye_COVID():
    print('Take injections')

bye_COVID()
bye_COVID()
```

9. Use functions to construct a program for figuring out a number is prime or not.
10. Use functions to construct a program that helps in finding out the smallest of four integer numbers.

REFERENCES

[1] https://www.geeksforgeeks.org/python-int-function/
[2] https://www.techbeamers.com/python-float-function/
[3] https://www.tutorialsteacher.com/python/float-method
[4] https://www.geeksforgeeks.org/python-type-function/
[5] https://www.programiz.com/python-programming/methods/built-in/round
[6] https://levelup.gitconnected.com/5-types-of-arguments-in-python-function-definition-e0e2a2cafd29
[7] https://www.digitalocean.com/community/tutorials/how-to-use-args-and-kwargs-in-python-3
[8] https://www.geeksforgeeks.org/default-arguments-in-python/
[9] https://www.geeksforgeeks.org/python-lambda-anonymous-functions-filter-map-reduce/
[10] https://www.guru99.com/python-lambda-function.html

UNIT V: OBJECT-ORIENTED PROGRAMMING

STRUCTURE

5.0 Objectives

5.1 Object-Oriented Programming (OOP)

5.2 Building Blocks of OOPS in Python

- **5.2.1 Defining a Class**
- **5.2.2 Object Instantiation**
- **5.2.3 Invoking Methods**
- **5.2.4 Class Variable vs Instance Variable**

5.3 Four Principles of OOPs

- **5.3.1 Encapsulation**
- **5.3.2 Abstraction**
- **5.3.3 Inheritance**
- **5.3.4 Polymorphism**

5.4 Special Methods in OOPS

5.5 Modules and Packages

- **5.5.1 In-built Modules**
- **5.5.2 User-Defined Modules**
- **5.5.3 Alternative form of import statement**

5.5.4 Packages in Python

5.5.5 Python Packages vs Python Modules

5.5.6 Installing Python packages (pip-PyPi)

5.0 OBJECTIVES

In this chapter we address Python classes that can be used in a way similar to C structures, but also in a complete object-oriented way. In the first two subsection, we shall explore the use of classes as constructs for the benefit of readers who are not object-oriented programmers. The rest of the chapter deals with Python's OOPs. This is just a summary of Python's structures and constructs which is clear demonstration of object-oriented programming itself.

5.1 OBJECT-ORIENTED PROGRAMMING (OOP)

Object-oriented programming is a methodology in the programming sector that offers a tool for structuring programmes to package properties and its corresponding behaviors. OOP is often used by Python programmers because it makes programming more reusable and makes working with bigger programs easier. The methodology is used to bind properties and privileges in order to structure the program into individual objects. As a result, OOP finds it easy to follow the principle of "Don't Repeat Yourself" (DRY).

OOPs often benefit from the ability of users to represent data as a single relationship to evolving market issues. For e.g., if you were to build the programme representing the management structure of your employees in such a way that the classes involved represent something such as an employee and then objects are specific example or otherwise properties involving each employee. Likewise, one can think of another example by considering the software as a kind of factory mounting line such that a device part processes some raw material at each stage of the assembly line and eventually with additional functionalities transforms it into a finished product. Therefore, it is eventual that an object will include information such as raw or previously manufactured materials on a line at each point of time, and actions, such as the movement corresponding to every part on an assembly line.

Another traditional model for programming is procedural programming, which structures a programme, in that it offers sequentially a series of steps in the form of functions and code blocks to complete a task. The main takeover is that in Python OOP is not only represented by the details, but also by the general structure of the program.

5.2 BUILDING BLOCKS OF OOPS IN PYTHON

5.2.1 Defining a Class

There are many reasons why Python allows developers to define new classes. Classes tailored to a certain program would allow the application software to be created, debugged, read and maintained more intuitively and easily.

A prototype specified by the user or user-defined prototype for an object which sets out a number of attributes characterize the object belonging to the specific class. Since a class can be specified once and repeated several times, OOP programmes keep you from repeating code. The attributes include its data members and processes, which are obtained through the dot notation. Both

Python data types are classes, and Python gives you powerful tools to handle all aspects of the actions of a class. With the class keyword, you can define a class as:

```
class ClassName:
    'Optional string for documentation purposes‟
    body
```

Here body is a set of Python statements, usually function definitions and variable assignments. However, the assignments or function definitions are not necessary and the defined body may only be a single line statement.

Many Python users, on the other hand, are unaware of Python's strong reliance on classes under the hood until they learn what a class is, just as we have progressed in this class so far without learning what classes are. The below code is an example of creating a null operation using pass statement while defining *Player* class such that nothing happens on the execution of the below sample code:

```
class Player:
    pass
```

By standard the class identifiers are in CapCase, i.e. the first letter is capitalized on each part word to differentiate and making it much easier for the developer to read around.

5.2.2 Object Instantiation

Only the object's definition or outline is created when we define a class. There will be no memory allocation until the object is created. True data or information is stored in the objector instance inst as:

inst = ClassName()

Likewise, after defining the class you can build a new class type by calling the class name as a function and this is generally referred to as a class instance.

One can create (we will name it *new_player*) an instance of the *Player* class and print the type of the variable *player_type* as:

Player.py

```
class Player:
      player_type=”batsman”
```

```
new_player=Player()
print(type(new_player))
```

Running the program will return the type of the variable as:

```
Output:
<class '__main__.Player'>
```

Further the value corresponding to instance of defined class can be done through the usage of dot notation as:

Player.py

```
class Player:

        #class attribute

        player_type=”batsman”

new_player=Player()
print(new_player.player_type)
```

Running the program will return the value of the variable as:

```
Output:
batsman
```

5.2.3 Invoking Methods

Classes may also have features such as methods that are only applicable to objects of that type. These functions are specified within the class and perform any behavior that is beneficial to that particular object category. Methods have two distinctions, much like functions:

- In order to make clear the relationship between the class and the method the methods are specified in a class description.
- The method invoking syntax is distinct from the function call syntax

Player.py

```
class Player:

        #instance attributes

        def init (self, matches, name):
            self.matches = matches
            self.name=name

        def odi_history(self, runs):
              return "{} has scored {} number of runs".format(self.name, runs)

         def t20_history(self, runs):
              return "{} has scored {} number of runs".format(self.name, runs)

        def tennis(self):
              return "{} loves playing tennis".format(self.name)

#object instantiation

new_player= Player(1987, "Rishabh Pant")

#instance methods calling

print(new_player.odi_history(687))
print(new_player.t20_history(1056))
print(new_player.tennis())
```

5.2.4 Class Variable Vs Instance Variable

Class Variable is a variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are. A class variable or instance variable that holds data associated with a class and its objects. Defined apart from any method, class variables are usually placed below the class header and before the constructor method and other methods by default.

An instance is an object that is constructed from a class and includes actual data, while a class is the blueprint. It may refer to as a variable that is defined inside a method and belongs only to the current instance of a class.

```
class Player:
```

```
        #class attribute

        player_type=”batsman”

        #instance attribute

        def init (self, matches, name):
            self.matches = matches
            self.name=name

#parametric instantiation of Player class

player1=Player(124, “Rohit Sharma”)
player2=Player(132, “Virat Kohli”)

# class variable access

print(“Rohit Sharma is an opening {}”.format(player1. class .player_type))
print(“Virat Kohli is also a {}”.format(player2. class .player_type))

# instance variable access

print(“{} has played {} matches”.format(player1.matches,player1.name))
print(“{} has played {} matches”.format(player2.matches,player2.name))
```

5.3 FOUR PRINCIPLES OF OOPS

5.3.1 Encapsulation

Encapsulation is performed by maintaining a private state of any object within a class. Other objects will only call a list of public functions if this state is not accessible directly instead. Via these functions the object maintains its own state and no other class can modify it unless specifically permitted. You will have to use the methods presented to communicate with the object.

5.3.2 Abstraction

Abstraction is an encapsulation expansion. Data are selected from a wider pool to only provide the corresponding information.

5.3.3 Inheritance

Inheritance is one object's ability to acquire one or more of those properties corresponding to another object. For example, a child inherits his/her parents' traits as well as behavior. Reusability is a significant advantage of inheritance. You may reuse the existing class fields and

methods. There are different types of inheritances in Java: single, multiple, multi-level, hierarchical, and hybrids.

We need a parent class and a child class to incorporate inheritance in the Python programme. Let's see how these two classes can be formed:

```
#creating parent class

class Sports:

        channel=”star productions”

        def details(self):

            print(“The channel broadcasts the live updates from sports corner”)
```

As in the above program, we have created a parent class which was similar to that of defining the normal class. So next, we create child class whose syntax is like similar to method calling where Parent Class is an argument as represented below:

```
class childClass(parentClass):
      #body of child class
```

So, the example code of inheritance where child class is able to access the data members of itself as well as the parent class is shown below:

```
# creating parent class

class Sports:

        channel=”star productions”

        def details(self):

            print(“The channel broadcasts the live updates from sports corner”)

#creating child class

class Cricket(Sports):
```

```
        def no_hrs(self):

            print("Cricket has broadcast time of 10 hours a day")

#main() method

new_chn = Cricket()
print(new_chn.details())
print(new_chn.no_hrs())
```

Types of Inheritance in Python

The five types of inheritance is supported by python namely

a) Single Inheritance

A child class inherits all the features of a parent class in a single inheritance.

```
# creating parent class

class Sports:
        channel="star productions"
        def details(self):
            print("The channel broadcasts the live updates from sports corner")

#creating child class

class Cricket(Sports):
        def no_hrs(self):
             print("Cricket has broadcast time of 10 hours a day")

#main() method

new_chn = Cricket()
print(new_chn.details())
print(new_chn.no_hrs())
```

b) Multiple Inheritance

A class in Python can be derived from one base class and this type of inheritance is referred to as multiple inheritance.

```
# creating parent class

class Sports:
        channel=”star productions”
        def details(self):
            print(“The channel broadcasts the live updates from sports corner”)

#creating child class

class News:
        def no_hrs(self):
             print(“News has broadcast time of 10 hours a day”)

class Match(Sports,News):
        def time(self):
             print(“Match has broadcast time of 7 hours a day”)


#main() method

new_chn = Match()
print(new_chn.details())
print(new_chn.no_hrs())
print(new_chn.time())
```

c) Multilevel Inheritance

We also can inherit a derivative class and the succession of such process is known as multilevel inheritance. In Python, it can be of some depth as:

```
# creating parent class

class Sports:
        channel=”star productions”
        def details(self):
             print(“The channel broadcasts the live updates from sports corner”)

          #creating child class

          class News(Sports):
                def no_hrs(self):
                     print(“Sports News has broadcast time of 10 hours a day”)

          class Cricket(News):
```

```
        def time(self):
            print("Cricket News has broadcast time of 7 hours a day")


    #main() method

    new_chn = Cricket()
    print(new_chn.details())
    print(new_chn.no_hrs())
    print(new_chn.time())
```

d) Hierarchical Inheritance

It is considered as hierarchical inheritance if there are more than one class inherited from the base class. The characteristics typical in child class are included in the base class in hierarchical inheritance.

```
# creating parent class

class Sports:
    channel="star productions"
    def details(self):
        print("The channel broadcasts the live updates from sports corner")

    #creating child class

    class News(Sports):
        def no_hrs(self):
            print("Sports News has broadcast time of 10 hours a day")

    class Match(Sports):
        def time(self):
            print("Cricket News has broadcast time of 7 hours a day")


    #main() method

    new_chn1 = News()
    new_chn2 = Match()
    print(new_chn.details())
    print(new_chn1.no_hrs())
    print(new_chn2.time())
```

e) Hybrid Inheritance

The combination of various types of above-mentioned inheritance is called hybrid inheritance.

```
# creating parent class

  class Sports:
      channel=”star productions”
      def details(self):
          print(“The channel broadcasts the live updates from sports corner”)

        #creating child class

        class News(Sports):
            def no_hrs(self):
                print(“Sports News has broadcast time of 10 hours a day”)

        class Match(Sports):
            def time(self):
                print(“Cricket News has broadcast time of 7 hours a day”)

        class TalkShow(Sports, Match):
            def time(self):
                print(“Talk Show of match has broadcast time of 7 hours a day”)


        #main() method

        new_chn1 = TalkShow()
        print(new_chn.details())
```

The super() function can be applied to in the inherited subclass of a parent class. The super function returns a temporary superclass object, allowing access to its child class through all its methods with an example as:

```
# creating parent class

class Sports:
      channel=”star productions”
      def details(self):
          print(“The channel broadcasts the live updates from sports corner”)

#creating child class
```

```
class Cricket(Sports):
        def no_hrs(self):
            Super().details()
            print("Cricket has broadcast time of 10 hours a day")

#main() method

new_chn = Cricket()
print(new_chn.no_hrs())
```

5.3.4 Polymorphism

Polymorphism has a means for us of using a class much as its parent such that combining forms are not confused. That being said, any subclass of children maintains its own functions/methods. Polymorphism is an object-oriented programming term, meaning multiple forms or different types. Polymorphism allows for one interface with the input of several data types, classes and various inputs. On such process is method overriding or function overloading is a kind of polymorphism in which a variety of methods can be declared with the same name but different parameters and with different parameters types. These strategies may play a similar or different role.

For example, the len function below takes string and list as an argument showing multi-purpose form of the function:

```
len("hello")
len([1,2,3,4,5])
```

Polymorphic Classes and Methods

However, it is well known that there are different forms of cricket i.e., ODI, T20 and Test but there are some attributes as well as methods which might remain same under such regard. Now we will employ the technique of method overloading in such scenario where function will act same as that of another class as shown below:

```
class Batsman:
        runs_scored=2000
        matches=40
        def calculate_average(self):
            return self.run_scored / self.matches

class Bowler:
```

```
        wickets_taken=50
        matches=30
        def calculate_average(self):
            return self.wickets_taken / self.matches

bat= Batsman()
bowl=Bowler()

print(“Average of a batsman: ”, bat.calculate_average())
print(“Average of a bowler: ”, bowl.calculate_average())

#Another form of writing

bat1= Batsman()
bowl1=Bowler()

for(obj in (bat1,bowl1)):
    obj.calculate_average()
```

5.4 SPECIAL METHODS IN OOPS

A set of special object methods is used in all integrated data forms. Double underscores (_) often precede the names of special methods. The interpreter triggers these methods automatically as the programme runs. For instance, a + b is mapped to an internal process, and a.__add__(b), and likewise an indexing operation for defined list a[i], is mapped to a._getitem_(i). Each data type's behavior depends entirely on the selection of particular methods it utilizes. For example, the__new()__method in python is called implicitly before the call of __init()__method such that a new object is returned by __new()__method and then initialized by_init()_method.

```
class Sports:

    def__new__(mtd):
        print ("Here_new()___magic method is being invoked")
        inst = object._new_(mtd)
            return inst

    def__init__(self):
        print ("Here__init__magic method is being invoked")
        self.name=“Virat“

spr = Sports()
```

The output on the execution of above program is :

```
Here__new()___magic method is being invoked
Here___init___magic method is being invoked
```

Moreover, for the case of string operation, the special method __str()____is useful. The usage for the same is represented as below:

```
j=6

print(str(j))

print(int.__str(j)__)
```

The output on the execution of above program is :

```
Output:

„6“
„6“
```

Below table entails the use of special methods in OOPS:

Most commonly used special methods in object-oriented python programming

Methods	**Description**
__new__(cls,args)	The method is used as an alternative method of object instantiation
__init__(self,args)	This method is to be called by the above __new_____ method for the intialisation purposes
__del__(self)	Self-Destructor method

5.5 MODULES AND PACKAGES

In reality, in Python, there are three different ways of defining a module:

- Self-writing of the module itself in Python language
- A module like the re- (regular expression) module can be written in C and loaded dynamically at runtime.
- An integrated module like the module of itertools is inherently contained in the interpreter.

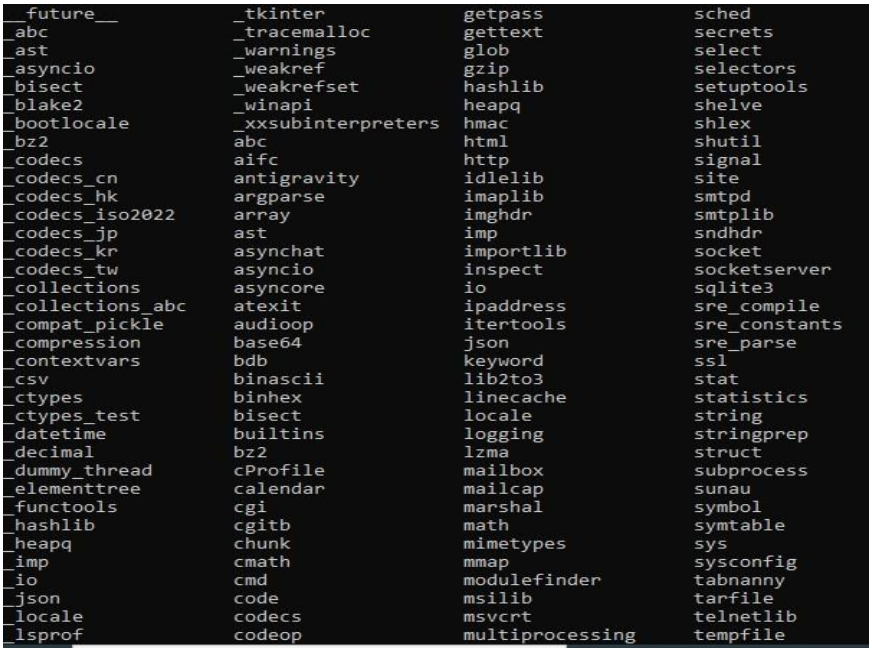
In all three cases: with the import statement, the content of a module are accessed in the same manner.

5.5.1 In-built Modules

Generally, the in-built modules are stored in the directory where the python has been installed and likewise all modules can be displayed using the following command on Python IDLE as:

```
>>> help(„modules“)
```

On running in the python IDLE, the output of the command is as:



```
__future__          _tkinter            getpass             sched
_abc                _tracemalloc        gettext             secrets
_ast                _warnings           glob                select
_asyncio            _weakref            gzip                selectors
_bisect             _weakrefset         hashlib             setuptools
_blake2             _winapi             heapq               shelve
_bootlocale         _xxsubinterpreters  hmac                shlex
_bz2                abc                 html                shutil
_codecs             aifc                http                signal
_codecs_cn          antigravity         idlelib             site
_codecs_hk          argparse            imaplib             smtpd
_codecs_iso2022     array               imghdr              smtplib
_codecs_jp          ast                 imp                 sndhdr
_codecs_kr          asynchat            importlib           socket
_codecs_tw          asyncio             inspect             socketserver
_collections        asyncore            io                  sqlite3
_collections_abc    atexit              ipaddress           sre_compile
_compat_pickle      audioop             itertools           sre_constants
_compression        base64              json                sre_parse
_contextvars        bdb                 keyword             ssl
_csv                binascii            lib2to3             stat
_ctypes             binhex              linecache           statistics
_ctypes_test        bisect              locale              string
_datetime           builtins            logging             stringprep
_decimal            bz2                 lzma                struct
_dummy_thread       cProfile            mailbox             subprocess
_elementtree        calendar            mailcap             sunau
_functools          cgi                 marshal             symbol
_hashlib            cgitb               math                symtable
_heapq              chunk               mimetypes           sys
_imp                cmath               mmap                sysconfig
_io                 cmd                 modulefinder        tabnanny
_json               code                msilib              tarfile
_locale             codecs              msvcrt              telnetlib
_lsprof             codeop              multiprocessing     tempfile
```

The import statement as shown below is used for calling the in-built modules, such as an example of system module (sys) in this case

```
import sys
```

Likewise, the resulting search path where all these modules are being located can be assembled altogether from their respective sources of location as:

- The directory that was used for the input script or the actual directory whether the interpreter is interactively running
- The directory list in the environment variable PYTHONPATH, if it has been set. (PYTHONPATH's format depends on the OS but should be imitated as variable PATH environment.)
- An installation-based directory list set up on the installation of Python

Below python code (path_mod.py) lists the resulting search path corresponding to the sys module as:

path_mod.py

```
import sys #importing in-built module

print(sys.path)
```

The output on the execution of the code can be found as:

```
Output:

['', 'C:\\Users\\DELL\\AppData\\Local\\Programs\\Python\\Python38\\python38.zip',
'C:\\Users\\DELL\\AppData\\Local\\Programs\\Python\\Python38\\DLLs',
'C:\\Users\\DELL\\AppData\\Local\\Programs\\Python\\Python38\\lib',
'C:\\Users\\DELL\\AppData\\Local\\Programs\\Python\\Python38',
'C:\\Users\\DELL\\AppData\\Local\\Programs\\Python\\Python38\\lib\\site-packages']
```

5.5.2 User-Defined Modules

For implementing the case of self-written modules, the file (pmod.py) with extension .py needs to be created containing string (str), list of earnings (ear), self-defined method and class (with no specific operation)

pmod.py

```
#initialising string
str = “Ram is a student and does a part time job to meet his earnings”

#initialising list
l_ear = [1000,2000,3000]

#self-defined method
def savings(val):
    print(f“val={val}“)

class Test:
      pass
```

If mod.py is saved in a suitable location that you can eventually understand about importing the self-written modules by creating another file (mod_test.py) at the same location of the previous one.

mod_test.py

```
import pmod

print(pmod.str)

print(pmod.l_ear)

print(savings([,,10000“,“20000“,“24000“]))

x= pmod.Test()

print(x)
```

Once a module is imported, the location of its existence can be determined using the __file__ attribute as:

mod_exs.py

```
import re

print(re._file_)
```

The output on the execution of the code can be found as:

```
Output:

'C:\\Users\\DELL\\AppData\\Local\\Programs\\Python\\Python38\\lib\\re.py'
```

5.5.3 Alternative form of Import Statement

An alternative type of the import statement may be imported directly to the caller's symbol table by individual artefacts of the module with syntax as:

```
from <module-name> import <name(s)>
```

In such cases for the implementation of the same we can invoke the file pmod.py (used in 1.5.2) as a module and create another python code file (anthr_import.py) for implementing the various methods of using such an alternative form of import statement as:

anthr_import.py

```
from pmod import str, savings #module as defined in section 1.5.2

print(str)

print(savings ([„10000“,“20000“,“24000“]))
```

Even the class being defined in the self-written module can be imported using this alternative form of import statement as:

anthr_import.py

```
from pmod import Test #module as defined in section 1.5.2

x=Test()

print(x)
```

Furthermore, if user wishes to import the name of all objects being used in the specific module, then he/she can opt for using * operation after import keyword in the aforementioned alternative form of import statement as:

anthr_import.py

```
from pmod import * #module as defined in section 1.5.2

print(str)

print(savings ([„10000“,“20000“,“24000“]))

x=Test()

print(x)
```

It is also possible to import the name of any module using user-defined alternative name. This is done with an objective of avoiding any conflicts with previously defined names. The import statement in such cases can be represented as:

```
from <module-name> import <name> as <alt-name>[, <name> as <alt-name> ….]
```

The entire module can also be imported using alternative name such that:

alt_mod.py

```
import pmod as ud_module

print(ud_module.str)

print(ud_module.savings ([„10000“,“20000“,“24000“]))
```

5.5.4 Packages in Python

Essentially, a package is like a directory containing subpacks and modules. We can also use one of the Python Package Index (PyPI) for our own projects when creating our own packages. Suppose below is the directory structure for the user-defined modules or subpacks to be used

```
cricket
|-- batsman
|  |-- run.py
|   |--__init__.py
|   |-- matches.py
| `-- mom.py
|-- bowler
|  |-- wickets.py
|   |--__init__.py
|  |-- average.py
| `-- economy.py
```

A package in python must include the_init .py file, though this file may be an empty file. However, only the immediate components are shipped while we import a package, not the sub packages. It will raise an AttributeError if you want to access them.

So as an example, we type the following code to import a package:

package_exmp1.py

```
import cricket

print(cricket)
print(cricket.batsman)
```

5.5.5 Python Packages vs Python Modules

Now that all modules and packages have been revamped, let's see how differing they are:

- A module is a Python-coded format. However, a package is like a directory containing subpackages and modules.
- The__init__.py file must be held by a package. This is not true of modules.

- We use the wildcard * to import anything from a module. But in packages, such wildcard doesn't work.

5.5.6 Installing Python packages (pip-PyPi)

We learn how to use a pip to install and handle packages for Python in this section. Pip is the default Python package manager. In the Python Standard Library, we can load additional packages with pip.

On Python versions 3.4 or higher, pip comes pre-installed. However, if there are two versions of python in the system, then there might be chance that there are two pips – one pip corresponding to Python2 version (pip) and another pip corresponding to Python3 version. However, python2 is near to the deprecation in near future such that pip alone refers to third version of Python i.e., Python3. Likewise, the following command in the console may be used to check for the existence of pip:

```
>>> pip –version
```

Pip is a programme on the command line. A pip command will be added to be used with the command prompt after launch. The standard pip syntax is:

```
pip <pip-arguments>
```

Various arguments linking to the pip command can be implemented where the examples for such cases from installation to usage to removal of any package are defined in the block below:

```
pip install pandas #command to install pandas package
```

User can install the specified version of the package as:

```
pip install pandas==1.2.4 #installing specified version of pandas

Successfully installed pandas-1.2.4
```

For rechecking purposes, the user can run for the same command as during the installation of any package as:

```
pip install pandas

Requirement already satisfied
```

To uninstall a package with PIP, enter the following command in the prompt (don't forget to enter this command with a path of Python Scripts):

```
pip uninstall pandas

Successfully uninstalled pandas-1.2.4
```

In the era of data science, one is in hurry where user doesn‟t want to enter manually for every single package but wants automation in such process. Therefore, pip allows the use of the requirements file which contains all the name of python packages to be installed. For example, let us consider the requirements.txt file containing the name of packages to be installed in the system

requirements.txt

```
librosa
pandas
keras
tensorflow
```

Now user can invoke pip command and install all the packages and its corresponding dependencies using single command

```
pip install -r requirements.txt
```

Likewise, one can search for the installed packages such that all packages containing the name or similar identity as:

```
pip search pygame
```

UNIT VI: ERRORS AND EXCEPTION HANDLING

STRUCTURE

6.0 Objectives

6.1 Introduction

- **6.1.1 Syntax Errors**
- **6.1.2 Exception**

6.2 Built-in Exceptions

6.3 Raising Exceptions

- **6.3.1 raise statement**
- **6.3.2 assert statement**

6.4 Handling Exceptions

6.5 Using Pylint in Python

- **6.5.1 Detecting Multi-statement with implicit continuation**
- **6.5.2 Using Operators**
- **6.5.3 Whitespaces following the use of comma, semicolon, or colon**

6.0 OBJECTIVES

Sometimes the programme does not work at all while running a Python programme or the programme runs but produces unexpected output or is strangely behavioral. This is when one of the syntax or runtimes or logical errors arise in the code. Exceptions in Python are errors which are automatically triggered. Exceptions can, however, be strongly triggered and managed by computer code. We will be studying in this chapter about Python's exceptions such that the error can be managed and caught easily rather than debugging the whole code again and again.

6.1 INTRODUCTION

The reason for an exception is typically outside of the programme itself. Incorrect input, an improper IO device etc. are examples for the origination of such exceptions. Since the programme ends suddenly with an exception, the system resources, e.g. files, might be compromised. The exceptions should thus be properly dealt with so that the application is not abruptly shut down.

It is crucial to understand before we grasp why exception handling and forms of built-in exceptions supported by Python is needed to understand that an error and an exception are different. Therefore, let us make an effort to grasp what errors are in Python before explaining how to deal with problems. Errors are merely code errors that can be damaging leading to loss of useful information/content available in the files/systems.

In Python, there are two kinds of errors:

6.1.1 Syntax Errors

If we have not followed the rules for the programming language when developing a programme, syntax errors are identified. These errors are also called parsing errors. The interpreter does not run the programme on a syntax error unless the mistakes have been corrected, the programme is saved and re-started. If during shell mode, a syntax error occurs, Python shows the name of the error and a little explanation of the mistake.

Errors cannot be managed whereas exceptions to Python may be dealt with at run time. An error may be a (parse) syntax error, but many kinds of exceptions may happen during performance and are not unreservedly inoperative. An error can point to significant flaws that should not be detected by a sensible programme, whereas an exception might identify circumstances for an application to attempt to capture. Errors are a sort of uncontrolled exception, and they can be irretrievably handled by a programmer like an OutOfMemoryError.

6.1.2 Exceptions

Even though an expression or statement is syntactically accurate, an error might occur during its implementation. For instance, try to open a non-existent file, zero-divide, etc. Such mistakes can disturb regular programme execution and are known as exemptions.

An exception is an object from Python that is an error. If a mistake occurs during programme execution, an exception is reported. The programmer must deal with this exception so that the programme is not abnormally terminated. Therefore, a programmer may foresee and resolve such

erroneous scenarios in the design of a programme by the inclusion of the proper code for this exception.

6.2 BUILT-IN EXCEPTIONS

In the compiler/interpreter, common exceptions are often defined. These are referred to as built-in exceptions.

The standard library in Python is a comprehensive collection of built-in exceptions which address common faults (exceptions) by giving the specified remedies for these mistakes. In the case of any included exceptions, a relevant exception handler code is called that shows the reason and the exception name raised. The programmer must take suitable measures to deal with it. Some of the common built-in exceptions in Python are discussed in the table below:

Exception	Description
SyntaxError	Raised when a Python syntax error occurs.
ValueError	Raised when an in-built data-type function has the appropriate type of argument, arguments, but incorrect values are utilized corresponding to the argument
IOError	Raised for errors linked to the operating system.
KeyboardInterrupt	Raised, generally by hitting Ctrl+c or Ctrl+z, after the user stops running the programme.
SystemExit	Raised using the in-built function of sys.exit().
ArithmeticError	Base class for all numerical calculation mistakes.
OverflowError	Raised when the maximum limit for a number type exceeds a computation.
FloatingPointError	Raised if a computation of the floating point fails.
ZeroDivisionError	For all numeric types, its value is raised when division or modulo by zero occurs.
AssertionError	If the Assert statement fails, this exception is raised.
EOFError	When the end of the file is reached and there is no input from either the raw_input() or input() function, this exception is raised.
IndexError	Raised if a sequence does not find an index.
NameError	Raised when a local or global name space does not include an identifier.
IndentationError	Raised if not correctly provided indentation.
TypeError	Raised when attempting an operation or function which is invalid for the data type given.

6.3 RAISING EXCEPTIONS

The Python interpreter raises (throws) an exception every time an error is identified in a programme. Exception managers are intended to run when there is a particular exception. Program makers may also use raise and assert statements to forcibly raise exceptions in a

programme. When there is a derogation, no more statement is performed in the current code block. An exception therefore means that the usual flow of the programme is interrupted and that section of the programme is jumped into (exclusion handling code).

6.3.1 Raise Statement

The raise keyword, on the other hand, is used to raise an exception, whereas the try and except blocks are used to handle exceptions. The syntax of raise statement is as:

```
raise [Exception [, args [, trback]]]
```

Here, in the above syntax, Exception is the exception type (e.g., NameError) and the argument is the exception value. The argument is optional; the exception is None if not provided. Otherwise, trback(traceback) is likewise an optional (and often seldom used) input, and the traceback object for the exception is utilized if provided.

Only an exception handler (or a procedure called directly or indirectly by an exception handler) can use raise without any expressions. The identical exception object that the handler got is re-raised by a simple raise command. The handler is terminated, and the exception propagation mechanism continues to look for other handlers that are appropriate. When a handler realizes that it is unable to handle an exception it gets, and the exception should continue to propagate, it is beneficial to use a raise without expressions.

Likewise, a string, a class or an object can be an exception. The classes with an argument that is a class instance are mostly exceptions raised by the Python core. It is relatively straightforward to define new exceptions and may be done as follows:

```
class SalaryRangeError(Exception):
    """Exception raised for errors in the input salary.
    """
    def__init_(self, sal, mes="Salary here is not in (2000, 10000) range"):
        self.sal = sal
        self.mes = mes
        super().__init__(self.mes)


sal = int(input("Enter the amount of salary: "))
if not 2000 < sal < 10000:
    raise SalaryRangeError(sal)
```

The output of the above code is produced as:

```
Output:

Enter the amount of salary: 12000
Traceback (most recent call last):
```

```
  File "<string>", line 13, in <module>
__main_.SalaryRangeError: Salary here is not in (2000, 10000) range
```

6.3.2 Assert Statement

The declaration of *assert* is used in Python to continue the execution if the particular condition is true. If the condition assesses False, then with the supplied error message, the exception AssertionError will be raised with syntax as follow:

```
assert condition [, ErrorMessage]
```

In Python an expression in the programme code is tested using a declaration. If the test result is wrong, the exception is raised. This statement is usually used to validate the correct entry at the beginning of the function or after a function call

```
def discount_offer(pr, dis):
    new_pr = int(pr['price'] * (1.0 - dis))
    assert 0 <= new_pr <= pr['price']
    return new_pr

clothes = {'name': 'Moda Clothes', 'price': 14000}

print(discount_offer(clothes,0.35)) #35% discount

discount_offer(clothes,2.00) #200% discount
```

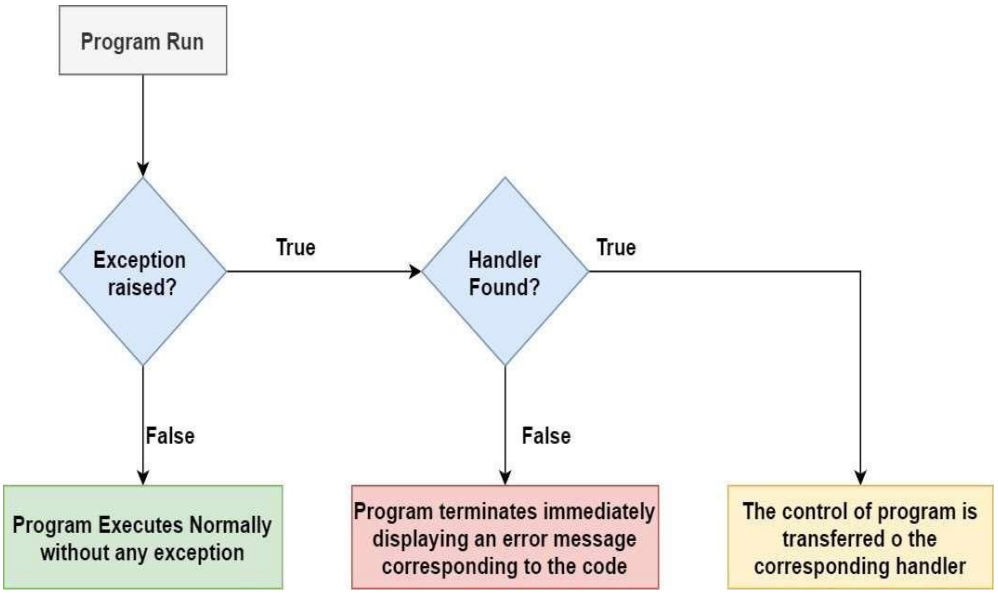
As you can see, attempting an invalid discount generates an exception to AssertionError, which links to the violated assumption. If one of these mistakes is ever encountered during the testing of our online shop, it is easy to find out by looking at the trace with output as shown below:

```
Output:

9100
Traceback (most recent call last):
  File "<string>", line 12, in <module>
File "<string>", line 3, in discount_offer
AssertionError
```

6.4 HANDLING EXCEPTIONS

The flow chart for the process of handling exception in python programming language is as follows:



You may safeguard your programme by placing the suspended code in a block if you have a questionable code which might produce an exception, it is recommended to include a statement after the try: block, followed by an elegant bit of code to deal with the situation.

An exceptional except for a class that is the same class or the same base class (but not the other way round – an exclusively clause listing a derived class is not consistent with a basic class) is compatible with an except for a clause. For instance, in this order, the following code will display B, C, D:

```
class Sports(Exception):
    pass

class Cricket(Sports):
    pass

class News(Cricket):
    pass

for cls in [Sports, Cricket, News]:
    try:
        raise cls()
    except News:
        print("This is news section of cricket")
    except Cricket:
```

```
        print("This is section of cricket game")
    except Sports:
        print("This is section of sports center")
```

On running the above code, the following output is being obtained:

```
Output:

This is section of sports center
This is section of cricket game
This is news section of cricket
```

There may be several test statements except statements for one attempt. The test block contains statements that might produce different kinds of exceptions. This is considered as useful for the developer‟s prospective such that a general excluding clause should cover any exception that may also be provided. Moreover, one can put a different clause after such exception(s). The code in the other block executes if the block tries: no exception is raised. Therefore, the use of other block is a suitable area for code that needs little effort, thus ensuring protection of blocks while coding for bigger enterprise applications. Below is an example of exception handling in file systems:

```
#Exception handling alongside the use of file handling

try:
   f_obj = open("newfile.txt", "w") #opening file in writing mode
   f_obj.write("This is a check for exception handling!!") #writing file
except IOError:
   print "Error: Unable to find file or read data"
else:
   print "Successfully written content into the file"
   f_obj.close()
```

On running the above code, the following output is being produced:

```
Output:

Successfully written content into the file
```

However, exception handling was not useful in the above case as legitimate conditions for opening the file were met as well as adequate. However, when we specify the wrong mode and

then start writing the file, then the try block executes to an exception. The illustration of such case in file handling is shown below:

```
#exception handling alongside the use of file handling

try:
   f_obj = open("newfile.txt", "w") #opening file in writing mode
   f_obj.write("This is a check for exception handling!!") #writing file
except IOError:
   print "Error: Unable to find file or read data"
else:
   print "Successfully written content into the file"
   f_obj.close()
```

On running the above code, the following output is being produced:

```
Output:

Error: Unable to find file or read data
```

The try ... except the statement contains else optional clause, which must follow everything except the provisions when present. It is advantageous if a trial clause does not trigger an exception for code which must be performed. For instance, the below code details the use of exception handling in such scenarios:

```
for i in sys.argv[1:]:
    try:
        f = open(i, 'r')
    except OSError:
        print('cannot open the file', arg)
    else:
        print(i, 'has', len(f.readlines()), ' number of lines')
        f.close()
```

Instead of adding more code to the test clause, the usage of the other clause is better, since it avoids mistakenly catching an exception not produced by the code that is shielded from the test unless statement. If an exception occurs, a value, also known as an argument for the exception, may occur. The type and existence of the argument will depend on the kind of exception. The exception clause can specify an exception name variable. The variable is linked with the argument specified in instance.args to an exception instance.

6.5 USING PYLINT IN PYTHON

The knowledge of the other programmes is one of the main issues of the day. Even worse, situations when there is no informative stuff in the code, such as comments or docstrings. As a programmer, our code should be legible and comprehensible. This is the tale of a Pylint tool that discovered a production-impacting problem the day before the code was deployed. This is also the storey of a tool that, for good reason, no one uses. By the end of this section, one can easily understand why this tool is valuable, why it isn't, and how to utilize it with your Python project.

Pylint is a library for third parties which is not in Python by default and is quite simple to set up. Since Pylint is not part of the standard Python library, we have to separately install it. The pip package can simply accomplish this. Pip is Python's standard Package Manager, enabling packages not included in the Python Standard Library to be installed and managed. Simply run the install command through pip packages, and Pylint and all of its dependencies will be installed.

```
pip install pylint
```

When we have pylint installed, the command pylint with the file name is easy to use by running the following code utilizing pylint package:

```
pylint exp_file.py
```

6.5.1 Detecting Multi-statement with Implicit Continuation

In Python, inside parathesis we utilize implicit line, brackets ([]) and braces ({{}). Implicit indicates that the line continuation character (\) is not written so that a statement is extended over many lines.

The wrapped element should be vertically aligned or hanging by use of the implied continuation lines. Hanging indentation in Python indicates that the start of a parenthesized declaration is the ultimate non-whitespace character of the line, and successive lines are indented until the closing parentheses. Let‟s take an example where we will see handing indentation while defining the function and its corresponding arguments:

test.py

```
#multi-line statement

def mult_func(a1, a2, a3,
  a4, a5):
    """The function performs the numerical operations on numbers"""
    return a1 * a2 - a3 * a4 + a5
```

```
mult_func(2,4,1,6,5)
```

On running the above code using the output comes out to be

```
> python3 mult_pylint.py

7
```

The code will run but as said with pylint one can discover a production-impacting problem such that the readability of code for cross-platform deployments can be tested as:

```
> pylint test.py

************** Module test
test.py:1:0: C0114: Missing module docstring (missing-module-docstring)
test.py:2:0: C0103: Argument name "a1" doesn't conform to snake_case naming style (invalid-name)
test.py:2:0: C0103: Argument name "a2" doesn't conform to snake_case naming style (invalid-name)
test.py:2:0: C0103: Argument name "a3" doesn't conform to snake_case naming style (invalid-name)
test.py:2:0: C0103: Argument name "a4" doesn't conform to snake_case naming style (invalid-name)
test.py:2:0: C0103: Argument name "a5" doesn't conform to snake_case naming style (invalid-name)
test.py:2:0: C0116: Missing function or method docstring (missing-function-docstring)

- - - - - - - - - - - - - - - - - - - - - - - -
Your code has been rated at -13.33/10
```

6.5.2 Using Operators

A warning at the beginning of the code may be easily deactivated by inserting a statement (#pylint; disable=C0114)

The warning C0114 shows the absence of a module docstring. A docstring is a string in module, function, class, or method declaration which is the initial statement. Under PEP 257, all modules should include a docstring stating what the module performs in the starting point. We will not put a docstring to the top of each module in order to make things simpler. However, the practise of writing doctrines is strongly advised.

Below code shows the use of operators where python recommends the use employing single space on either side:

test1.py

```
# pylint: disable=C0114

#Comparison operators
print(9<8)

# Membership operator
if 5 in [1, 2, 3, 4, 5]:
    print('element is present in this list')
else:
    print('element is not present in this list')
```

```
> python3 test1.py
False
element is present in this list

> pylint test1.py
************* Module test1
test1.py:11:0: C0305: Trailing newlines (trailing-newlines)

- - - - - - - - - - - - - - - - - - - - - - -
Your code has been rated at 7.50/10
```

6.5.3 Whitespaces following the use of comma, semicolon, or colon

The following problematic practices can be observed in the code:

- After the comma separating each item in the list, a whitespace is lacking.
- Following the colon (:), which divides the key value pair in the dictionary, there is no whitespace.
- There is a whitespace just before the comma separating each tuple element.

test2.py

```
# pylint: disable=C0103
# pylint: disable=C0114

# list
num=[1,2,3,4,5] #trailing whitespace at the end
```

```
# dictionary - players and grades
score_grade=  {'Rohit':10,  'Virat':2.5, 'Pant':8.5} #trailing whitespace at the end

# tuple - strike rate and average
perf = (128.5 , 51.576) #trailing whitespace at the end

print(score_grade, end=",") #trailing whitespace at the end
print(perf) #trailing whitespace at the end
```

```
>python3 test2.py
{'Rohit': 10, 'Virat': 2.5, 'Pant': 8.5},(128.5, 51.576)

> pylint test2.py

********** Module test2
test2.py:5:15: C0303: Trailing whitespace (trailing-whitespace)
test2.py:8:54: C0303: Trailing whitespace (trailing-whitespace)
test2.py:11:25: C0303: Trailing whitespace (trailing-whitespace)
test2.py:13:27: C0303: Trailing whitespace (trailing-whitespace)
test2.py:14:11: C0303: Trailing whitespace (trailing-whitespace)

----------------------------------------------
Your code has been rated at 0.00/10
```

PYTHON PROGRAMMING

UNIT VII: PYTHON GENERATORS

STRUCTURE

7.0 Objectives

7.1 Generators in Python Programming

7.1.1 Using yield keyword in Python

7.1.2 Using generator in Python

7.1.3 Difference of normal function and generator function

7.2 Using yield from in Python Generator

7.3 Real-life use cases of Python

7.4 Making an iterable from a generator

7.5 Recursive Generator

7.6 Generator Expressions

7.7 Summary

7.0 OBJECTIVES

In this chapter, we study the importance of effective utilization of large sets of results files without the allocation of the memory for all the outcomes simultaneously. Likewise, this can be achieved in Python through providing our own iterator method by employing a generator. A generator is a specific function type that does not return a single value but rather returns a stream of values for an iterator object or in instances when the generator utilizes or consumes another generator, and when it is done as early as feasible, it is more convenient. A yield statement is used instead of a return statement in a generator function. A basic generating feature and functions are further detailed in this chapter

7.1 Generators in Python Programming

Have you ever had to read big datasets or files in a circumstance that was too overwhelming to put into memory? Or perhaps you intended to make an iterator but the manufacture was simple enough to create the iterator, as opposed to creating the needed values?

Keeping in mind the growing outrage of data science in normal life scenarios, the effective use of generator under some of these situations can be quite helpful and simple. The prospects of positivity are generalized on both the sides whether the person is developer or the reader of beautifully written code.

The generator functions introduced with PEP 255 are a specific type of function which returns some form of lazy iterator. Objects can be looped over like a list, although lazy iterators do not save its contents in the memory, unlike lists. The quantity of code required for code is one of the advantages of employing generator functions for using iterators.

7.1.1 Using yield keyword in Python

The keyword yield functions as similar to that of yield in python where the only difference is that it returns a generator object to the caller instead of returning a value.

The function execution stops at the line itself when a function is called and the running thread discovers an output keyword in the function, and returns a generator object to the caller.

If the statement begins an iteration over a collection of items, the generator is running. When the function code of the generator reaches a statement "render," the generator returns its execution to the loop, returning a new value from the set. The generator function can produce as many (maybe unlimited) values as it wishes, each of which in turn results.

Let's see some instances of generators and yield in action after this introduction:

test_yield.py

```
def yield_func():
    yield “This is the way of using yield similar to that of return function”

print(yield_func())
```

Above was the basic syntax covering yield keyword where on running the program, the corresponding output obtained is as following:

Output:

```
<generator object yield_func at 0x7fb3f0ea6510>
```

The essential points considering the usage of yield are as follows:

- A yield produces a function exit but we start 'where we have left off' next time the function is called, i.e., on the line after the output rather than at the starting of the function.
- All local variables values that existed at the time of the yield action are kept intact at the time of resuming.
- The same generator can have many yield lines.
- There are also return statements, however if a StopIteration exception is generated if the next() function is called once again, the execution of a such statement will occur.
- Returns one argument to yield (or none). That can be a tuple parameter, however.

7.1.2 Using Generator in Python

As of now, we are very familiar with syntax employing yield keyword in python. Therefore, the next task is to fetch the corresponding values from generator object. One has to remember that these generator objects are able to be fetched one at a time instead of the whole list together. So for carrying out operations where the whole list is required to be fetched, we can use loop, next() or preferably the list() method.

Following are the examples of the generators corresponding to their fetched values in each scenario

test_generator.py

```
def generator_func():
    yield “first statement”
    yield “second statement”
```

```
    yield “third statement”
    yield “fourth statement”
    yield “fifth statement”

gen = generator_func()

print(gen)

 for i in gen:
     print(i)
```

On running the above program, one can clearly see the use of for loop to get the values stored at particular address corresponding to that generator. Thus, the output of the above code is as:

```
Output:

<generator object generator_func at 0x7f2fdb56d510>
first statement
second statement
third statement
fourth statement
fifth statement
```

7.1.3 Difference of normal function and generator function

The use of generator function sounds similar to that of normal function employed in python programming. Therefore, with an example below, we try to investigate the difference between the two such that the sample code ought to return only the value back which is the form of string.

```
#Generator Function

def gen_func():
     yield “This is the way of using yield in generator function”

#Normal Function

def nor_func():
    return “This is the way of using return in normal function”

print(gen_func()) #calling generator function

print(nor_func()) #calling normal function
```

On running the program, the output will clearly define the difference of using yield and return statement such that the yield keyword corresponds to the address instead normal function returns the string.

```
Output:

<generator object gen_func at 0x7f2fdb56d510>
This is the way of using return in normal function
```

7.2 USING YIELD FROM IN PYTHON GENERATOR

Let's first get out of the way one item. The rationale that the yield of *val* equals *for val in g* is because yield v is not even eq-ual to what yield is. Well let's face it, if the output of the entire loop expands, then the adding of the output of the language does not merit the addition of all new features in a Python 2.x.

What yield from keyword used in generator does is that it sets up a transparent two-way link between the caller and the sub-generator such that:

- The link is "transparent" in that it also spreads everything appropriately, not just the pieces that are created (e.g. exceptions are propagated).
- The link is "bidirectional" because data may be transmitted from and to a generator.

Let‟s illustrate an example where does the actual role of yield from keywords originate and how will it help in solving the rationale problem equivalent to that of for loop. In the below code, we designate the use of manual iteration over read_value() function as:

```
def read_value():
    for i in range(4):
        yield '<< %s' % i

def read_wrap(g):
    # Manually iterate over data produced by reader
    for v in g:
        yield v

wrap = read_wrap(read_value())
for i in wrap:
    print(i)
```

Likewise, what recommened here is to use yield from keyword in read_value() function instead of iterating manually over the function. Therefore, below changes in the code clearly depicts the role of yield from keyword and moreover, the readability of code increases through elimination of one line of code as:

```
def read_value():
    yield from g

def read_wrap(g):
    # Manually iterate over data produced by reader
    for v in g:
        yield v

wrap = read_wrapper(read_value())
for i in wrap:
    print(i)
```

7.3 REAL-LIFE USE CASES OF PYTHON

Generators often work with large files or data streams, such as CSV files considering real-life applications. For implementing such scenario of handling larger files, the code assumes how many rows on a text file we have to count for which the python program may look like:

csv_read.py

```
def read_file (name):
    f = open(name)
    res = f.read().split("\n")
    return res

#csv_reader for reading large text file
gen_csv = read_file("largefile.txt")

#intializing count of the rows
count = 0

for i in gen_csv:
    count += 1

print(f"The number of rows in document are {count}")
```

The above code will probably work on any modern machine if the file contains a few thousand lines, but if the file is large enough, then we will have a few problems. The issues can start to slow down from the machine, until the programme kills the machine, so that the programme must be terminated, to the end.

We supplied large number of files with thousand number of rows and we had to stop the code manually. So due to long-time processing the code eventually on manual trigger resulted in:

```
Output:

Traceback (most recent call last):
Memory Error
```

So we made use of generator in such scenarios and modified the above written code csv_read.py as:

csv_read_gen.py

```
def read_file (name):
    for i in open(file_name, “r”):
        yield i

#csv_reader for reading large text file
gen_csv = read_file("largefile.txt")

#intializing count of the rows
count = 0

for i in gen_csv:
    count += 1

print(f"The number of rows in the given file are {count}")
```

So, after using the yield in the given function read_file(name) in csv_read_gen.py file, the output of the code yielded as:

```
Output:

The number of rows in the given file are 55182343
```

But that's not the end of a tale, there's even easier and more fascinating ways of implementing a generator expression (also known as a generator comprehension) which has a syntax that makes it look like list comprehension.

```
gen_csv = (i for i in open("largefile.txt"))
```

7.4 MAKING AN ITERABLE FROM A GENERATOR

Even although the object-oriented technique to create an iterator is really fascinating, it is not a computationally efficient approach. A generator function is the most common and easiest way to build a iterator in Python. So in order to discuss such a method will implement iterator through generator i.e. an effort will be made to make an iterable from generator

As we had had already studied for the use of for loop while getting the value of generator object, the next() function can also be employed to get the value in this regard. When a generator function is invoked, a generator object is returned without even starting the function. When the next method is initially invoked, the function begins to execute until the return statement is reached. The value received is returned by the next call.

```
#Generator Function

def gen_func():
      yield "This is the way of using yield in generator function"

print(next(gen_func))
```

On running the above program, the next() makes the generator function to return the value instead of the address with output as shown below:

```
Output:

This is the way of using yield in generator function
```

First of all, we would be looking for the process where we will implement an iterator as a Class. An iterable is an object in Python that defines an iterator or an index (index) using_init__ or the __next__ method. In brief, every object that can provide us an iterator may be iterable. What's an iterator then? Such a way can be used to cycle over an iterable object forever through the implementation as:

```
class Sports(str):

      def__init_(self, itrb):
          self.itrb = itrb
```

```
            self.obj_iter = iter(iter)

        def iter (self):
            return self

        def next (self):
            while True:
             try:
                  obj_next = next(self.obj_iter)
                  return obj_next
              except StopIteration:
                   self.obj_iter = iter(self.itrb)

obj=Sports("Cricket")

print(obj)

for i in range(20):
      print(next(obj), end = ", ")
```

7.5 RECURSIVE GENERATOR

We know in Python that one function can call another function can call yet at the same time, the function can even be called by itself. These building types are called recursive functions.

Now you need to ask if recursion in Python generators may be used?

The below is the code demonstrating the use of recursive generator function. The code aims to print the even numbers in the series till 20 *(num<20)* through recursive calling of generator function *even_num(arg)*as detailed below:

recur_gen.py

```
#using recursive generator function

def even_num(begin):
   yield begin
   yield from even_num(begin+2)

#using loop for printing even numbers till 20 from 1
for num in even_num(2):
   if num<20:
     print (num)
```

```
  else:
    break
```

```
Output:

2
4
6
8
10
12
14
16
18
```

7.6 GENERATOR EXPRESSIONS

Using generator expressions, simple generators can be created on fly easily. It facilitates the construction of generators. Similar to lambda functions, generator expressions create anonymous generator functions.

The syntax is similar to the Python list comprehension for the generator expression. But round brackets replace the square brackets. The biggest difference between a list understanding and a generator statement is that a list understanding produces the entire list and the generator statement produces one item at a time. Below is example of it:

```
iterator = ('Iterator' for i in range(5))
```

The above-mentioned generator expression produces the sequence of values that we developed in my generator lesson when it was iterated. Again, here, your memory is refreshed:

```
def repeating_val(val, count):
    for i in range(count):
        yield val

iterator = repeating_val(„Iterator“, 5)
```

Generator Expressions vs List Comprehensions

As you might say, generator terms are pretty comparable to list comprehensions:

```
list_comp = ['Iterator' for i in range(5)]

gen_expr = ('Iterator' for i in range(5))

print(list_comp)

print(gen_expr)
```

However, generator expressions do not build list objects, as opposed to list understandings. Rather, they create "on time" data, such as a class-based iterator or generator function. All you obtain is an iterable "generator object" by assigning a generator expression to a variable with output of above program as:

```
Output:

['Iterator', 'Iterator', 'Iterator', 'Iterator', 'Iterator']
<generator object <genexpr> at 0x7f2af112b580>
```

You must call next() in the same way as you want in any other iterator to get the values generated by the generator expression:

```
gen_expr = ('Iterator' for i in range(5))

print(next(gen_expr))
print(next(gen_expr))
print(next(gen_expr))
print(next(gen_expr))
print(next(gen_expr))
```

```
Output:

Iterator
Iterator
Iterator
Iterator
Iterator
```

Alternately, in a generator expression, you can also invoke the list() method to build a list object that contains all the values generated:

```
gen_expr = ('Iterator' for i in range(5))

print(list(gen_expr))
```

```
Output:
['Iterator', 'Iterator', 'Iterator', 'Iterator', 'Iterator']
```

In-line Generator Expressions

As generator expressions are, well... expressions, you may utilize them in-line with additional statements. You may for example define and use an iterator with a for-loop immediately:

```
for x in („Iterator“ for i in range(5)):
    print(x)
```

```
Output:

Iterator
Iterator
Iterator
Iterator
Iterator
```

7.7 SUMMARY

Generators allow you, in a pythonic way, to create iterators. Iterators only generate the next element of a requested iterable object, and allow lazy evaluation. For really big data sets, this is beneficial. Only over one time, Iterators and generators can be iterated over the inputs. It's better than iterators to employ generator functions. The expression of generators is better than iterators (for simple cases only). Generator expressions are comparable to list comprehensions. They don't build list objects, though. Generator expressions instead create 'time-only' values such as a class-based iterator or generator function. It cannot be restarted or reused after a generating expression is spent. For implementing basic adhoc iterators, generator expressions are optimal. It is best to create a generator or a class-based iterator for complicated iterators.

PYTHON PROGRAMMING

UNIT VIII: FILE HANDLING

STRUCTURE

8.0 Objectives

8.1 Introduction to File Handling

8.2 Types of Files and Formats

8.2.1 Text Files

8.2.2 Binary Files

8.3 Opening Files in Python

8.4 Modes of opening file in Python

8.5 File Positioning

8.6 Closing File in Python

8.7 Creating and appending text file in python

8.8 File Methods in Python

8.9 Working with response data files

8.9.1 Working with CSV file

8.9.2 Working with XML file

8.9.3 Working with JSON file

8.0 OBJECTIVES

Python offers us a key functionality to read file data and write data to a file. So, in this chapter, the objective is to study the important aspects of using various kinds of files and their methods. However, all values or data in programming languages are saved in certain volatile variables. Because data is only saved in such variables during runtime and is lost once the running of the program is over. It is therefore best to save these data with files permanently.

8.1 Introduction to File Handling

So far, we've written Python programmes that receive input, alter it, and show the results. However, that output is only available while the application is running, and input must be supplied using the keyboard. This is due to the fact that the variables used in a programme have a lifespan that lasts until the programme is executed. What if we wanted to save the data that was entered as well as the generated output indefinitely so that we could utilise it again later? Typically, businesses would wish to save information on personnel, inventory, sales, and other items indefinitely to prevent having to enter the same information over and over again. As a result, data is permanently preserved on secondary storage devices for reusability. With a .py extension, we save Python programmes produced in script mode. Each programme is saved as a file on the secondary device. Similarly, the data input and the result can be saved to a file indefinitely.

Files are identified locations on disc where associated data is stored. They're used to keep data in a non-volatile memory for a long time (e.g., hard disk). We utilize files for future usage of the data by permanently saving it since Random Access Memory (RAM) is volatile (it loses its contents when the machine is switched off). We must first open a file before we can read from or write to it. When we're finished, it has to be closed so that the file's resources may be released. As a result, a file operation in Python is performed in the following order:

- Opening a file
- Reading or writing
- Closing the file

8.2 TYPES OF FILES AND FORMATS

Every file on a computer is stored as just a series of 0s and 1s, or in binary form, as we all know. As a result, each file is really nothing more than a sequence of bytes saved one after the other. Text files and binary files are the two most common forms of data files. Any text editor can open a text file, which is made up of human readable characters. Binary files, on the other hand, are made up of non-human readable letters and symbols that must be accessed using special tools.

8.2.1 Text Files

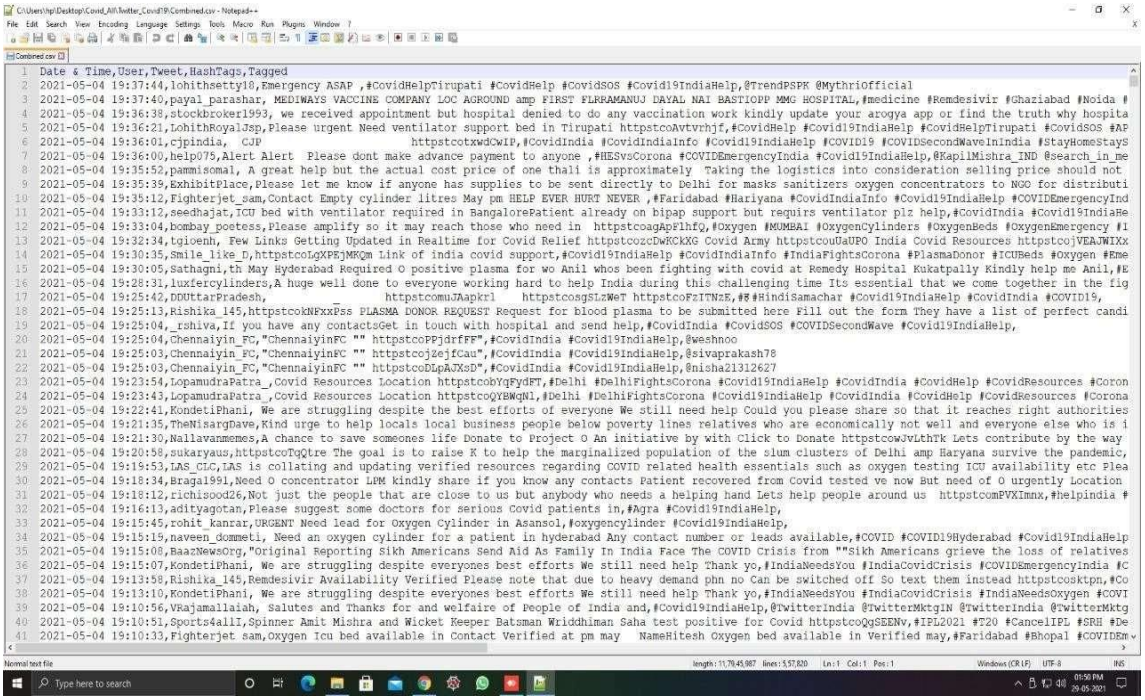
A text file is a sequence of characters that encompasses alphabets, numerals, and other special symbols. Text files include files with extensions such as.txt,.py,.csv, and others. We see many lines of text when we open a text file in a text editor (e.g., Notepad). Internally, however, the file

contents are not saved in this manner. Rather, they are saved as a series of 0s and 1s in a byte sequence. The value of each character in a text file is recorded as bytes in ASCII, UNICODE, or any other encoding system. As a result, when we open a text file, the text editor converts each ASCII value and displays the human-readable comparable character.

Comma Separated Files

A CSV file has some data as a text file. A CSV file is usually used for transferring data across applications. A CSV file holds data, including numbers and text in a simple form, for clarification. The plain text compresses and enables text formatting, as you would recall.

Typically, when there are a bunch of data that is to be transmitted to another programme, an extension of CSV is employed. The file extension, however, assists an operating system to determine which software the file is connected with, in particular. The usage of a spreadsheet application can also better suit the user's needs, as it features cells in which data is arranged in rows and columns.

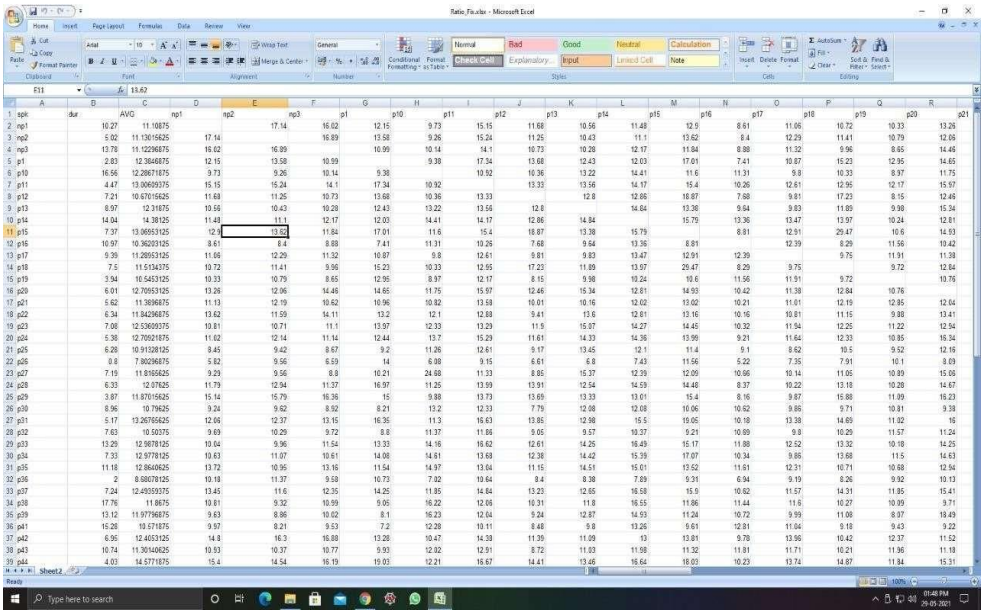


XLSX Files

A Microsoft Excel Open XML Format Spreadsheet file is an XLSX file extension. It's a ZIP Compressed, Microsoft Excel 2007 and later XML-based spreadsheet file.

XLSX files manage data in cells stored in worksheets and saved in workbooks (files that contain multiple worksheets). The cells in a table are positioned according to rows and columns and are capable of including designs, formatting, math and more.

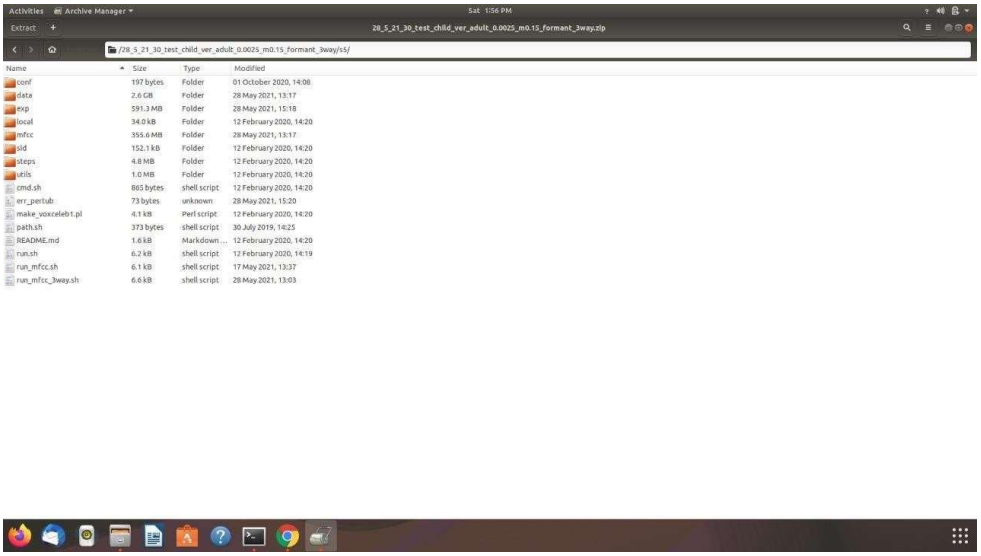
Files produced in previous Excel versions are kept in XLS format. XLSM files are Excel supporting macros.



Zip Files

A zip file is a means to gather many files or archive them in order to function as one file. Let's imagine for instance you would like to send someone with a folder containing Word documents. A zip file is a means to gather many files or archive them in order to function as one file. It would take a long time to attach each file, especially if a large number of papers exist. It might be best to put all the files in a zip file then attach your e-mail to the zip file. Software downloads are the most frequent usage of ZIP files. Zipping software saves the server space, reduces the time required for your computer download and preserves hundreds or thousands of files in a simple ZIP file well structured.

Let's imagine for instance you would like to send someone with a folder containing Word documents. It would take a long time to attach each file, especially if a large number of papers exist. It might be best to put all the files in a zip file then attach your e-mail to the zip file. ZIP file is one of the most frequently used archive formats in which you run, as a ZIP file extension. A file ZIP is merely a collection of one or more files and/or directories, but a single file is compressed for simple transmission and compression like other file formats.

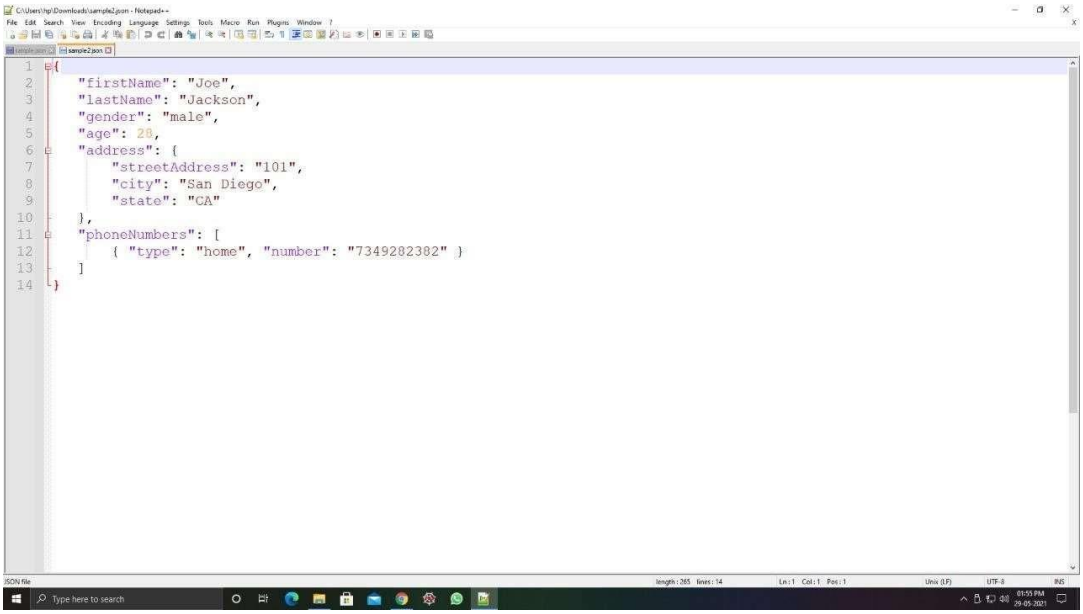


JSON Files

A JSON file is a file that holds basic data structures and its corresponding objects in mostly utilized JavaScript Object Notation (JSON), a standard format for data exchange. It is mostly used for data transmission between a web application and a server. JSON files are small, text-based, and readable by people and can also be changed with a text editor.

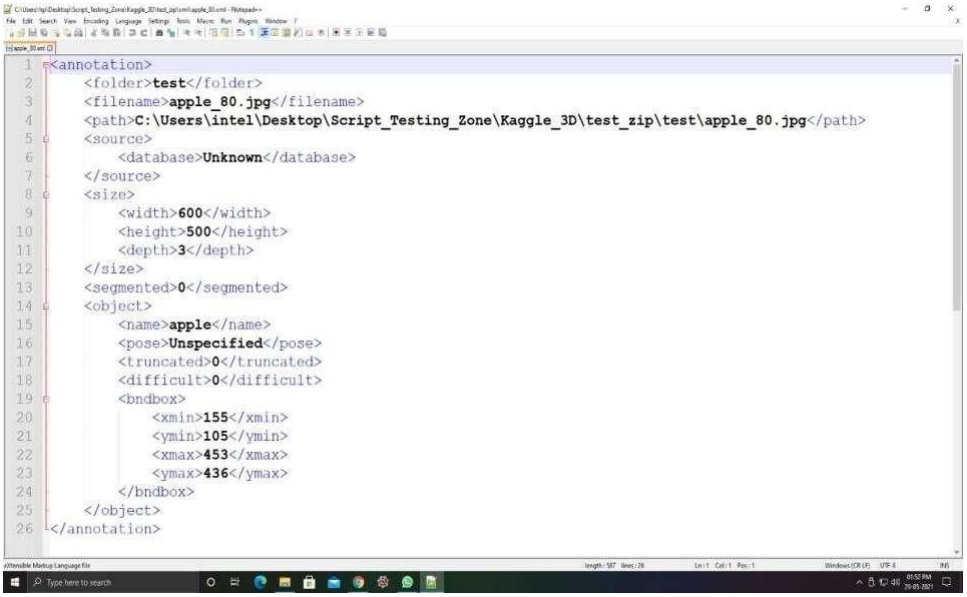
While many apps employ JSON for data exchange, files with the .json cannot effectively be saved on local disk, as there is data exchange between linked machines. But users can save .json files in some programmes. One example is Google+, where JSON files are used to store profile data. You may pick "Data liberation" after login and pick "Download Profile Data."

An XML file for storage and transmission is an extensible marking language file. Tags and text are available in an XML file. The tags supply the data structure. The text in the file you want to keep is enclosed by the tags that comply with certain syntactic requirements. At the heart of an XML file is a conventional text file with specific tags to specify the document's structure, storage and transmission



XML Files

XML is a markup language, which implies that it is a computer language employing tags to specify file components. Instead of writing a syntax this markup language incorporates real words. Likewise, HTML and XML are the most common markup languages.



8.2.2 Binary Files

Binary files, like text files, are contained in bytes (0s and 1s), however these bytes do not reflect the ASCII values of characters. Instead, they represent actual content like images, audio, video, compressed copies of other files, executable files, and so on. These files are not readable by

humans. As a result, using a text editor to access a binary file will result in some junk values. To read or write the contents of a binary file, we'll need specialized software.

Furthermore, binary formats have advantages in terms of access speed. While the underlying unit of information in a plain text file is simple (one byte = one character), locating the actual data values is frequently more difficult. To discover the third data value on the tenth row of a CSV file, for example, the reader programme must continue reading bytes until nine end-of-line characters are detected, followed by two delimiter characters. This implies that in order to discover a certain value in a text file, you must generally read the entire file.

To locate the position (and meaning) of any value in a binary format, some type of format description, or map, is necessary. However, having such a map has the advantage of allowing any value inside the file to be retrieved without bothering to read the entire file.

8.3 OPENING FILES IN PYTHON

To open a file in Python, the built-in open() function is being used. This method returns a file object (commonly known as a handle) that may be used to read or change a file. The syntax of how to open file in python using file object (file_obj) is shown below:

```
file_obj = open(“file”,”mode”)
```

Here file corresponds to the name of the corresponding file that needs to opened or on which any operations need to be performed. Likewise, mode represents the attribute telling the mode in which the actual file needs to be open. When we open a file, we may define the mode. We specify whether we want to read (r), write (w), or append (a) to the file with the mode parameter. We may additionally indicate whether the file should be opened in text or binary form.

For example, one would like to read the contents of the already existing file in your system, then first of all we need to create the existing exmpl.txt as shown below.

exmpl.txt

```
Hello, this is an example of just reading the file using the default mode which is r mode.

To accomplish this, consistent grammar, pronunciation, and more common terms would be required.

When many languages merge, the new language's grammar is more basic and regular than the separate languages.

The new common language will be more straightforward and consistent than current European languages.
```

Now, the task is to read the file such that we assume that the location of the file is similar to that where the python code is being saved, otherwise instead of file name, we need to provide the absolute or relative path in order to open the file. Likewise, for reading the contents inside the file, we opt to use in-built python‟s read() function as shown in file_open.py below:

file_open.py

```
file_obj = open(“exmpl.txt”, “r”)

„„„
or we may also write file_obj=open(“exmpl.text”)
as by default it open() function opens file in read mode
„„„

print(file_obj.read()) #in-built read() function
```

On running the above program, the contents present in the file are displayed as:

Output:

```
Hello, this is an example of just reading the file using the default mode which is r mode.

To accomplish this, consistent grammar, pronunciation, and more common terms would be required.

When many languages merge, the new language's grammar is more basic and regular than the separate languages.

The new common language will be more straightforward and consistent than current European languages.
```

The read() function employed above provides the entire text stored in the file by default, but user can also specify the number of characters he/she wishes to return:

```
file_obj= open(“exmpl.txt”, “r”)

print(file_obj.read(5))
```

The output of the above code will be as:

```
Output:

Hello
```

Likewise, there are other alternative ways of reading the file including using readline() function which returns single line in the file and secondly using for loop through file line by line. Both the functionalities are being represented on the same plain text file () as we had used earlier:

file_readline.py

```
file_obj = open("exmpl.txt","r")

print(file_obj.readline()) #print first line of file
print(file_obj.readline()) #print second line of file
```

```
Output:

Hello, this is an example of just reading the file using the default mode which is r mode.

To accomplish this, consistent grammar, pronunciation, and more common terms would be required.
```

file_loop.py

```
file_obj = open("exmpl.txt","r")

for line in file_obj:
    print(line)
```

```
Output:

Hello, this is an example of just reading the file using the default mode which is r mode.

To accomplish this, consistent grammar, pronunciation, and more common terms would be required.
```

When many languages merge, the new language's grammar is more basic and regular than the separate languages.

The new common language will be more straightforward and consistent than current European languages.

readline() vs readlines() in Python

Let‟s assume the inbuilt text file, listdata.txt, with the contents as shown below:

listdata.txt

```
Rohit
Virat
Rishabh
Hardik
```

readline() reads a file line till the end of that line is reached. In the string is maintained a trailing newline character (\n). Likewise, readlines() in contrast produces a list with all the lines of the file (strings). The syntax is this:

file_readlines.py

```
f ile_obj = open("listdata.txt")

print(file_obj.readlines()) #readlines() return list of files
```

The output of running the file is shown as below:

Output:

```
[„Rohit\n‟,‟‟Virat\n‟,‟‟Rishabh\n‟,‟‟Hardik\n‟]
```

8.4 MODES OF OPENING FILE IN PYTHON

It's a number that indicates the file's opening mode, such as read, write, append, and so on. It's a non-mandatory parameter. It is set to read-only by default (r). After reading from the file, we obtain data in text form in this mode. Below you'll find a table providing a list of the many

access options for the case of opening text file and their corresponding modes. Likewise, the binary mode, on the other hand, returns bytes. It's better for accessing non-text files like images and executable files. Here, you'll find a table. It provides a list of the many access options for the case of opening binary files and their corresponding modes.

Modes	**Description**
r	Opens a file in a reading mode
rb	Opens a binary file in a reading mode
w	Opens a writable file. Create a new file if it doesn't exist or if it already exists, truncate the file.
wb	Opens a writable binary file and also like mode w, it will create a new file if it doesn‟t exist or truncates on the existence
a	Opens a file without truncating at the end of a file. Create a new file if not available.
ab	Opens a binary file without truncating at the end of a file. Create a new binary file if not available.
+	Opens a file for both modes of reading and writing

In case of object-oriented case, open() function when employed in python have several linked attributes which run down as shown in below table.

Attributes	**Description**
files.closed	A Boolean attribute telling if the file is closed or not.
file.mode	Returns the file-opened access mode.
file.name	Returns file name
file.softspace	Returns false if space with print is expressly necessary, else true.

8.5 FILE POSITIONING

We notice that a newline returns as a '\n' in the read() function. When we reach the end of the file, the next time we read, we obtain a blank string. Using the seek() function, we may modify our current cursor file (position) with syntax as:

```
file_obj.seek(offset, pos)
```

where you are dealing with *file_obj* is really the file pointer; *offset* indicates how many places you move; your point of reference is defined by *pos*:

- 0: implies the start of the file is your reference point
- 1: signifies that the current file location is your reference point.
- 2: implies that the end of your file is your reference point

If *pos* argument is missed, the default is 0. Suppose we have a file with contents as shown below

```
filepos.txt

First Line of the file
```

Now, we want to read the character at the suitable position. The example of such implementation is shown below:

```
file_obj=open(“filepos.txt”,”r”)
file_obj.seek(3)
print(file_obj.readline())
```

So we have jumped three bytes over the character such that the output of the above program on running is as:

```
Output:

st Line of the file
```

Likewise, the tell() function also returns our current position (in number of bytes). The example code of the tell() function is as below considering the same file filepos.txt:

```
f_obj = open("filepos.txt", "rw+")
print("File Name: ", f_obj.name)

line = f_obj.read(4)
print(line)

# Get the current position of the file.
pos = f_obj.tell()
print(pos)

# Close opened file
fo.close()
```

8.6 CLOSING FILE IN PYTHON

The file object's closing method i.e. in-built function close() flushes any unwritten information and closes the file object, so it is no longer possible to write. However, when a reference object of a file is reassigned to a different file, Python automatically closes a file. Therefore, it is

believed that the close() function to end a file is always a good practice specially while solving real-life market related problems.

close_exmpl.py

```
file_obj=open("filename.txt","wb")
print("The file name is: ", file_obj.name)

#closing file that was opened

file_obj.close()
```

8.7 CREATING AND APPENDING TEXT FILE IN PYTHON

We have to open it in writing w, add an or exclusive x modes to write a file in Python. We must be careful with w mode, since if it already exists, it is overwritten onto to the file. This causes the deletion of all prior data. The write() function is used to write a byte string or sequence (for binary files). This returns the number of characters in the file. Also in this example, we will study the another way opening file employing *with* keyword as:

```
with open("newfile.txt ","w") as file_obj:
        for lines in range(5):
            print("This is way of writing file with print method", file=file_obj)
            file_obj.write("Rohit\n")

file_obj.close()

with open("newfile.txt","r") as file_read:
        print(file_read.readlines())

file_read.close()
```

In the above code, we have explored the two ways i.e. using print() and write() function of writing a new file. Therefore, the output of the above program is as:

Output:

['This is way of writing file with print method\n', 'Rohit\n', 'This is way of writing file with print method\n', 'Rohit\n', 'This is way of writing file with print method\n', 'Rohit\n', 'This is way of writing file with print method\n', 'Rohit\n', 'This is way of writing file with print method\n', 'Rohit\n']

The file handle position is also defined in these modes. The handle of the file is like a cursor defining from which the information is to be read or written into the file. Open the file as a new line in append mode, with 'a' or 'a+' as an access mode, to add a new line to the existing file as:

```
#creating file in python

f_obj1 = open("newtextfile.txt", "w")
list1 = ["Rohit\n", "Virat \n", "Sharma"]
file1.writelines(L) #adding list in file using writelines() function
file1.close()

# Append-adds at last

f_obj2 = open("newtextfile.txt", "a") # append mode
file1.write("Pant \n") #adding contents at the bottom of file
file1.close()

f_obj3 = open("newtextfile.txt", "r") #reading file for printing
print("Output after appending into the file ", f_obj3.name)
print()
print(f_obj3.read())
print()
f_obj3.close()
```

```
f_obj1 = open("newtextfile.txt", "w") # write mode
f_obj1.write("Hardik \n")
f_obj1.close()

f_obj1 = open("newtextfile.txt", "r")
print("Output after writing the already existing file in python ")
print(f_obj1.read())
print()
f_obj1.close()
```

8.8 FILE METHODS IN PYTHON

The file object contains numerous methods. Most of them are not much utilized in the context of python programming but eventually plays critical role. Therefore, below is the full set of methods with their respective brief description corresponding to the file in python:

Methods	Description
detach()	Separates from the TextIOBase the underlying binary buffer and returns
fileno()	Returns the file descriptor (integer number)
isatty()	Returns True if interactive file stream.
flush()	Flushes the file stream writing buffer.
readable()	Returns true on reading from the stream of the file.
seekable()	Returns True when random access is supported by the file stream.
truncate(size=None)	Resize the stream file to bytes in size. If not given, the size will be resized to the current location
writable()	Returns True if you can write the file stream to.

8.9 WORKING WITH RESPONSE DATA FILES

Whether you develop a thin client or a thick client (client server application), you definitely ask for a Web server and require a proper data format to answer your questions at some point. There are three primary types of data that are currently utilized to deliver data to a client from a web server: CSV, XML, and JSON. It is a good idea to grasp the difference between each format in order to design an application with a robust architecture and know when to utilize it. This post is intended to outline every data format, to explain the advantages and disadvantages for every single format and to find out which conditions work best.

8.9.1 Working with CSV File

You must use the reader feature to construct a reader object to read data from CSV files. The reader function is built to produce a list of all columns for each row of the file. The column for which the variable data is required is then chosen. It sounds much more complex than it is. Let us look at this CSV code in Python, and we will find out that it is not too difficult for us to deal with csv file.

Writing to CSV File

```
import csv #import modules for csv file

with open(„dataset.csv', 'w') as f_obj:
    data_write = csv.writer(f_obj, delimiter=';', quotechar="", quoting =
csv.QUOTE_MINIMAL)

    #way to write to csv file
    data_writerow([„Shirt No“;“Player Name“; „Team“; „Scores“]
    data_writerow([“45“;“Rohit“;“Mumbai“;“99“])
    data_writerow([“18“;“Virat“;“Bangalore“;“23“])
```

```
    data_writerow([„7“;“Dhoni“;“Chennai“;“4“])
```

Reading from CSV File

```
import csv #import modules for csv files

with open('datatest.csv','rt')as f_obj:
  data_test = csv.reader(f_obj)
  for line in data_test:
        print(line)
```

On running the above code, the output is:

```
Output:

[„Shirt No;Player Name; Team; Scores“]
[“45;Rohit;Mumbai;99“]
[“18;Virat;Bangalore;23“]
[„7;Dhoni;Chennai;4“]
```

8.9.2 Working with XML File

We will use the next XML file in the examples below that we save as "players.xml":

```
                                        players.xml
<data>
    <players>
        <player name="rohit">rohit2india</player>
        <player name="virat">virat2india</player>
    </players>
</data>
```

Writing XML file

ElementTree is suitable for write XML file data. The following code illustrates how to generate an XML file with the same structure as the file in earlier instances. The following steps:

- Create an element that acts as our root. The tag for this element is "data" in our example.
- Use the SubElement function to construct sub-elements after we have our root element.

The syntax of this function is:

write_xml.py

```
import xml.etree.ElementTree as etree

# creating the structure of the XML File

data = etree.Element('data')
players = etree.SubElement(data, „players“)
player1 = etree.SubElement(players, „player“)
player2 = etree.SubElement(players, „player“)
player1.set('name',“rohit“)
player2.set('name','virat“)
player1.text = 'rohit2india'
player2.text = 'virat2india'

# create a new XML file with the results

my_data = etree.tostring(data)
my_file = open("players.xml", "w")
myfile.write(mydata)
```

Reading XML File

The *minidom* is a reduced implementation of the document object model (DOM). DOM is a programming interface for application, which handles XML as a tree structure, where each tree node represents an object. Therefore, we must be aware of the capabilities of this module.

```
from xml.dom import minidom  #importing module for xml

# parsing an xml file by name
mydoc = minidom.parse('players.xml')

players = mydoc.getElementsByTagName('player')

# printing for one specific player attribute
print('Player #2 attribute:')
print(players[1].attributes['name'].value)

# all player attributes
print('\nAll attributes:')
for elem in players:
    print(elem.attributes['name'].value)

# printing one specific player's data
print('\Player #2 data:')
```

```
print(players[1].firstChild.data)
print(players[1].childNodes[0].data)

# all players data
print('\nAll players data:')
for elem in players:
    print(elem.firstChild.data)
```

A more "Pythonic" interface to XMl is provided in ElementTree module and is an excellent solution for people not aware of the DOM. It is also probably best for more rookie programmers because of its basic interface, as you will see in this post.

```
import xml.etree.ElementTree as etree
tree = etree.parse('players.xml')
root = tree.getroot()

# printing one specific player attribute
print('Player #2 attribute:')
print(root[0][1].attrib)

# all player attributes
print('\nAll attributes:')
for elem in root:
    for sube in elem:
        print(sube.attrib)

# one specific player‟s data
print('\nPlayer #2 data:')
print(root[0][1].text)

# all player data
print('\nAll player data:')
for elem in root:
    for sube in elem:
        print(sube.text)
```

The output of the both the above programs is as:

```
Output:

Player #2 attribute:
player2

All attributes:
```

```
player1
player2

Player #2 data:
virat2india

All player data:
rohit2india
virat2india
```

8.9.3 Working with JSON File

The JSON format has been one of the common ways, if not the most, to serialize data for the previous 5-10 years. You will probably be encountered with JSON especially in the web development industry through one of the various REST APIs, application settings or simply basic data storage.

Writing JSON file

You may easily send your data to a Python file in JSON format by storing your data in a dict object that may include additional nesting dicts, lists, booleans or other primitive kinds, such as integers or string. A comprehensive list of supported data types may be found here.

```
import json #imporitng json module

data = {}
data[,,player‟] = []
data['player'].append({
    'name': 'Rohit',
    'Team': 'Mumbai',
    'from': 'India'
})
data['player'].append({
    'name': 'Maxwell',
    'Team': 'Bangalore',
    'from': 'Australia'
})
data['player‟].append({
    'name': 'Gayle',
    'Team': 'Punjab',
    'from': 'WestIndies'
})

with open('playerdata.txt', 'w') as out_file:
    json.dump(data, out_file)
```

We create some simple data to publish to our file once we load the json library. The key section ends when we use the statement to open our target file and use json.dump to write the data object to the outfile file. Every file object, even if it is not a real file, can be passed on to the second parameter. The socket that can be opened, closed, and written like a file would be an excellent example. This is another scenario that you can meet with, as JSON is widespread all over the web.

Reading JSON file

On the other hand, it is just as straightforward to read JSON data from a file. We can extract and parse the JSON string from a file object using the same json package again. We do precisely this and then publish the data we have received in this example:

```
import json

with open('playerdata.txt') as json_file:
    data = json.load(json_file)
    for p in data['player']:
        print('Name: ' + p['name'])
        print('Team: ' + p['Team'])
        print('From: ' + p['from'])
        print('')
```

The quality and effective to note here is json.load. It will read the file string, scanning the JSON contents, adding the contents to a Python Dict and returning it.

Pretty-print in JSON file

It is as easy as supplying the integer value on the *indent* option, for JSON human readable (although "pretty printing"):

```
import json
data = {'player':[{'name': 'Rohit', 'Team': 'Mumbai', 'from': 'India'}]}
json.dumps(data, indent=4)
```

Output:

```
{
   "player": [
         "Team": "Mumbai",
         "from": "India",
         "name": "Rohit"
      }
   ]
}
```