2024

Problem Solving Using Python

Dr. Babasaheb Ambedkar Open University

MCA-302 Problem Solving using Python

Expert Committee

Prof. (Dr.) Nilesh Modi Professor and Director, School of Computer Science, Dr. Babasaheb Ambedkar Open University, Ahmedabad	**(Chairman)**
Prof. (Dr.) Ajay Parikh Professor and Head, Department of Computer Science Gujarat Vidyapith, Ahmedabad	**(Member)**
Prof. (Dr.) Satyen Parikh Dean, School of Computer Science and Application Ganpat University, Kherva, Mahesana	**(Member)**
Prof. M. T. Savaliya Associate Professor and Head, Computer Engineering Department Vishwakarma Engineering College, Ahmedabad	**(Member)**
Dr. Himanshu Patel Assistant Professor, School of Computer Science, Dr. Babasaheb Ambedkar Open University, Ahmedabad	**(Member Secretary)**

Course Writer

Dr. Harshal A. Arolkar Professor & Head, FCAIT-GLS University
Dr. Vishal Narvani Assistant Professor, FCAIT-GLS University
Dr. Snehal Shukla Assistant Professor, FCAIT-GLS University
Dr. Rachana Chaudhari Assistant Professor, FCAIT-GLS University

Content Editor

Dr. Shivang M. Patel Associate Professor, School of Computer Science, Dr. Babasaheb Ambedkar Open University, Ahmedabad

Subject Reviewer

Prof. (Dr.) Nilesh Modi Professor and Director, School of Computer Science, Dr. Babasaheb Ambedkar Open University, Ahmedabad

ISBN- 978-81-982671-5-3

Printed and published by: Dr. Babasaheb Ambedkar Open University, Ahmedabad

Dr. Babasaheb Ambedkar Open University

MCA-302

Problem Solving Using Python

Block-1

Fundamentals of Python

Unit-1: Getting Started with Python

1

Unit Structure

1.0 LEARNING OBJECTIVE

After studying this unit student should be able to:

- What is Python and requirement to work with Python
- Understand how to write python programs
- Proper indentation required for program
- Basic understanding of code

1.1 INTRODUCTION

Python is a very popular, object-oriented and interactive language that can be used in various fields like data analysis, computing, real time data processing etc. It was developed by Guido van Rossum during 1985-1990. Python is an open source programming language, code of it is available under GNU General Public License (GPL). Python is a high level language, so it provides a very easy way to write a program. Programs written in python are very easy to read. In other languages like C / C++ we need to write lengthy code to perform small tasks, but using python, it can be done in a few lines of code.

In this chapter we will see what are the basic requirements to work with the Python programming language. Learn how to create a simple python program and understand the requirement of indentation.

1.2 BASIC REQUIREMENT TO WORK WITH PYTHON

Before we can start writing a program in Python, we need to ensure that appropriate software is installed on our computers. The first thing you need to do is to install Python if not already installed. The official Python website (https://www.python.org/) hosts the latest stable versions that can be downloaded and installed. The latest version is Python 3.x. Python runs on all major operating systems, including Windows, MacOS, Linux and other Unix-based systems.

The installation process is very straightforward and can be easily completed. It might differ a bit in different operating systems. The basic steps have been mentioned here for your reference:

- On Windows, download the executable installer in your machine. Once downloaded double click on the installer and follow the instructions. Make sure to check the box that says “Add Python to PATH.”
- On MacOS and Linux, Python is usually pre-installed, but it can be updated using the terminal or installed via a package manager like Homebrew in (Mac) or apt in (Linux).

To check whether Python is installed on your computers one can open the terminal window and type *python3* at the prompt. If Python is installed, we will be taken to a Python prompt similar to the one shown here (it might differ based on the operating system that we are using. The one shown here is from MacOS and Ubuntu Linux):

```
Python 3.8.9 (default, Apr 13 2022, 08:48:07)
[Clang 13.1.6 (clang-1316.0.21.2.5)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

```
Python 3.10.12 (main, Nov  6 2024, 20:22:13) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The >>> is known as Python prompt. As python is an interactive language, we can directly interact with the python terminal. We can type the command on the terminal and the python interpreter will execute it. Let us write our first Python code on the prompt and see its output. Type print("Hello World") at the prompt and then press Enter key. You will see an output similar to one shown in Example 1.1:

Example 1.1: Working on Python prompt

```
>>> print("Hello World")
Hello World
>>>
```

The code in the Example 1.1 has used two concepts of Python; first is called function and second is a string. Both these concepts will be discussed in detail later, for now we need to understand that print() allows us to display contents on our screen. Thus, the contents, *Hello World* are displayed on the screen as soon as we press the Enter key.

It is also possible to perform mathematical operations at the prompt as shown in Example 1.2.

Example 1.2: Performing mathematical operations on Prompt

```
>>> 3 + 5
8
>>> 4 - 6
-2
>>> 9 * 2
18
>>> 7 / 14
0.5
>>>
```

As can be seen in the Example 1.2, we have performed four mathematical operations, addition, subtraction, multiplication and division respectively.

In both the above examples we have worked on Python prompt directly. For proper programming though we need to write a Python program. To write a program we need a text editor or an Integrated Development Environment (IDE). The most popular IDE for Python programming is PyCharm. It is a full featured IDE, great for large projects and it comes with a lot of built-in tools. In this book we will use a notepad available on the system to write a program.

1.3 SIMPLE PYTHON PROGRAM AND INDENTATION

Python programming language is an English like language. It only uses indentation to define blocks within the program. There is no use of curly braces {} or keywords like begin or end to define the blocks. Indentation means adding white space in the starting of statements. Any block like if, while, function starts, colon (:) is inserted and the statements under the block will have whitespace or tab before them. The general rule of indentation says that each new level of indentation is created using 4 spaces.

Python programming language has a large set of data types and supports more structures to accommodate many types of data. As it supports a variety of data structures by default, it is very popular amongst programmers these days. It checks for errors and handles it more efficiently as compared to other programming languages.

Let us write a small Python program to display a message “Hello Word” on the screen. Open a notepad editor of your choice and write the given lines in it as shown in Example 1.3.

Example 1.3: Simple Python program

 # Program to print Hello World *print(“Hello World”)*
OUTPUT: Hello World

Save the file with extension .py, Let us save it as HelloWorld.py. Once the file has been saved we can execute the program using Terminal. Open terminal on your computer, go to the path of the folder where our file is stored and type command ***python3 HelloWorld.py*** and press Enter key. This command will execute our code and generate the output on the terminal. The output of the HelloWorld.py file will be a single line Hello World as seen above.

Let us have a look at another simple Python program given in Example 1.4 that shows the use of indentation.

Example 1.4: Python program showing indentation

```
# Program to show use of indentation
age = 25
if age >= 18:
    print("You can vote.")
else:
    print("You cannot vote.")
```

OUTPUT:

You can vote.

The program shown in Example 1.4 starts with a comment line that defines the purpose of the program. Line 2 defines a variable called age and assigns value 25 to it. The statement *if age >= 18:* checks whether the value stored in age is greater than or equals to 18 or not. Observe the spaces before the statement *print("You can vote.")*

This is what indentation is, it tells the compiler that the current statement is part of the 'if' block. The else statement in the program begins a new block that is executed only when the condition in the if statement is not satisfied (not true). The statement *print("You cannot vote.")* is again indented as it belongs to the else block. Thus, we can say that in this program the statements, *age = 25*, *if age >= 18:* and *else:* have top level indentation.

The output of this program will always be 'You can vote' as the value of age is 25 which is greater than or equals to 18

Note:

- Indentation is used to mark the start and end of code blocks.
- If proper indentation is not used in a Python program we will get an IndentationError because it can't determine which statements belong to which block.

Check Your Progress-1

a) The Python program has extension .pyc. (True/False)
b) It is also possible to perform mathematical operations at the Python prompt. (True/False)
c) The tern IDE refers to Internal Development Environment. (True/False)
d) Indentation is not important in python. (True/False)
e) The print() function can be used to display the content as output on the screen. (True/False)

1.4 BASIC SYNTAX OF PYTHON PROGRAM

A Python program consists of three main components; statements, comments and indentation. Statements are the instructions that Python executes. A statement can be a variable assignment, decision making, a loop or a function call.

Comments are used to make the code more readable and explainable. A single line comment in Python begins with the # symbol. The statement written after # will be ignored by the interpreter. It can be used for understanding code. For multiple line comments we can use triple quotes (''' or """) before and after the comment statements. The following example shows the use of comments.

As mentioned earlier Python uses indentation, i.e. whitespace at the beginning of a line, to define blocks of code, such as decision making, loops and functions. As such a Python program has a free format syntax. The general syntax of using a function in a Python program is as mentioned:

def func():

statement1

statement2

..

..

statement N

New statement out of block

Python uses a newline (\n) character as the end of the statement. When the statements of one block completes, the new statement outside the block will not have any whitespace or tab before it.

Python is a case sensitive language, meaning the same name with different cases will be considered a different name in python. For example, ‘python’ and ‘Python’ both are different.

At times when writing a program, a sentence may become too long. Such a sentence can be converted to multiple lines with the use of backslash (\) character. For example, if we are initializing a variable where the statement is of two lines, so for better readability, we can add backslash (\) to break the line. The Python program given in Example 1.5 shows the use of all the above mentioned features.

Example 1.5: Sample Python program showing its components

```
''' A program to show the basic syntax of a python program. It uses a function to compare two parameters. The function checks whether num1 is greater, lesser or equal to num2 and prints appropriate messages. It also shows use of backslash character'''

#Function compare
def compare(num1, num2):
  if num1 > num2:
      print(f"{num1} is greater.")
  elif num1 < num2:
      print(f"{num2} is greater.")
  else:
      print("Both numbers are equal.")

# Get user input
num1 = int(input("Enter the first number: "))
num2 = int(input("Enter the second number: "))
```

```
# Function call
compare(num1, num2)

# use of backslash
statement="This line has been made too long to accommodate"\
" in a single line to show the use of backslash characters."
print(statement)
```

OUTPUT-SCENARIO 1:

Enter the first number: 5
Enter the second number: 8
8 is greater.
This line has been made too long to accommodate in a single line to show the use of backslash characters.

OUTPUT- SCENARIO 2:

Enter the first number: 6
Enter the second number: 2
6 is greater.
This line has been made too long to accommodate in a single line to show the use of backslash characters.

OUTPUT- SCENARIO 3:

Enter the first number: 5
Enter the second number: 5
Both numbers are equal.
This line has been made too long to accommodate in a single line to show the use of backslash characters.

The Python program shown here is designed to compare two numbers provided by the user. It determines their relationship, whether one is greater, lesser, or if they are equal. The initial four lines are multiline comments that explain the purpose of the program. These lines are ignored by the interpreter. Then we have defined a function named compare(num1, num2), which accepts two parameters, num1 and num2, representing the two numbers to be compared. Inside the function, conditional statements if, elif and else are used to check whether num1 is greater than, less than, or equal to num2. An appropriate

message is printed based on the comparison. The program then prompts the user to input two integers. As all data entered by the user through the keyboard in Python is in the form of text the int() function is used to convert the text into integer. Next, the compare() function is called by passing the numbers entered by the user, and the result is displayed to the user as can be seen in OUTPUT-SCENARIO 1, OUTPUT- SCENARIO 2 and OUTPUT- SCENARIO 3. The last process that this program does is to assign a multiline value to a variable *statement* and print it on the screen. Observe that in all three outputs the statement has been displayed as a single line.

1.5 STEPS INVOLVED IN PYTHON PROGRAM EXECUTION

Let's understand how python programs execute. Python is an interpreted language. Interpreter reads a single line from code and translates it and executes it at the same time. Python Interpreter works in the same way. It will read one by one lines from the program and execute it and generate the output.

The steps involved in the execution of Python program are shown herewith:

Step 1: The first step is to write the Python program using a text editor. (You can use an integrated development environment (IDE) also).

Step 2: Save the program with a .py extension.

Step 3: Run the program using the **python3** command on the terminal. When the program is run, the Python interpreter first performs lexical analysis. Here the source code is broken into tokens that represent the smallest units of meaning, such as keywords, operators, and identifiers.

Step 4: The tokens are then parsed to generate an Abstract Syntax Tree (AST) that represents the program's syntax and ensures that the code is grammatically correct. If the program is not grammatically correct we will get an error and the program will stop.

Step 5: Once the code is parsed, Python compiles the AST into bytecode. Bytecode is a lower-level, platform-independent representation of the source code.

Step 6: The bytecode is sent then to the Python Virtual Machine, which is responsible for interpreting and executing the bytecode on the system's hardware. The PVM converts the bytecode into machine code that the operating system can execute.

Step 7: The PVM executes the program line by line. It performs tasks such as mathematical operations, function calls, and I/O operations.

Step 8: After execution, the program may produce output, such as displaying results to the console, writing to files, or any other operation mentioned by the user.

Though eight steps have been mentioned here the user generally does not get aware about internal details of step 3 to step 7.

Check Your Progress-2

a) To create a block in a Python program, we need to enter colon (:) at the end of the statement. (True/False)

b) PVM executes the program line by line. (True/False)

c) Python is a compiled language. (True/False)

d) A new line character is used to write a string in multi-line. (True/False)

e) A multiline comment is enclosed within ''' and '''. (True/False)

1.6 LET US SUM UP

In this chapter we have discussed what Python is and learnt about the basic syntax of Python programs. We have written simple Python code and saw how to execute it. We also learnt the importance and use of indentation, comments and multiple line separators. The last section explained the detailed steps involved in the execution of a Python program.

1.7 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

1-a False
1-b True
1-c False
1-d False
1-e True
2-a True
2-b True
2-c False
2-d False
2-e True

1.8 ASSIGNMENTS

1. What is Python?
2. How can we run the python code?
3. What is meant by case sensitivity in python?
4. How can the comments be added in python code?
5. How can we write one python statement in multiple lines?
6. List steps involved in the execution of a Python program.

Unit-2: Variables and Data Types

2

Unit Structure

2.0 LEARNING OBJECTIVE

After studying this unit student should be able to:

- Understand how to create a variable
- Understand the use of data types with variables
- Understand the use of type casting
- Write programs to understand how variables are used

2.1 INTRODUCTION

After understanding the basic structure of python programming, now it's time to know how we can store the required data and use it in our program. Programming is all about using and processing the data. We always try to perform some operations on the data. So, it is very important to understand how we can access the data in the Python program.

Data that is useful for programming may be in the form of names, numbers or combinations of both the numbers and words. To use data in the program, we have to give the data holder a name. Without giving the name, we cannot access it multiple times in our code.

In this chapter we will learn what a variable is, how to define and use variables in Python programming. We will also learn the concept of data types and look into the data types that are supported in Python.

2.2 VARIABLES

Variables refer to the names of an entity that stores the data value. A variable declaration creates a memory space to store the value of data. We can refer to the data by using the variable name. Variables are very important concept in programming because of the reasons as mentioned herewith:

1. We can access the same data multiple times using a variable. For example, if we have performed some time-consuming operation and its result is stored in a variable, then we can access the result again and again without executing the same operations again.

2. By giving useful names to variables, we can give meaning to the value stored in it. For example, value 75 is a number, but if we store 75 in a variable called marksOfPython, it has a meaning which tells the user that it refers to the marks in the subject of Python.
3. Variables allow users to handle data easily. Data can be very huge in size, we can access it using the name of a variable instead of memory address. For example, if we have created a variable to access a file, we can perform any operation using a variable on the file.

In Python, we can create a variable by writing its name and assigning a value to it. Value can be assigned to the variable by making use of assignment operator (=). The general syntax to define a variable and assign a value to it is as mentioned:

variable_name = value

Example 2.1 demonstrates the program that shows how to declare a variable and assign a value to it.

Example 2.1: Declaring variable and printing the value of it

```
# Program to create a variable and print it
subject = "Python"
print(subject)
```

OUTPUT:

Python

The above example creates a variable named *subject* and stores the value "Python" in it. The Python interpreter will automatically decide the type of data and store it accordingly in the memory. In Python we can check the data type of the variable that has been created by using the *type()* function. Example 2.2 shows a program to demonstrate the use of type() function.

Example 2.2: Checking the type of created variable.

```
#Program to check the type of variable
subject = "Python"
print(subject)
print(type(subject))
marks = 59
print(marks)
print(type(marks))
```

OUTPUT:

```
Python
<class 'str'>
59
<class 'int'>
```

In the above program we have defined two variables: subjects and marks. The variable subject has been assigned a string “Python” while variable marks is assigned integer value 59. Thus, when we try to find out the type of data that is stored in both these variables using statements *print(type(subject))* and *print(type(marks))*, Python returns <class 'str'> and <class 'int'> respectively.

We can create variables with any name. But it is very important to give meaningful names to the variables. If we create some variables as *a = 21*, we may not understand the exact meaning and use of this variable. Instead, if we create a variable as *age = 21*, by reading the variable name itself we can understand that it stores a value that refers to the age of a person or any other entity.

It is always better to use long names in place of short names when defining a variable. One of the good practices of programming is to use camel case notation to create variables. A camel case notation uses multiple words to define a single variable. For example, we can create variables such as studentName, marksOfPython, dateOfBirth that stores the name of a student, his marks in the subject of Python and his birthdate. We can also use another

convention called snake case or lowercase with underscore to define a variable. Some examples of snake case variable names are student_name, marks_of_python, date_of_birth.

Variable name is known as an identifier in a programming language. We must follow some predefined rules when creating an identifier. The rules are as listed:

1. Identifiers can be created using alphabets (A-Z, a-z), digits(0-9) and underscore(_) only.
2. Any other special character (symbols) and white space are not allowed.
3. First letter in the identifier must be an alphabet or an underscore.

Table 2.1 gives examples of some valid and invalid identifiers (variables).

Table 2.1: Example of valid and invalid identifiers

Valid identifiers	**Invalid identifiers**
student_name	123greetings
SubjectCode1	2Subjects
a121	age of student@
name_of_student	#name
_collegeCode	Birth date

Variable names in Python are case sensitive. It means that entities, ‘name’, ‘NAME’ and ‘Name’ will be treated as three different variables. Example 2.3 shows a program to demonstrate the case sensitivity of variables.

Example 2.3: Check the case sensitivity of the variable

```
#Program to check case sensitivity of variable names
subject = "Python"
print("subject :", subject)
SUBJECT = "Maths"
print("SUBJECT :",SUBJECT)
print("Subject :",Subject)
```

OUTPUT:

```
subject : Python
SUBJECT : Maths
Traceback (most recent call last):
  File "12_3.py", line 6, in <module>
    print("Subject :",Subject)
NameError: name 'Subject' is not defined
```

In the above program we have defined two variables with name: *subject* and *SUBJECT*. They have been assigned values ‘Pyhton’ and ‘Maths’ respectively. We have then printed the values of both the variables. Observe that when we use statement *print("Subject :",Subject)* we get an error, NameError: name 'Subject' is not defined. This clarifies that subject, SUBJECT and Subject are treated as different variables in Python.

Python supports the concept of multiple assignment in a single statement using assignment operator. This concept allows us to assign the same value to multiple variables or specified values to each variable using only one assignment statement. The general syntax to assign same values to multiple variables is as mentioned:

variable_name1 = variable_name2 = ….. variable_namen = value

Example 2.4 shows a program to demonstrate the use of assigning the same value to multiple variables using a single assignment statement.

Example 2.4: Assign same value to multiple variables

```
# Program to assign same value to multiple variables
marksOfPython = marksOfJava = marksOfC = 75
print("Python Marks: ",marksOfPython)
print("Java Marks: ",marksOfJava)
print("C Marks: ",marksOfC)
totalMarks = marksOfPython + marksOfJava + marksOfC
print("Total Marks: ",totalMarks)
```

OUTPUT:

```
Python Marks:  75
Java Marks:  75
C Marks:  75
Total Marks:  225
```

In the above program variables *marksOfPython, marksOfJava* and *marksOfC have* each been assigned an integer value *75* using the statement *marksOfPython = marksOfJava = marksOfC = 75*. The program then prints the value of each of these variables. It then assigns the addition of all these values to variable totalMarks and finally prints its value.

The general syntax to assign different values to multiple variables using single assignment statement is as mentioned:

variable_name1, variable_name2 , .. variable_namen = value1, value2,.. valuen

Example 2.5 shows a program to demonstrate the use of assigning a different value to multiple variables using a single assignment statement.

Example 2.5: Assign different value to multiple variables

```
# Program to assign different values to multiple variables
marksOfPython, marksOfJava, marksOfC = 80,90,75
print("Python Marks: ",marksOfPython)
print("Java Marks: ",marksOfJava)
print("C Marks: ",marksOfC)
totalMarks = marksOfPython + marksOfJava + marksOfC
print("Total Marks: ",totalMarks)
```

OUTPUT:

```
Python Marks:  80
Java Marks:  90
C Marks:  75
Total Marks:  245
```

In the above program variables *marksOfPython, marksOfJava* and *marksOfC have* each been assigned different integer values using the statement *marksOfPython, marksOfJava, marksOfC = 80,90,75*. The program then prints the value of each of these variables. It then assigns the addition of all these values to variable totalMarks and finally prints its value.

Check Your Progress – 1

a) A variable declaration creates a memory space to store the value of data. (True/False)
b) The Python interpreter does not decide the type of data for a variable. (True/False)
c) The NameError: name 'Test' is not defined, indicates the variable Test is not defined in the program. (True/False)
d) SubjectCode$1 is a valid variable name. (True/False)
e) Python supports a concept of multiple assignment in single statement. (True/False)

2.3 DATA TYPES

Every data stored within a variable that is used in a program is of some specific type.

The operations we perform on variables, will vary depending on the type of data stored in them. For example, we can perform arithmetic operations on numeric data, logical operations can be performed on Boolean data, string operations can be implemented on text type data. So, it is very much important to know the type of data.

The concept of data type provides the idea of value that can be stored in a variable and the actions that can be applied to it. Python is considered to be a dynamically typed language, meaning that variable type is determined at runtime based on the value they are allocated. Thus, like many of the other programming languages we do not need an explicit declaration of data type in Python. Programming in Python requires an understanding of data types

because they facilitate efficient data manipulation and a wide range of activities, from text processing and data storage to mathematical computations.

Some of the standard (built-in) data types defined in Python are Text, Numeric, Sequence, Mapping, Set, Boolean, Binary and None. Let us discuss each of these data types in detail.

Text (String) Data type:

The data of text type is stored as a string. A string usually refers to a set or collection of characters. The string data type can be used for any type of collection of characters. The text data type is represented by the *str* class in Python. A string variable have three important properties:

1. Characters: The individual letters in the string are called characters of string.
2. Length: Count of the number of characters in the string is called length of string.
3. Sequence: the characters in the string may appear in a particular sequence, which means each character has a numbered position within the string.

String is a fundamental data type of Python. We often need to use string to print some message or to assign value to the variable. Everything that we enclose within a single quote ('), double quote (") or triple quote (''') is called a string literal. The quotes we use marks the start and end of the string and is known as a **delimiter**.

The delimiters are interchangeable, that is when we use single quotes as a delimiter then double quotes can be used as a character within the string and vice versa. The program in Example 2.6 demonstrates the use of delimiters as string characters.

Example 2.6: Use of delimiter as string character

```
# Program to understand use of string delimiters
myString = "We've got good book to study Python"
print(myString)

myString1 = 'We have got good book named "Python Programming" to study'
print(myString1)
```

OUTPUT:

```
We've got good book to study Python
We have got good book named "Python Programming" to study
```

In the above program two strings *myString* and *myString1* have each been assigned text values. In the variable *myString* double quote (") has been used as a delimiter while single quote (') has been used as a character within a string. Observe that the single quote has been displayed in output along with other characters. Similarly, in the variable *myString1* single quote (') has been used as a delimiter while double quote (") has been used as a character within a string.

Once a string has been defined, we can perform operations like fetching a single character or fetching a sub string (set of characters) from the string using indexing and slicing. It is also possible to concatenate two strings and create a new string. Each character within the string has a numbered position called index. The general representation of a string is shown in Figure 2.1.

str	m	y		c	o	u	n	t	r	y
index	0	1	2	3	4	5	6	7	8	9

Figure 2.1: Representation of string

We can access any character at any position in the string using square brackets [ ].

Concatenation

Two strings can be combined to form a new string using concatenation operation. The + operator when used with strings performs this operation. The general syntax to perform concatenation is as mentioned:

newString = string1 + string2 + … + stringn

The program in Example 2.7 demonstrates various forms of slicing along with concatenation operation.

Example 2.7: Different operations on string

```
# Program to demonstrate slicing and concatenation operation
myString = "my country"
print("Original String: ",myString)

indexChar=myString[4]      # Fetch the character at index 4
print("Character at index 4 is: ",indexChar)

str=myString[0:4].    # Fetch first four characters
print("First 4 characters are: ",str)

str1=myString[:4]    #Omit first number in range
print("First 4 characters are: ",str1)

str2=myString[5:]    #Omit last number in range
print("Substring from 5th index to end of string is: ",str2)

newString = str1 + " " + str2    #Concatenation of strings
print("Concatenated String: ",newString)
```

OUTPUT:

```
Original String:  my country
Character at index 4 is:  o
First 4 characters are:  my c
First 4 characters are:  my c
Substring from 5th index to end of string is:  untry
Concatenated String:  my c untry
```

In the above program a string *myString* has been assigned a value “my country”. We have then fetched and printed the character (‘o’) at 4^{rth} index. We have then created three strings namely: *str*, *str1* and *str2* using slicing techniques explained previously and printed their values. Finally, we have concatenated the string *str1* (“my c”), a space and the string *str2* (“untry”) and generated a new string newString with value “my c untry”. We will discuss the concept of string in detail in later chapters.

Numeric Data type:

Numeric data type can be categorized into two types: integer and float. Integer numbers are whole numbers and floats are decimal numbers. We can perform arithmetic operations such as addition (+), subtraction (-), and division (/) on numeric type data. Example 2.8 demonstrates the use of numeric data types in python.

Example 2.8: Operations on numeric data type

```
# Operations on numeric data type
intVarX = 15
intVarY = 23
floatVarZ = 3.14

# Addition
intSum = intVarX + intVarY
print("Sum of two integers: ",intSum)
mixSum = intVarX + floatVarZ
print("Sum of mixed data type variables: ",mixSum)

# Division
intDiv = intVarX / intVarY
print("Division of two integers: ",intDiv)
mixDiv = intVarX / floatVarZ
print("Division of mixed data type variables: ",mixDiv)
```

```
# Multiplication
intMul = intVarX * intVarY
print("Multiplication of two integers: ",intMul)
mixMul = intVarX / floatVarZ
print("Multiplication of mixed data type variables: ",mixMul)
```

OUTPUT:

```
Sum of two integers:  38
Sum of mixed data type variables:  18.14
Division of two integers:  0.6521739130434783
Division of mixed data type variables:  4.777070063694267
Multiplication of two integers:  345
Multiplication of mixed data type variables:  4.777070063694267
```

The program above shows the basic arithmetic operations that can be performed with numeric data. Observe that the data type of the output depends on the data types of the operands used. For the sum and multiplication operations the integer operands have resulted in an integer output. In all other cases the data type of the output will be <class 'float'>.

Sequence Data type:

Sequence data types in Python include List, Tuple and Range. A list is an ordered, mutable collection of items. A tuple is an ordered, immutable collection of items. A range is an immutable sequence of numbers. Different operations like indexing, slicing etc can be performed on these sequences. We will learn the sequence data type in detail in the coming chapters.

Mapping Data type:

A dictionary (dict) is considered to be a mapping data type. It is an unordered and mutable collection of key-value pairs. Operations like retrieving a value using a key, modifying key-value pairs or removing a key-value pair can be performed in the dictionary. The detailed discussion on the dictionary will be done in later chapters.

Set Data type:

A set is an unordered collection of unique items. Operations like adding items, removing items, performing union, intersection, or finding differences etc can be performed on sets. The detailed discussion on the dictionary will be done in later chapters.

Boolean Data type:

The boolean (bool) data type represents one of two values: True or False. We can usually perform comparison or logical operations on this data type.

Binary Data type:

At times we need to work with binary data. In Python Bytes and Bytearray are used to store binary data. Bytes refer to an immutable sequence of bytes, while Bytearray refers to a mutable sequence of bytes. Example 2.9 demonstrates the use of binary data type in python.

Example 2.9: Use of binary data type

```
# Program to show use of binary data type
byteData = b"Python Program"
print("Byte Literal : ",byteData)

# Accessing bytes
firstByte = byteData[0]
print("Data of first byte :",firstByte)
secondByte = byteData[1]
print("Data of second byte :",secondByte)

# Creating bytearray
byteArray = bytearray([80, 121, 116, 104, 111, 110])
print("Values in array: ",byteArray)
```

OUTPUT:

```
Byte Literal :  b'Python Program'
Data of first byte : 80
Data of second byte : 121
Values in array:  bytearray(b'Python')
```

In the above program the statement *byteData = b"Python Program"* creates a byte literal. A byte literal in Python is represented by a string prefixed with the letter *b*. It is used to store the string "Python Program" as a sequence of bytes. When *b* is prefixed to a string, each character in the string is represented by its corresponding ASCII or Unicode value when it is stored in memory. For example, the character 'P' is represented as 80, character 'y' is represented as 121 in ASCII and so on. The statement *print("Byte Literal : ", byteData)* displays the byte literal on the screen. Here, *b* in the output indicates that the string is a byte object. Then we have tried to access individual elements located at index 0 and 1 from the byte literal. The statement *byteArray = bytearray([80, 121, 116, 104, 111, 110])* creates a mutable byte array. The elements within it can be changed. The *bytearray()* function here takes a list of integers as an argument, these integers represent byte (ASCII) values for each character. The list elements 80, 121, 116, 104, 111 and 110 corresponds to the ASCII values for each of the characters in the word "Python" as can be seen in the output.

None Data type:

The None data type represents the absence of a value or a null value. Such a data type is used for assignment or comparison. Example 2.10 demonstrates the use of none data type in python.

Example 2.10: Use of None data type

```
# Program to show use of None data type
varNone = None
varInt = 99
```

```
isNone = (varNone is None)
print(isNone)  # Output: True
print(varInt == None)
```

OUTPUT:

True

False

In the above program we have defined and declared two variables: *varNone* and *varInt*. The variable *varNone* is assigned the value *None*. None is a special constant that represents a null value in Python (Observe that it is not enclosed in a single or double quote). The variable *varInt* is assigned an integer value 99. The statement *isNone = (varNone is None)* first checks whether the value of variable *varNone* is null. The *is* operator here checks if two variables refer to the same object in memory. Since variable *varNone* was explicitly set to *None*, this condition returns True. The last statement *print(varInt == None)* compares the value of variable *varInt* (which is 99) to *None*. In this case, 99 == None returns False.

Check Your Progress – 2

a) The 'int' data type is used to store decimal numbers in Python. (True/False)

b) The 'str' data type is used to store sequences of characters in Python. (True/False)

c) The 'set' data type allows duplicate elements to be stored in Python. (True/False)

d) Python supports automatic type conversion between 'int' and 'float' values. (True/False)

e) The `bool` data type in Python only has two possible values: 'True' and 'False'. (True/False)

2.4 LET US SUM UP

In this chapter we have discussed what a variable is, how to define and assign value to it. We also saw how multiple operations can be done using a single assignment statement. We further looked at the basic data types supported by Python. We understood the use of the data types like Text, Numeric, Sequence, Mapping, Set, Boolean, Binary and None.

2.5 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

1-a True
1-b False
1-c True
1-d False
1-e True
2-a False
2-b True
2-c False
2-d True
2-e True

2.6 ASSIGNMENTS

1. What is a variable?
2. Explain how a single value can be assigned to multiple variables using an assignment statement?
3. What is the importance of data type?
4. Explain what is the use of text data type. How can we find out the length of the text variable?
5. Explain the use of binary data types.

Unit-3: Operators and Type Casting

3

Unit Structure

3.0 LEARNING OBJECTIVE

After studying this unit student should be able to:

- Understand the use of operators
- Implementation of operators for different use
- Understand the type casting of different variables
- Write programs to use operators and type casting

3.1 INTRODUCTION

In the previous chapter we looked into the basics of variables and datatypes. Using variables, we are able to perform different operations on it. To perform the operations, we need to use a different set of operators.

Python supports a wide range of operators. Operators are special symbols which are useful when we need to perform operations on the operands (variable or constants). In this chapter we will learn what operator is, we will also look into different types of operators and their usage. Lastly, we will learn the concept of type casting.

3.2 OPERATORS

Operators are the special symbols or keywords which are useful to perform operations on various variables and constant values. Operators are the building blocks of a program, they allow us to implement a wide range of operations including simple arithmetic operations, complex logical operations and data manipulation. Python's operators work well with a wide range of data types, it thus increases the language's efficiency and versatility.

Python supports different categories of operators, like arithmetic, comparison, logical, bitwise, assignment, membership, and identity operators. It is very important to understand how the operators work to implement it effectively in our programs. In this section, we will discuss operators in detail.

Arithmetic Operators:

The arithmetic operators in Python are used to perform basic mathematical operations like addition, subtraction, multiplication, etc. Table 3.1 shows the list of arithmetic operators along with its description and example.

Table 3.1: Arithmetic Operators

Operator	Description	Example Use
+	Addition	11 + 3
-	Subtraction	7 – 3
*	Multiplication	8 * 3
/	Division	7 / 2
//	Floor Division	9 // 2
%	Modulus (remainder)	11 % 2
**	Exponentiation	3 ** 2

Each of the operators shown in Table 3.1 performs a specific mathematical operation in Python. Let us discuss them one by one.

Addition (+): The addition operator returns a sum of numeric values or it can also be used to concatenate (join) strings. Few examples of using addition operator are as mentioned:

```
>>> print(11 + 3 + 2 + 7)
23
>>> print('Hello' + ' ' + 'Good Morning' )
Hello Good Morning
```

Subtraction (-): The subtraction operator is used to subtract a set of numeric values. Few examples of using subtraction operator are as mentioned:

```
>>> print(11 - 3)
8
>>> print(11 - 3 - 2 - 7)
-1
```

Multiplication (*): The multiplication operator is used to multiply numeric values or it can also be used to repeat strings. Few examples of using multiplication operator are as mentioned:

```
>>> print(8 * 3)
24
>>> print('Hi' * 3)
HiHiHi
>>> print('Hi ' * 3)
Hi Hi Hi
```

Observe the difference in the output of print('Hi' * 3) and print('Hi ' * 3). In the first case we get repetition of string without a blank space.

Division (/): The division operator is used to divide a set of numeric values. The division operation always returns a decimal value. Few examples of using division operator are as mentioned:

```
>>> print(7 / 2)
3.5
>>> print(10 / 2)
5.0
>>> print(3.14 / 1.4)
2.242857142857143
>>> print(7 / 2 / 4)
0.875
```

Floor Division (//): The floor division operator is used to divide a set of numeric values, after dividing it truncates the decimal part. Few examples of using floor division operator are as mentioned:

```
>>> print(7 // 2)
3
>>> print(10 // 2)
5
>>> print(7 // 2 // 4)
0
>>> print(3.14 // 1.4)
2.0
>>> print(3.14 // 2)
1.0
```

Note:

- **If all operands of floor division are integers, we will get an integer value as output.**
- **If even one operand of the floor division is decimal, we will get a truncated decimal value as output.**

Modulus (%): The modulus operator is used to perform a division operation, it returns a remainder that is obtained from the division of numeric values. Few examples of using modulus operator are as mentioned:

```
>>> print(11 % 2)
1
>>> print(15 % 3)
0
>>> print(17.5 % 3)
2.5
>>> print(17 % 3.5)
3.0
```

Note:

- **For 17.5 % 3 the result is calculated as 17.5 - (3 * 5) = 17.5 - 15 = 2.5**
- **For 17 % 3.5 the result is calculated as 17 - (3.5 * 4) = 17 - 14 = 3**

Exponentiation (**): The exponent operator raises the first number to the power of the second. Few examples of using exponent operator are as mentioned:

```
>>> print(3 ** 2)
9
>>> print(3.5 ** 2)
12.25
```

The Python program in Example 3.1 demonstrates the use of arithmetic operators.

```
#program to understand various arithmetic operators.

var1=25
var2=3

#op + will add var1 and var2
add=var1+var2
print("Addition of ",var1," and ",var2," is ",add)

# op – will subtract var1 from var2
sub=var1-var2
print("Subtraction of ",var1," from ",var2," is ",sub)

# op * will multiply var1 and var2
mul=var1*var2
```

```
print("Multiplication of ",var1," and ",var2," is ",mul)

# op / will divide var1 with var2
div=var1/var2
print("Division of ",var1," and ",var2," is ",div)

# op // will divide var1 with var2 and truncate the decimal part
floorDiv=var1//var2
print("Floor division of ",var1," and ",var2," is ",floorDiv)

# op % will divide var1 with var2 and return the remainder
mod=var1%var2
print("Modulus division of ",var1," with ",var2," is ",mod)

# op ** will raises var1 to the power of var2
exp=var1**var2
print("Exponential of ",var1," with ",var2," is ",exp)
```

OUTPUT:

```
Addition of 25 and 3 is 28
Subtraction of 25 from 3 is 22
Multiplication of 25 and 3 is 75
Division of 25 and 3 is 8.333333333333334
Floor division of 25 and 3 is 8
Modulus division of 25 with 3 is 1
Exponential of 25 with 3 is 15625
```

The above program demonstrates the use of various arithmetic operators in Python. It begins by initializing two variables, *var1* and *var2* with values 25 and 3 respectively. The program then performs addition, subtraction, multiplication, division, floor division, modulus and exponentiation operations on var1 and var2 and prints the result of each of these operations.

Comparison or Relational Operators:

The comparison or relational operators in Python are used to compare two values; it returns a Boolean result as True or False. Table 3.2 shows the list of comparison operators along with its description and example.

Table 3.1: Arithmetic Operators

Operator	Description	Example Use
==	Equal to	15 == 15
!=	Not equal to	15 != 13
>	Greater than	15 > 13
<	Less than	13 < 15
>=	Greater or equal	15 >= 15
<=	Less or equal	13 <= 15

Let us discuss the comparison operators one by one.

Equal to (==): The equal to operator checks whether two numeric or string values are equal. If both operands are the same it returns True, otherwise it returns False. Few examples are as mentioned:

```
>>> print(15 == 15)
True
>>> print(15 == 14)
False
>>> print("Hi" == "Hi")
True
>>> print("Hi" == "Bye")
False
```

Not equal to (!=): The not equal to operator checks whether two numeric or string values are different. If both operands are different it returns True, otherwise it returns False. Few examples are as mentioned:

```
>>> print(15 != 15)
False
>>> print(15 != 14)
True
>>> print("Hi" != "Hi")
False
>>> print("Hi" != "Bye")
True
```

Greater than (>): The greater than operator checks whether the value of first operand is greater than the value of second operand (in case of string it compares the ASCII values of the letters). If it is greater, we get True as output otherwise we get False. Few examples are as mentioned:

```
>>> print(15 > 14)
True
>>> print(15 > 19)
False
>>> print("Hi" > "Bye")
True
>>> print("Hi" > "Hello")
True
>>> print("Hi" > "Hi")
False
>>> print("Ape" > "Ate")
False
```

Less than (<): The less than operator checks whether the value of the first operand is lesser than the value of second operand. If it is lesser, we get True as output otherwise we get False. Few examples are as mentioned:

```
>>> print(15 < 14)
False
>>> print(12 < 14)
True
>>> print("Ape" < "Ate")
True
>>> print("Hi" < "Hello")
False
>>> print("Hi" < "Hi")
False
```

Greater or equal (>=): The greater than or equal operator checks whether the value of first operand is greater than or equal to the value of second operand (in case of string it compares the ASCII values of the letters). If it is greater or equal, we get True as output otherwise we get False. Few examples are as mentioned:

```
>>> print(15 >= 14)
True
>>> print(15 >= 19)
False
>>> print(15 >= 15)
True
>>> print("Hi" >= "Hello")
True
>>> print("Hi" >= "Hi")
True
>>> print("Ape" >= "Ate")
False
```

Less or equal (<=): The less than or equal operator checks whether the value of the first operand is lesser than or equal to the value of the second operand. If it is lesser or equal, we get True as output otherwise we get False. Few examples are as mentioned:

```
>>> print(15 <= 14)
False
>>> print(15 <= 15)
True
>>> print(11 <= 14)
True
>>> print("Ape" <= "Ate")
True
>>> print("Hi" <= "Hi")
True
>>> print("Hi" <= "Hello")
False
```

The Python program in Example 3.2 demonstrates the use of comparison operators.

Example 3.2: Example of Comparison Operators

```
# Program to examine the use of Comparison Operators.

perOfStud1=90

perOfStud2=90



print("Output of == operator:")

check = perOfStud1 == perOfStud2

print("",perOfStud2," == ",perOfStud1," : ",check)

perOfStud2=92
```

```
print("Output of != operator")

check = perOfStud1 != perOfStud2

print("",perOfStud2," != ",perOfStud1," : ",check)



print("Output of > operator")

check = perOfStud2 > perOfStud1

print("",perOfStud2," > ",perOfStud1," : ",check)



print("Output of < operator")

check = perOfStud1 < perOfStud2

print("",perOfStud1," < ",perOfStud2," : ",check)



print("Output of >= operator")

check = perOfStud2 >= perOfStud1

print("",perOfStud2," >= ",perOfStud1," : ",check)



print("Output of <= operator")

check = perOfStud1 <= perOfStud2

print("",perOfStud1," <= ",perOfStud2," : ",check)
```

OUTPUT:

Output of == operator:

90 == 90 : True

Output of != operator

```
 92  !=  90  :  True

Output of > operator

 92  >  90  :  True

Output of < operator

 90  <  92  :  True

Output of >= operator

 92  >=  90  :  True

Output of <= operator

 90  <=  92  :  True
```

The Python program in Example 3.2 demonstrates the use of comparison operators in different conditions. The example compares the percentage of two students using different comparison operators.

Logical Operators

The logical operators in Python are used to combine the result of more than one conditional statement. Table 3.3 shows the list of logical operators along with its description and example.

Table 3.3: Logical Operators

Operator	Description	Example Use
and	True if all conditional statements return True	(5 > 3) and (8 > 5)
or	True if one conditional statement returns True	(5 > 3) or (8 < 5)
not	Inverts the Boolean result	not(5 > 3)

Let us discuss the logical operators one by one.

AND (‘and’): The logical operator and, when used in conjunction with conditional statements, returns True only if all the conditions used in an expression return True. In all other cases it will return false. Few examples are as mentioned:

```
>>> print((15 > 14) and (12 > 10) and (14 > 10))
True
>>> print((15 > 14) and (12 > 10) and (14 > 17))
False
>>> print(('Hi' >= 'Hi') and (12 > 10) and (14 > 17))
False
>>> print(('Hi' >= 'Hi') and (12 > 10) and (14 < 17))
True
```

OR (‘or’): The logical operator or, when used in conjunction with conditional statements, returns True if at least one of the conditions used in an expression returns True. In all other cases it will return false. Few examples are as mentioned:

```
>>> print((15 > 14) or (12 < 10) or (14 == 10))
True
>>> print((15 < 14) or (12 < 10) or (14 == 10))
False
>>> print(('Hi' == 'Hi') or (12 > 10) or (14 < 17))
True
>>> print(('Hi' >= 'Hello') or (12 < 10) or (14 > 17))
True
```

NOT (‘not’): The logical operator not, when used in conjunction with conditional statements reverses the result of a condition. Few examples are as mentioned:

```
>>> print(not((15 > 14) or (12 < 10) or (14 == 10)))
False
>>> print(not('Hi' >= 'Hello'))
False
>>> print(not(14 == 10))
True
```

The Python program in Example 3.3 demonstrates the use of logical operators.

Example 3.3: Example of Logical Operators

```
#Program to use Logical Operators
perOf10th=78
perOf12th=65
check = (perOf10th >= 70) and (perOf12th >= 60)
print("",perOf10th," >= 70 AND ",perOf12th," >= 60 :",check)
check = (perOf10th >= 70) or (perOf12th >= 80)
print("",perOf10th," >= 70 OR ",perOf12th," >= 60 :",check)
check = not(perOf10th >= 70)
print(" NOT ",perOf10th," >= 70 ",check)
```

```
OUTPUT:
78  >= 70 AND  65  >= 60 : True
78  >= 70 OR  65  >= 60 : True
NOT  78  >= 70  False
```

Example 3.3 demonstrates the use of logical operators. Here, *perOf10th* and *perOf12th* are two variables that store the percentage of 10^{th} and 12^{th}. We are comparing the percentage of 10^{th} and 12^{th} to check whether it is greater than 70 and 60 using 'and' operator, it will return true. The 'or' operator will check whether the percentage of 10^{th} is greater or equals to 70 or percentage of 12^{th} is greater or equals to 80, it will return 'true' because perOf10th = 78 and it is >=70.

Bitwise Operators

Bitwise operators in Python are used to perform operations on individual bits of the stored integer value. These operators work directly on the binary representation of numbers rather than the decimal numbers. Table 3.4 shows the list of bitwise operators along with its description and example.

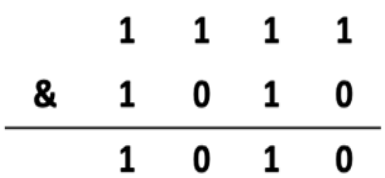
Table 3.4: Bitwise Operators

Operator	Description	Example Use
&	AND	5 & 3
|	OR	5 | 3
^	XOR	5 ^ 3
~	NOT	~5
<<	Left Shift	5 << 1
>>	Right Shift	5 >> 1

Let us discuss the logical operators one by one.

AND (&): The bitwise & operator compares each bit of two numbers and returns 1 if both bits are 1, otherwise it returns 0. For example, assume that we have numbers 15 and 10. First, convert the numbers 15 and 10 to their binary representations. The number 15 in decimal is represented as 1111 in binary. Similarly, the number 10 in decimal is represented 1010 in binary. Now perform the bitwise and of both these binary numbers. The result of this operation is binary 1010 which when converted to decimal number gives decimal number 10. Figure 3.1 shows the bitwise AND operation.

Figure 3.1: Bitwise AND operation

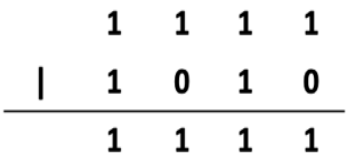


Few examples are as mentioned:

```
>>> print(15 & 10)
10
>>> print(15 & 15)
15
>>> print(5 & 15)
5
>>> print(25 & 15)
9
```

OR (|): The bitwise | operator compares each bit of two numbers and returns 1 if either of the bits is 1 or both the bits are 1, otherwise it returns 0. Figure 3.2 shows the bitwise OR operation on 15 and 10.

Figure 3.2: Bitwise OR operation

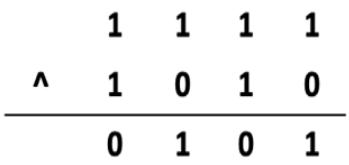


Few examples are as mentioned:

```
>>> print(15 | 10)
15
>>> print(15 | 15)
15
>>> print(5 | 15)
15
>>> print(25 | 15)
31
```

XOR (^): The bitwise ^ operator (exclusive OR) operator compares each bit of two numbers and returns 1 if the bits are different, otherwise it returns 0. Figure 3.3 shows the bitwise XOR operation on 15 and 10.

Figure 3.3: Bitwise XOR operation

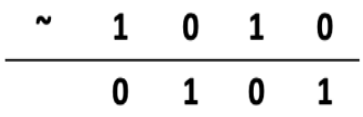


Few examples are as mentioned:

```
>>> print(15 ^ 10)
5
>>> print(15 ^ 15)
0
>>> print(5 ^ 15)
10
>>> print(25 ^ 15)
22
```

NOT (~): The bitwise NOT operator flips all the bits of a numeric value, it turns all 1s into 0s and vice-versa. Figure 3.4 shows the bitwise NOT operation on numeric value 10.

Figure 3.4: Bitwise NOT operation



Few examples are as mentioned:

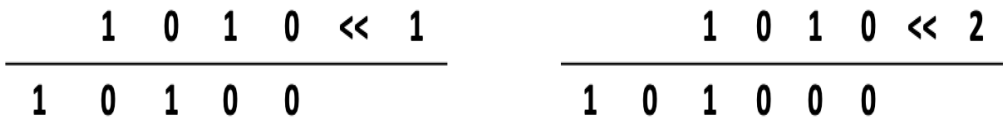
```
>>> print(~15)
-16
>>> print(~10)
-11
>>> print(~-5)
4
>>> print(~-25)
24
```

Note:

- **The negation operation is also known as the "one's complement".**

Left Shift (<<): The left shift << operator shifts the bits of a number to the left by a specified number of positions. Figure 3.5 shows the bitwise left shift operation on numeric value 10.

Figure 3.5: Bitwise Left shift operation with 1 bit and 2 bits



Few examples are as mentioned:

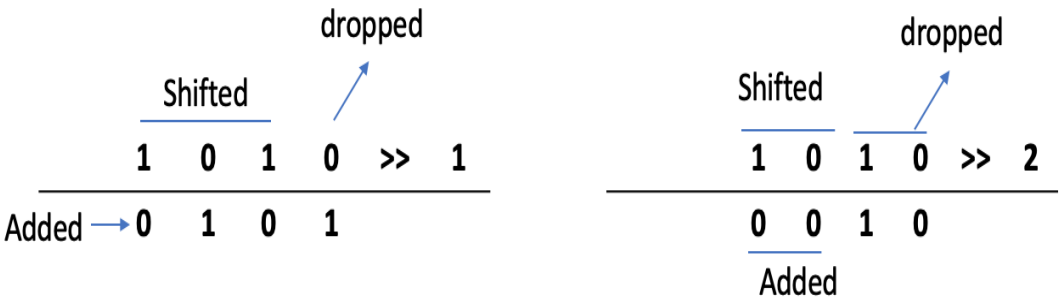
```
>>> print(10 << 1)
20
>>> print(10 << 2)
40
>>> print(-10 << 1)
-20
>>> print(-10 << 2)
-40
```

Note:

- **Each left shift by one position effectively multiplies the number by 2.**

Right Shift (>>): The right shift >> operator shifts the bits of a number to the right by a specified number of positions. Figure 3.6 shows the bitwise left shift operation on numeric value 10.

Figure 3.6: Bitwise Right shift operation with 1 bit and 2 bits



Few examples are as mentioned:

```
>>> print(10 >> 1)

5

>>> print(10 >> 2)

2

>>> print(-10 >> 1)

-5

>>> print(-10 >> 2)

-3
```

Note:

- **Each right shift by one position effectively divides the number by 2 and discards the remainder.**

The Python program in Example 3.4 demonstrates the use of bitwise operators.

Example 3.4: Example of bitwise Operators

```
#Program to check the usefulness of bitwise operators.
var1=5
var2=2
check=var1 & var2
print("",var1," & ",var2," : ",check)
check=var1 | var2
print("",var1," | ",var2," : ",check)
check=var1 ^ var2
print("",var1," ^ ",var2," : ",check)
check= ~var2
print("~",var2," : ",check)
check=var1 << var2
print("",var1," << ",var2," : ",check)
check=var1 >> var2
print("",var1," >> ",var2," : ",check)
```

OUTPUT:

```
5 & 2 : 0
5 | 2 : 7
5 ^ 2 : 7
~ 2 : -3
5 << 2 : 20
5 >> 2 : 1
```

Example 3.4 demonstrates the example of bitwise operators. The bitwise operator performs binary operations on the operands. In the example, *var1* and *var2* are variables of integer type having values 5 and 2 respectively. The output shows results of different bitwise operations performed on both these variables.

Check Your Progress – 1

a) Operators are special symbols or keywords of Python. (True/False)

b) The '+' operator can be used with strings. (True/False)

c) The comparison operator when used returns a string. (True/False)

d) The AND operator will return True if both the conditions of an expression result in TRUE. (True/False)

e) Bitwise operator will perform operations on bits of the numeric operands. (True/False)

Assignment Operators:

The assignment operators in Python are used to assign values to the variables used in the program. Table 3.5 shows the list of assignment operators along with its description and example.

Table 3.5: Assignment Operators

Operator	Description	Example Use
=	Assigns the value on the RHS of the expression to the variable on the LHS of the expression.	x = 5
+=	Adds the value on the RHS of the expression to the variable on the LHS of the expression and assigns the result to the variable on LHS.	x += 3 becomes x = x + 3
-=	Subtracts the value on the RHS of the expression from the variable on the LHS of the expression and assigns the result to the variable on LHS.	x -= 3 becomes x = x - 3
*=	Multiplies the value on the RHS of the expression by the variable on the LHS of the expression and assigns the result to the variable on LHS.	x *= 3 becomes x = x * 3
/=	Divides the variable on the LHS of the expression by the value on the RHS of the expression and assigns the result to the variable on LHS.	x /= 3 becomes x = x / 3
//=	Performs floor division of the variable on the LHS of the expression by the value on the RHS of the expression and assigns the result to the variable on LHS.	x //= 3 becomes x = x // 3

%=	Performs modulus division of the variable on the LHS of the expression by the value on the RHS of the expression and assigns the result to the variable on LHS.	x %= 3 becomes x = x % 3
**=	Performs exponentiation of the variable on the LHS of the expression by the value on the RHS of the expression and assigns the result to the variable on LHS.	x **= 3 becomes x = x ** 3
&=	Performs bitwise AND of the variable on the LHS of the expression by the value on the RHS of the expression and assigns the result to the variable on LHS.	x &= 3 becomes x = x & 3
|=	Performs bitwise OR of the variable on the LHS of the expression by the value on the RHS of the expression and assigns the result to the variable on LHS.	x |= 3 becomes x = x | 3
^=	Performs bitwise XOR of the variable on the LHS of the expression by the value on the RHS of the expression and assigns the result to the variable on LHS.	x ^= 3 becomes x = x ^ 3
<<=	Performs bitwise left shift of the variable on the LHS of the expression by the value on the RHS of the expression and assigns the result to the variable on LHS.	x <<= 3 becomes x = x << 3
>>=	Performs bitwise right shift of the variable on the LHS of the expression by the value on the RHS of the expression and assigns the result to the variable on LHS.	x >>= 3 becomes x = x >> 3

The Python program in Example 3.5 demonstrates the use of various assignment operators.

Example 3.5: Example of Assignment Operator

```
# Program to demonstrate assignment operations

number = 5

const = 3

print("Number = ",number)
```

```
print("Constant = ",const)

number += const

print("Value of number after addition of const",number)

number *= const

print("Value of number after multiplication of const",number)

number -= const

print("Value of number after subtraction of const",number)

number /= const

print("Value of number after division by const",number)

number //= const

print("Value of number after floor division by const",number)

number %= const

print("Value of number after modulo by const",number)

number **= const

print("Value of number after exponential by const",number)

number = 15

print("New value of number ",number)

number &= const

print("Value of number after bitwise AND by const",number)

number |= const

print("Value of number after bitwise OR by const",number)

number ^= const

print("Value of number after bitwise XOR by const",number)
```

```
number <<= const

print("Value of number after left shift by const",number)

number >>= const

print("Value of number after right shift by const",number)

print("Final value of number is ",number)
```

OUTPUT:

```
Number =  5

Constant =  3

Value of number after addition of const 8

Value of number after multiplication of const 24

Value of number after subtraction of const 21

Value of number after division by const 7.0

Value of number after floor division by const 2.0

Value of number after modulo by const 2.0

Value of number after exponential by const 8.0

New value of number 15

Value of number after bitwise AND by const 3

Value of number after bitwise OR by const 3

Value of number after bitwise XOR by const 0

Value of number after left shift by const 0

Value of number after right shift by const 0

Final value of number is  0
```

Example 3.5 demonstrates the use of assignment operators. In the example, *number* and *const* are variables having values 5 and 3 respectively. The output shows results of different assignments performed on both these variables. Observe that we have reassigned the value of variable *number* to 15 before performing the bitwise assignments. This was necessary as the value of the variable *number* has become 7.0 after performing the division. It is not possible to perform bitwise operations with decimal values.

Membership Operators

The membership operators check for membership of an element within a sequence. Table 3.6 shows the list of membership operators along with its description and example.

Table 3.6: Membership Operators

Operator	Description	Example
in	True if present	'a' in 'apple'
not in	True if not present	'b' not in 'apple'

The Python program in Example 3.6 demonstrates the use of membership operators.

Example 3.6: Example of Member Operator

```
# Program to show use of membership operator
text = "Good Morning"
print("Small g in text",'g' in text)
print("Small m not in text",'m' not in text)
```

OUTPUT:

```
Small g in text True
Small m not in text True
```

The above program the membership operator *'g' in text* checks for the occurrence of the lowercase 'g' in the string. In the string "Good Morning", the character 'g' is present at the end of the string, so operator *'g' in text* will return True. The membership operator *'m' not in text* checks if the lowercase 'm' is not found in the string. In the string "Good Morning", Upper case 'M' is present but lower case 'm' is not present, hence we False in output.

Identity Operators

The identity operators are used to compare the memory locations of two objects. These operators are used to check whether two variables point to the same object in memory or not. Table 3.7 shows the list of identity operators along with its description and example.

Table 3.7: Identity Operators

Operator	Description	Example Use
is	True if both objects point to same location	x is y
is not	True if both objects do not point to same location	x is not y

The Python program in Example 3.7 demonstrates the use of membership operators.

Example 3.7: Example of Identity Operator

```
# Program to show use of identity operator

list1 = [10, 20, 30, 'Python', 3.14, 7.28]

list2 = list1

print("Are list1 and list2 in same location :",list1 is list2)

list3 = [10, 20, 30, 'Python', 3.14, 7.28]

print("Are list1 and list3 in same location :",list1 is list3)

list3 = [10, 20, 30, 'Python', 3.14, 7.28]

print("Are list3 and list1 not in same location :",list1 is not list3)
```

OUTPUT:

```
Are list1 and list2 in same location : True
Are list1 and list3 in same location : False
Are list3 and list1 not in same location : True
```

The above program demonstrates the use of the identity operator (is and is not) in Python. It checks if two variables *list1* and *list2* refer to the same memory location or not. Initially a list named *list1* is defined along with values. Then *list2* is assigned to *list1*, Thus, both *list1* and *list2* now point to the same memory location. The statement, *list1 is list2* hence returns True.

We then create a new list named *list3* with the same contents as that of *list1*, but it is a different object in memory. Therefore, the statement *list1 is list3* will return False because although the content is the same, they are not the same object in memory. Finally, the statement *list1 is not list3* will return True, since list1 and list3 are different objects.

Check Your Progress – 2

a) The '=' and '==' operators will perform the same operation. (True/False)
b) The assignment operation x += 3 becomes x + 3 = x. (True/False)
c) The 'is' operator will return true if both the operands are in the same memory location. (True/False)
d) The 'in' and 'not in' operators are membership operators. (True/False)
e) The operation number &= const can only be performed when both number and const are of integer type. (True/False)

3.2 TYPE CASTING

Python is a dynamically typed language. It will assign the data type to the variable according to the data stored in it. Sometimes we need to convert data from one type to another to perform operations. The process of converting a data type into another is known as type casting. Python provides various built-in functions to perform type casting, making it a versatile and user-friendly language for managing data types. Type casting can be done in two ways namely: implicit and explicit.

Implicit Type Casting

Implicit type casting occurs automatically in Python when it is safe to perform. This happens during operations where Python converts smaller data types into larger data types to prevent data loss. Example 3.8 demonstrates the implicit type casting done by Python.

Example 3.8: Implicit Type Casting example

```
# Program to Implicit type casting example
var1 = 10      # Integer Variable
var2 = 2.5     # Float Variable
add = var1 + var2   # Integer is implicitly converted to float
print("Addition of",var1," and ",var2,"is",add)
print("Datatype of variable add is",type(add))
```

OUTPUT

```
Addition of 10  and  2.5 is 12.5
Datatype of variable add is <class 'float'>
```

In the above program variable *var1* is of integer type and *var2* is in float type, when we perform addition operation on it, Python compiler will first implicitly convert the integer variable *var1* to float, then add it to the float variable *var2* and assign the result to variable *add*. When we print the datatype of variable *add*, we get <class 'float'> as output, indicating that variable *add* stores a float value.

Explicit Type Casting

Explicit type casting, also known as type conversion, requires the programmer to manually convert one data type into another using Python's built-in functions. The list of common functions used for explicit type casting is as mentioned:

Common Functions for Explicit Type Casting:

- int() – Converts a value to an integer.
- float() – Converts a value to a float.

- str() – Converts a value to a string.
- list() – Converts an iterable to a list.
- tuple() – Converts an iterable to a tuple.
- set() – Converts an iterable to a set.
- bool() – Converts a value to a Boolean.

Example 3.9 demonstrates the example of explicit type casting using above listed functions.

Example 3.9: Example of Explicit Type casting

```
#Program of Explicit type casting examples
num = 5.8
num_int = int(num)  # Truncates the decimal part
print("Float",num,"converted to integer",num_int,"using int().")
print("Datatype of num is",type(num))
print("Datatype of num_int is",type(num_int))

str1 = "10.5"
num_float = float(str1)
print("String",str1,"converted to float",num_float,"using float().")
print("Datatype of str1 is",type(str))
print("Datatype of num_float is",type(num_float))

n = 100
n_str = str(n)
print("Integer",n,"converted to string",n_str,"using str().")
print("Datatype of n is",type(n))
print("Datatype of n_str is",type(n_str))
```

OUTPUT:

```
Float 5.8 converted to integer 5 using int().
Datatype of num is <class 'float'>
Datatype of num_int is <class 'int'>
String 10.5 converted to float 10.5 using float().
Datatype of str1 is <class 'str'>
Datatype of num_float is <class 'float'>
Integer 100 converted to string 100 using str().
Datatype of n is <class 'int'>
Datatype of n_str is <class 'str'>
```

In the given example, the *num* variable stores the value of float type, we are explicitly converting it into integer type with the use of int(). It will truncate the decimal part of the data and store it into the num_int variable. Another variable *str1* is a string, we are converting it into a float number using float(). Using the string data, we cannot perform arithmetic operations; if we convert it to a numeric data type, we can perform any arithmetic operation on it. Lastly, we have converted an integer variable *n* into a string using str().

Check Your Progress – 3

a) Implicit type casting will be performed by Python automatically. (True/False)

b) Python can convert float variables to integer variables automatically. (True/False)

c) The float variable can be converted to integer type using int(). (True/False)

d) If variable var=”20”, then var1=int(var) is the correct conversion. (True/False)

3.4 LET US SUM UP

In this chapter we have discussed various operators and use of those operators. Now we are capable of comparing the result of various conditions and perform as per the result of different conditions. The data type casting is also discussed. When we want to perform the conversion of data types, we can use explicit type casting. Python performs implicit type casting as and when required.

3.5 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

1-a False

1-b True

1-c False

1-d True

1-e True

2-a False

2-b False

2-c True

2-d True

2-e True

3-a True

3-b False

3-c True

3-d True

3.6 ASSIGNMENTS

1. What is an Operator? Explain types of operators.
2. Explain the use of arithmetic operators with examples.
3. Explain how member operators work with examples.
4. What is type casting?
5. Explain the types of typecasting with examples.
6. Write a program:
    - That takes two numbers and prints their sum, difference, product, and quotient.
    - That checks if a number is greater than 100.
    - To determine if a person is eligible to vote (age >= 18 and citizen).
    - To swap two numbers using bitwise XOR.
    - To demonstrate the use of left and right shifts.

Block-2

Flow Control Statements and Functions

Unit-1: Control Flow and Conditional Statements

1

Unit Structure

1.0 LEARNING OBJECTIVE

After studying this unit student should be able to:

- Understand how to use decision making in program
- Use if statement to make a decision
- Use if..else statements to make a decision
- Use if..elif statements to make a decision
- Use nested conditional statements to make a decision

1.1 INTRODUCTION

So far, we have worked with python programs that follow a sequential structure, where instructions are executed one by one in the exact order as they appear in the program. When the program execution starts, first statement is executed, then second, third and so on. The program execution comes to an end as soon as the last statement is executed.

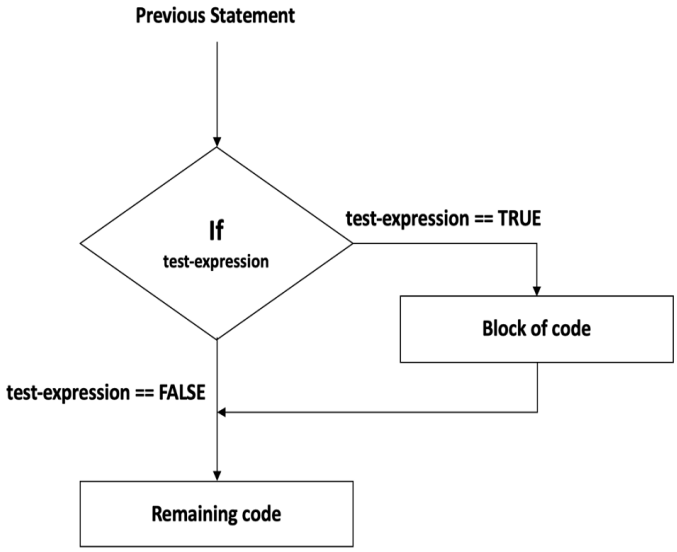
However, in real life applications, there are often scenarios where we want certain parts of the program to execute only when specific conditions are met. An example of such a scenario is to decide whether a person can vote in elections or not? The answer to this question is based on the age of the person. Hence we need to check the condition whether the age of the person is greater than or equals to 18 or not.

To handle such situations, it is necessary to change the flow of instructions within a program. Python language provides special types of statements known as conditional statements or decision structure statements. These statements allows us to navigate from one part of the program to another based on the outcome of certain conditions. Python language provides keywords like *if*, *elif* and *else* that can be used for decision making. In this chapter we will learn how to use such keywords.

1.2 if STATEMENT

The simplest type of conditional statement in Python can be created using the *if* keyword. The statement thus formed is commonly known as the *if* statement. It is used in programs for decision-making, as it enables us to alter the flow of program execution based on specific conditions.

The flow diagram of the *if* statement is shown in Figure 1.1 below:



The structure the if statement is mentioned below:

if (test-expression):

Block of code

We can execute a single statement or block of statements when the test expression is evaluated as TRUE. No statements under *if* statement are executed when the test expression is evaluated as FALSE. It is not compulsory to write the test expression in brackets. The test expression usually is used to compare two variables using relational operators like = = (equals to), != (not equals to), > (greater than), < (less than), >= (greater than equal to) and <= (less than equal to).

The example of how to use an if statement in a Python program is given in the program of Example 1.1.

Example 1.1: Decision making using if statement and fixed variable values

```
# Program to check whether coin 1 is less than coin 2 when coins have static values
coin1 = 5
coin2 = 10
if (coin1 < coin2):
 print("coin1 is less than coin2")
```

OUTPUT:

coin1 is less than coin2

In the above program the first line is known as comment. Here we have taken two variables named coin1 and coin2. We have assigned fixed integers values 5 and 10 to them respectively. Then using the test expression (*coin1 < coin2*), we check if the value of variable coin1 is less than the value of variable coin2. As can be seen the value of variable coin1 is 5 which is less than the 10 (value of variable in coin2), we print the message 'coin1 is less than coin2'. As we have fixed the values of both the variables we will get the same answer every time we execute this program.

Let us modify the above program such that the values of variables are entered by the user and may change every time. The modified Python program is given in Example 1.2.

Example 1.2: Decision making using if statement and values taken from user

```
# Program to check whether coin 1 is less than coin 2 using dynamic values
coin1 = int(input("Enter value of coin1: "))
coin2 = int(input("Enter value of coin2: "))
if (coin1 < coin2):
 print("coin1 is less than coin2")
```

OUTPUT – SCENARIO 1:

Enter value of coin1: 5

Enter value of coin2: 10

coin1 is less than coin2

OUTPUT – SCENARIO 2:

Enter value of coin1: 10

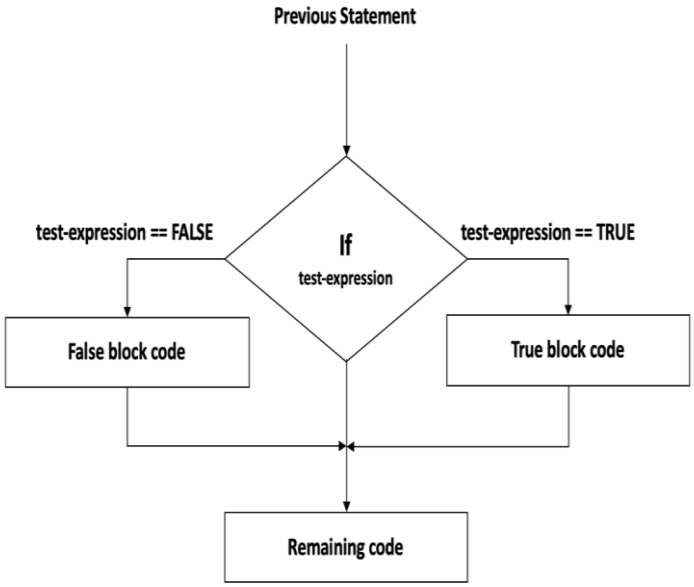
Enter value of coin2: 5

In the above program the values of variables coin1 and coin2 are to be given by users at the time of execution. Only integer numbers will be accepted as we have applied the constraint int on input. Then we compare the values in coin1 and coin2 using the test expression (*coin1 < coin2*). As can be seen in OUTPUT – SCENARIO 1 the user has entered integer value 5 in coin1 and 10 in coin2, as 5 is less than 10 we print the message 'coin1 is less than coin2'. When we run the program again as shown in OUTPUT – SCENARIO 2 the code under the if statement is not executed. This happened because the user entered integer value 10 in coin1 and 5 in coin2. As 10 is greater than 5 the test expression (*coin1 < coin2*) returned FALSE and thus no message was printed.

Note: The *if* statement can be written in single line as shown in the example below. Such statement is called *Short Hand if* statement.
if (coin1 < coin2): print("coin1 is less than coin2")

1.3 if..else STATEMENTS

The simple *if* statement has only one statement block, which gets executed when the test expression returns TRUE. What if we want to execute some other statement block if the test expression returns FALSE? In such cases, the combination of *if…else* statements is used. The flow diagram of the *if..else* statements is shown in Figure 1.2:



The structure the *if..else* statements is mentioned below:

if (test-expression):

True block code

else:

false block code

The example of how to use *if..else* statements in Python program is given in the program of Example 1.3.

Example 1.3: Decision making using if..else statements

```
# Program to check whether a person with specific age can vote or not.
age = int(input("Enter your age: "))
if (age >= 18):
 print("You can vote")
else:
 print("Sorry you will have to wait to vote")
```

OUTPUT – SCENARIO 1:

Enter your age: 20

You can vote

OUTPUT – SCENARIO 2:

Enter your age: 12

Sorry you will have to wait to vote

In the above program the user is prompted to enter the value of variable age. Then we compare the value entered using the test expression (*age >= 18*). As can be seen in OUTPUT – SCENARIO 1 the user has entered an integer value 20 in age, as 20 is greater than 18 we print the message 'You can vote'. When we run the program again as shown in OUTPUT – SCENARIO 2 the code under *else* statement is executed. This happened because the user has entered integer value 12 in age. As 12 is less than 18 the test expression (*age >= 18*) returned FALSE, thus the else statement block showing the message 'Sorry you will have to wait to vote' was executed.

Check Your Progress-1

a) The if statement is used for two way branching. (True/False)

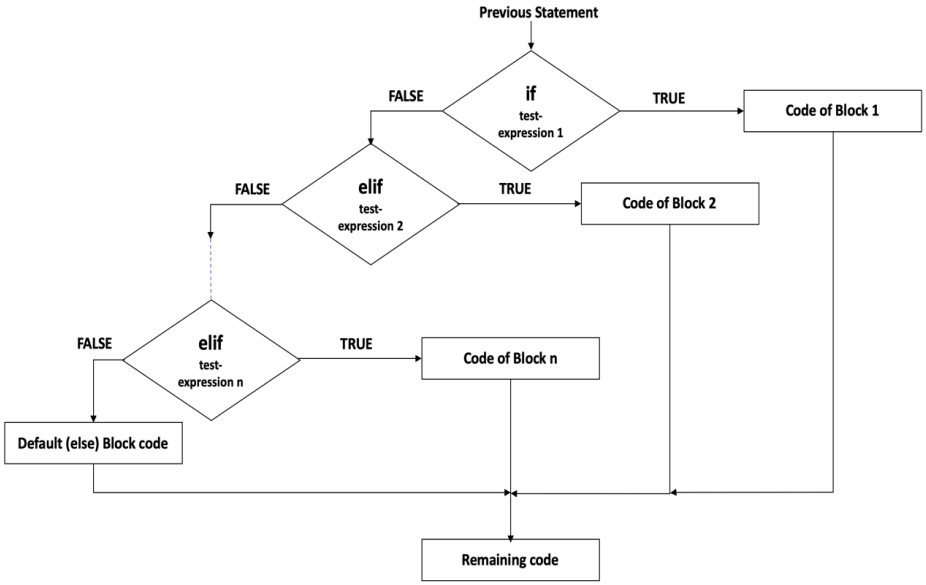
b) Given value of a=3 and b=5 the statement; if (a > b): print (b) will print 5. (True/False)

c) For two way decision making we can use if..elif statements. (True/False)

d) The condition statements return boolean values. (True/False)

1.4 if..elif STATEMENTS

The *if..else* statements allowed us to make two way decisions. In real life examples we need to evaluate multiple conditions one by one to solve a problem. To support such cases, the combination of *if…elif* statements is used. Here *elif* is a short form of *else if*. The flow diagram of the *if..elif* statements is shown in Figure 1.3:



The structure the *if..elif* statements is mentioned below:

if (test-expression 1):

Code of Block 1

elif (test-expression 2):

Code of Block 2

.

.

elif (test-expression n):

Code of Block n

else:

Default block code

The example of how to use *if..elif* statements in Python program is given in the program of Example 1.4.

Example 1.4: Decision making using if..elif statements

```
# Program to check what is the Grade of a student.
marks = int(input("Enter marks of student: "))
if (marks > 69):
 print("You have got A+ grade.")
elif (marks > 59):
 print("You have got A grade.")
elif (marks > 49):
 print("You have got B grade.")
elif (marks > 35):
 print("You have got C grade.")
else:
 print("You have got D grade.")
```

OUTPUT – SCENARIO 1:

Enter marks of student: 75

You have got A+ grade.

OUTPUT – SCENARIO 2:

Enter marks of student: 65

You have got A grade.

OUTPUT – SCENARIO 3:

Enter marks of student: 55

You have got B grade.

OUTPUT – SCENARIO 4:

Enter marks of student: 45

You have got C grade.

OUTPUT – SCENARIO 5:

Enter marks of student: 30

You have got D grade.

In the above program the user is prompted to enter the value of variable marks. Then we compare the value entered using test expression (*marks > 69*). As can be seen in OUTPUT – SCENARIO 1 the user has entered integer value 75 in marks, as 75 is greater than 69 we print the message 'You have got A+ grade.'. All the remaining conditions are ignored as soon as one of the test expressions is evaluated to TRUE. When we run the program again as shown in OUTPUT – SCENARIO 2 the code under the first *elif* statement is executed and so on. Observe that in OUTPUT – SCENARIO 5 the message under the else block is shown as none of the previous test expressions were evaluated to TRUE.

Note:

- **The *else* block in the *if..elif* statements is optional.**
- **The if..elif structure with multiple elif statement is also known as if..else ladder.**

1.5 NESTED CONDITIONAL STATEMENTS

Until now we saw programs wherein, we executed only a single statement after fulfilment of a condition. What happens if we need to execute multiple statements on fulfilment of a condition. Also, can one of these statements be a condition itself?

The program given in Example 1.4 though seems to work correctly, may give wrong outputs in cases where the user enters marks greater than 100 or less than 0. The Python program given in the program of Example 1.5 is a modified version of Example 1.4.

Example 1.5: Decision making using nested conditional statements

```
# Program to check what is the Grade of a student.
marks = int(input("Enter marks of student: "))
if (marks >= 0 and marks <= 100):
 # Statements to be executed when marks >= 0 and marks <= 100 is TRUE
 if (marks > 69):
  print("You have got A+ grade.")
 elif (marks > 59):
  print("You have got A grade.")
 elif (marks > 49):
  print("You have got B grade.")
 elif (marks > 35):
  print("You have got C grade.")
 else:
  print("You have got D grade.")
else:
        # Statement to be executed when marks >= 0 and marks <= 100 is
FALSE
 print("You have entered wrong marks.")
```

OUTPUT – SCENARIO 1:

```
Enter marks of student: 75
You have got A+ grade.
```

OUTPUT – SCENARIO 2:

```
Enter marks of student: 200
You have entered wrong marks.
```

OUTPUT – SCENARIO 3:

```
Enter marks of student: -10
You have entered wrong marks.
```

In the above program we have used the concept of nested conditional statements. The user is prompted to enter the value of variable marks. Then we compare the value entered using test expression (*marks >= 0 and marks*

<= *100*). The test expression used here has made use of the logical operator AND. It ensures that the value of marks should be in the range of 0 to 100.

As can be seen in OUTPUT – SCENARIO 1 the user has entered integer value 75 in marks, as 75 is greater than 0 and less than 100, we print the message 'You have got A+ grade.'. All the remaining conditions are ignored. When we run the program again as shown in OUTPUT – SCENARIO 2 and 3, the test expression (*marks >= 0 and marks <= 100*) evaluates to FALSE as the user has entered 200 and -10. Thus the message under the else block 'You have entered wrong marks.' is shown.

Note:

- **In Example 1.5 we have used if..elif statements under the if statement code block.**
- **This is known as nesting of if..elif under if statement**
- **It is possible to nest any conditional statement within each other.**

So far we have used only integer values as input in the programs. The text expression of conditional statements can work with float, string, list, tuple, dictionary, set or other data types too. You will learn about them in the coming chapters. Let us look at a python program that uses conditional statements with characters and numbers both. Example 1.6 shows uses of nested conditional statements.

Example 1.6: Decision making using nested conditional statements

```
# Program to check whether entered value is character or number.
# If the value entered is character then check whether it is vowel or consonant.
# If the value entered is positive integer then check whether it is even or odd.
data = input("Enter single character or positive integer: ")
if (data.isalpha() and len(data) == 1):
 # Statements to be executed when data is character
 if (data == 'a' or data == 'e' or data == 'i' or data == 'o' or data == 'u'):
```

```
  print("You have entered a vowel.")
 elif (data == 'A' or data == 'E' or data == 'I' or data == 'O' or data == 'U'):
  print("You have entered a vowel.")
 else:
  print("You have entered a consonant.")
elif (data.isdigit()):
        # Statement to be executed when data is number
 if (int(data) % 2 == 0):
  print("You have entered an even number.")
 else:
  print("You have entered an odd number.")
else:
 print("You have not entered a single character or positive integer.")
```

OUTPUT – SCENARIO 1:

Enter single character or positive integer: e

You have entered a vowel.

OUTPUT – SCENARIO 2:

Enter single character or positive integer: q

You have entered a consonant.

OUTPUT – SCENARIO 3:

Enter single character or positive integer: test

You have not entered a single character or positive integer.

OUTPUT – SCENARIO 4:

Enter single character or positive integer: 5

You have entered an odd number.

OUTPUT – SCENARIO 5:

Enter single character or positive integer: 18

You have entered an even number.

OUTPUT – SCENARIO 6:

Enter single character or positive integer: 35.6

You have not entered a single character or positive integer.

The above program checks whether the entered value is character or number. If the value entered is character then we further check whether the character is vowel or consonant. If the value entered is positive integer then we check whether it is even or odd. In all other cases we print a message 'You have not entered a single character or positive integer.'. Here the test expression *(data.isalpha() and len(data) == 1*) ensures that the value entered is a single digit alphabet in upper or lower case. The test expressions *(data == 'a' or data == 'e' or data == 'i' or data == 'o' or data == 'u')* and *(data == 'A' or data == 'E' or data == 'I' or data == 'O' or data == 'U')* ensures that value entered is vowel. The test expression *(data.isdigit())* ensures that the value entered is a positive integer. Observe that we have nested *if..elif* statements under *if* block and *if..else* statements under *elif* block.

Check Your Progress-2

a) For multiway way decision making we can use if..elif statements (True/False)
b) The else block in if..elif is compulsory. (True/False)
c) The statement block under conditional statements can have multiple statements. (True/False)
d) The text expression of a conditional statement can work with integer values only. (True/False)
e) If “bat” == “BAT” is a valid conditional statement. (True/False)

1.6 LET US SUM UP

In this unit we have discussed the concept of decision making. You have got a detailed understanding of how to use different conditional statements like if, if..else and if..elif. We also understood the concept of using conditional statements within conditional statements; thus learnt how nesting of conditions can be done.

1.7 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

1-a False
1-b False
1-c True
1-d True
2-a True
2-b False
2-c True
2-d False
2-e True

1.8 ASSIGNMENTS

- What is the disadvantage of using conditional statements?
- Explain the use of the if…else statement.
- What do you mean by nested if statement?
- State the importance of using if..elif statement.
- Write a program to perform the following activities:
    - Check whether the given number is positive or negative.
    - Calculate value of expression I = (P * R * N) / 100, only when P, R and N have values greater than 0.
    - Enter the value of two numbers, num1 and num2. Find whether num1 is less than, greater than or equal to num2.
    - Enter the value of three numbers, num1, num2 and num3. Find out which number is the greatest of the three.
    - Enter two names, name1 and name2. Check whether the names are same or not.

Unit-2: Loop Control Structures

2

Unit Structure

2.0 LEARNING OBJECTIVE

After studying this unit student should be able to:

- Understand how to use repetition in the program
- Use while statement to perform repetition
- Use for statement to perform repetition
- Understand the concept of nested loops
- Use nested loops to perform repetition

2.1 INTRODUCTION

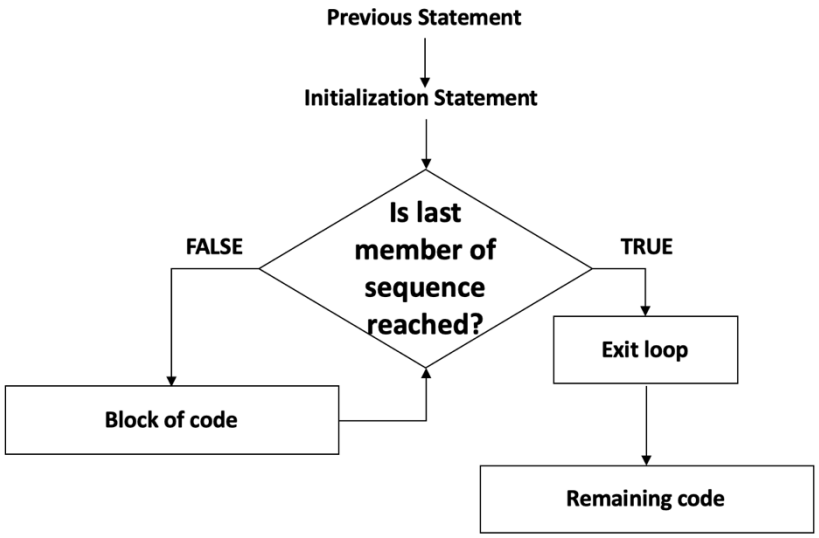
In the previous chapter we worked with python programs that allowed us to use decision making in our programs. In many real life applications, there are often scenarios where we want to execute a block of code multiple times based or certain decisions or unconditionally. All programming languages offer loop control structures that allow programmers to execute a statement or set of statements multiple times.

When looping is implemented in a program, the a statement or set of statements are executed multiple times until some defined condition is satisfied. The looping construct is composed of two parts: a control statement and body of the loop.

Python language provides keywords like *for* and *while* that can be used for performing repetition in the programs. In this chapter we will learn how to use such keywords.

2.2 for STATEMENT

The *for* statement generally known as *for loop* is used in the program when we want to repetitively execute a statement or set of statements for a fixed number of times. The *for* loop is used for iterating over a sequence of values in a list, a tuple, a dictionary, a set, a string or integer values. In this chapter we will see how to use a for loop with integers. The flow diagram of the generic *for* statement is shown in Figure 2.1:

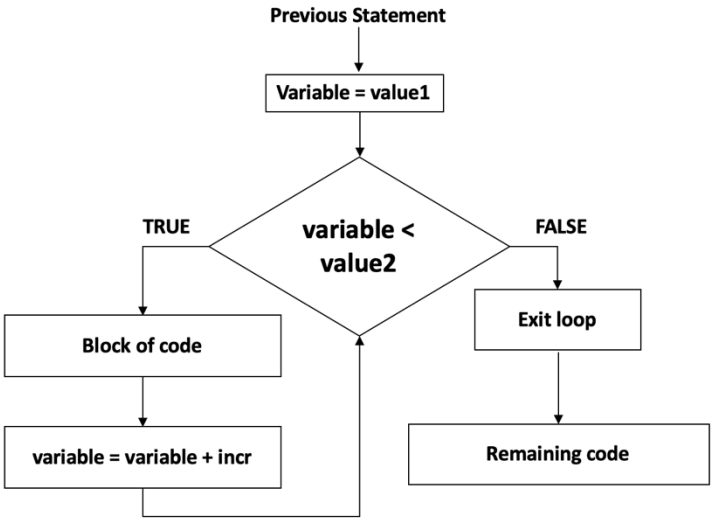


The general structure of the *for* statement is mentioned below:

for variable in range([value1], value2, [incr]):

Block of code

In the above structure range function has been used. Here *value1* is the start value of the variable. If it is not specified as it is optional the initial value will be equal to 0 (zero). The term *value2* is the end value of the variable excluding *value2*. The term *incr* specifies the value to be added to a variable in each iteration. The square brackets indicate that this value is optional too. If not specified the value 1 is added to the variable by default. The flow diagram of the *for* statement that uses integers is shown in Figure 2.2:



The example of how to use a for statement in a Python program is given in the program of Example 2.1.

Example 2.1(a) and 2.1(b): Print a series of number using for loop

# Program 2.1(a) *#Print series of number between 1 to 5* *for num in range(1, 5):* *print(num)*	*# Program 2.1(b)* *#Print series of number between 0 to 5* *for num in range(5):* *print(num)*
OUTPUT: 1 2 3 4	**OUTPUT:** 0 1 2 3 4

In the above program 2.1(a) we have taken a variable *num*. Then in the *for* loop we have given a range of 1 to 5 (value1 = 1 and value2 = 5). Here the increment value has not been specified. Thus initially variable *num* would be assigned with integer value 1 and with every iteration integer value 1 will be added to the existing value of *num*.

Program 2.1(b) is also printing a series of numbers. Here we have given only one value 5. This value is by default assigned to value2. The default starting value thus becomes 0 and the default increment value is 1. Thus we get 0 to 4 as output. Observe that the last value printed in both programs is 4 and not 5.

Note:

- **The *for* loop is also known as counter controlled loop.**
- **If value1 is not specified its default value is 0 (zero).**
- **If incr is not specified its default is 1.**
- **If value1 is specified and is greater than value2, then loop will not be executed.**

Let us look at another Python program given in Example 2.2 that uses a for loop to print the odd numbers between 1 to 10.

Example 2.2: Print odd numbers between 1 to 10 using for loop

```
# Program to print odd numbers between 1 to 10
for num in range(1, 10, 2):
  print(num)
```

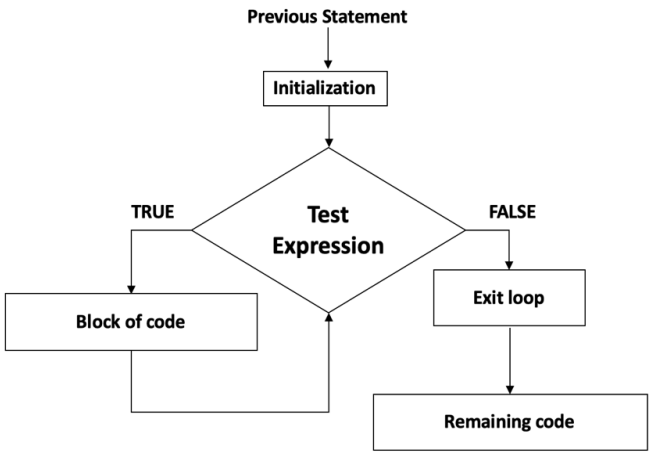
OUTPUT:

```
1
3
5
7
9
```

In the above program we have taken a variable *num*. Then in the *for* loop we have given a range of 1 to 10 (value1 = 1 and value2 = 10). Here the increment value has been given as 2. Thus initially variable *num* would be assigned with integer value 1 and with every iteration integer value 2 will be added to the existing value of *num*.

2.3 while STATEMENT

With the *for* loop we executed instructions for an already known number of times. What if we do not know the exact number of times we want to execute an instruction? In such cases, the *while* statement is used. The flow diagram of the *while* statement is shown in Figure 2.3:



The structure the *while* statement is mentioned below:

while (test-expression):

Code block

The example of how to use a while statement in a Python program is given in the program of Example 2.3.

Example 2.3: Program to find sum of entered numbers

```
# Program to print sum of entered numbers
number = int(input("Enter a number: "))
sum = number
while (number != 0):
  number = int(input("Enter a number: "))
  sum = sum + number
print(f"Sum = {sum}")
```

OUTPUT – SCENARIO 1:

```
Enter a number: 2
Enter a number: 3
Enter a number: 0
Sum = 5
```

OUTPUT – SCENARIO 2:

```
Enter a number: 0
Sum = 0
```

In the above program the user is prompted to enter the value of the variable *number*. This number is then assigned to the variable *sum*. Then we compare the test expression (*number != 0*) to check whether the value of the number is 0 or not. If it is not 0 (zero) we enter inside the *while* loop. The body of the loop contains two instructions. The first instruction again requires us to enter a value in a variable *number*. The value of the *number* is then added into the previous value of variable *sum*. The execution control will now again go back to test expression and evaluate it. If the test expression is evaluated to TRUE the instructions in the while loop will be executed again. If the test expression is evaluated to FALSE, the statement after the while loop will be executed.

As can be seen in OUTPUT – SCENARIO 1; the user has entered integer value 2 in variable *number*, thus the variable *sum* will be assigned value 2. Now as 2 is not equals to 0, we enter the while loop. The new value of the *number* is 3, thus the sum will now become 5. Also as 3 is not equals to 0, we once again enter into the *while* loop. Now the user has entered 0, thus the value of sum remains 5. When the control is transferred to test expression, we get expression (0 != 0), which results in FALSE. Thus the while loop ends and we get the output as 5. Observe that the while loop was executed two times.

When we run the program again as shown in OUTPUT – SCENARIO 2; the user has entered integer value 0 in the variable *number*, thus the variable *sum* will be assigned value 0. Now for test expression, we get expression (0 != 0), which results in FALSE. Thus the while loop ends and we get the output as 0. Observe that the while loop was not executed even once in the scenario 2.

Note:

- **As we check the test expression before entering the loop, while loop is also known as entry-controlled loop.**
- **print(f"Sum = {sum}") is used to concatenate a message "Sum =" and value of variable given in curly brackets, here {sum} .**

The two loop constructs *for* and *while* can be used interchangeably. Let us re-write the program given in Example 2.2 using a while loop. The modified Python program is given in Example 2.4

Example 2.4: Print odd numbers between 1 to 10 using while loop

```
# Program to print odd numbers between 1 to 10 using while loop
num = 1
while (num < 10):
  print(num)
  num = num + 2
```

OUTPUT:

```
1
3
5
7
9
```

In the above program the variable *num* is assigned integer value 1. Then we compare the test expression (*num* < *10*) to check whether the value of *num* is less than 10 or not. If the test expression evaluates to TRUE we print the value of variable *num* and add an integer value 2 to the existing value of *num*. This process goes on till the value of variable *num* becomes equal to or greater than 10. The given program is going to always execute for five times.

Activity 2.1:

Modify the program given in Example 2.4 to ask an end number from the user. Then print all odd numbers starting from 1 till the end number. For example if the user enter end number as 20 then program should print odd numbers between 1 to 20.

Check Your Progress-1

a) In for loop it is compulsory to specify increment value. (True/False)
b) The for loop will execute only if value1 is less than value2. (True/False)
c) The for loop will execute till the value of variable becomes equal to value2. (True/False)
d) The for loop is also known as counter controlled loop. (True/False)
e) The while loop is executed only when the test expression is evaluated to FALSE. (True/False)
f) The while loop may or may not be executed even once based on the value of test expression. (True/False)
g) The while loop is also known as entry controlled loop. (True/False)

2.4 SKIPPING PART OF LOOP

Sometimes during the execution of a program we may want to terminate a loop without executing some instructions. Or at times we may want to skip some of the instructions but still continue iterations of the loop. Python provides three keywords; *break, continue* and *pass* to handle such cases.

The *break* keyword:

The *break* keyword is used to terminate the loop, when encountered within the loop it stops further execution of the loop. It can be used with both *for* and *while* loop.

The example of how to use *break* keyword in a Python program is given in the program of Example 2.5.

Example 2.5: Program showing use of break

```
# Program to check whether entered number is odd
# Shows use of while loop and break
while (1):
  num = int(input("Enter a number: "))
  remainder = num % 2
  if (remainder == 0):
   print(f"{num} is an even number. While loop ends here.")
   break
  else:
   print(f"{num} is an odd number\n")
```

OUTPUT – SCENARIO 1:

```
Enter a number: 3
3 is an odd number

Enter a number: 5
5 is an odd number

Enter a number: 6
6 is an even number. While loop ends here.
```

OUTPUT – SCENARIO 2:

Enter a number: 4

4 is an even number. While loop ends here.

In the above program the program execution starts from the line *while(1)*. As the test expression has value 1, it will always evaluate to TRUE. Now, the first statement under the while loop will be executed. The user is prompted to enter the value of variable *num*. Then the variable *remainder* assigned a value using expression *num % 2*. We have then used the *if* statement to decide whether the loop should be continued or stopped. The test expression in *if* checks whether the value of variable *remainder* is 0. In case the value is 0, we print a message and exit the while loop. The while loop is continued otherwise until the user enters an even number.

As can be seen in OUTPUT – SCENARIO 1; the while loop is executed three times. When the user enters integer value 3 in *num*, we get a message "3 is an odd number" and the loop continues. When the user enters integer value 5 in *num*, we get a message "5 is an odd number" and the loop continues. When the user enters integer value 6 in *num*, we get a message "6 is an even number. While loop ends here."

When we run the program again as shown in OUTPUT – SCENARIO 2; we get the message "4 is an even number. While loop ends here."; in the first instance only. There is no further repetition of the while loop.

Note:

- **The while loop implemented in Example 2.5 is known as exit control loop as we decided to continue or stop the loop at the time of exit.**
- **In most cases the break statement will be used along with if or if..else statement.**
- **If a break statement is used without an if statement, then the loop will terminate immediately.**

- **The test expression in loops needs to be used carefully. Using wrong test expressions may lead to infinite loops.**
- **Infinite loops are the loops that go on executing as the test expression never evaluates to FALSE.**

The *continue* keyword:

The *continue* keyword is used to skip some instructions within the loop and go back to check the test expression. The keyword *continue* can also be used *for* and *while* loop.

Let us modify the program in Example 2.5 to ensure that only positive numbers are checked when considering odd or even numbers. Example 2.6 shows the modified program.

Example 2.6: Program showing use of continue keyword

```
# Program to check whether entered number is odd
# Shows use of while loop, continue and break

while (1):
  num = int(input("Enter a number: "))
  if (num <= 0):
   continue
  remainder = num % 2
  if (remainder == 0):
   print(f"{num} is an even number. While loop ends here.")
   break
  else:
   print(f"{num} is an odd number\n")
```

OUTPUT – SCENARIO 1:

```
Enter a number: -3
Enter a number: 3
3 is an odd number
Enter a number: 4
4 is an even number. While loop ends here.
```

OUTPUT – SCENARIO 2:

Enter a number: -1

Enter a number: 0

Enter a number: 2

2 is an even number. While loop ends here.

The above program is performing the same operation as the program in Example 2.5. A minor change has been done in this program though. After accepting the value of variable *num* from the user we have used the *if* statement to decide whether the flow control of loop should come back to test expression (1) or the execution is continued or stopped.

As can be seen in OUTPUT – SCENARIO 1; the while loop is executed three times. When the user enters integer value -3 in *num*, we are asked to enter the value of *num* again. When the user enters integer value 3, we get a message "3 is an odd number" and the loop continues. When the user enters integer value 4 in *num*, we get a message "4 is an even number. While loop ends here."

When we run the program again as shown in OUTPUT – SCENARIO 2; we are asked to enter the value of *num* again in both cases when the user entered -1 and 0. When the user enters integer value 2 in *num,* we get message "2 is an even number. While loop ends here."; There is no further repetition of the while loop.

The *pass* keyword:

The *pass* keyword is used to do nothing within the loop. The keyword *pass* can also be used *for* and *while* loop. Example 2.7 shows the use of the pass keyword.

Example 2.7: Program showing use of pass keyword

```
#Program showing example of pass
for num in range(10, 15):
    if num % 2 == 0:
     pass # Do nothing for even numbers
    else:
     print(f"{num} is odd")
```

OUTPUT:

11 is odd

13 is odd

In the above program a *for* loop that starts from value 10 to 15 has been used. Here for values 10, 12 and 14 the pass statement is executed, effectively skipping any action. When *num* is odd (11,13), the program prints the message indicating the number is odd.

2.5 NESTED LOOPS

The for and while statements are used for repetitions of certain statements. Thus, it is possible to use these statements as part of loops themselves. A nested loop is created when a for or while statement is used within a for or while loop. The program given in Example 2.8 shows an example of using nested loops.

Example 2.8: Example of using nested loops

```
#Program showing example of nested loop
num = int(input("Enter a number: "))
for o_ctr in range(1, num):  # Outer loop
 for i_ctr in range(1, 11):  # Inner loop
   product = o_ctr * i_ctr
   print(f"{o_ctr} * {i_ctr} = {product}")
 print("\n")  # New line after each row
```

OUTPUT:

```
Enter a number: 3
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5
1 * 6 = 6
1 * 7 = 7
1 * 8 = 8
1 * 9 = 9
1 * 10 = 10
```

```
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10
2 * 6 = 12
2 * 7 = 14
2 * 8 = 16
2 * 9 = 18
2 * 10 = 20
```

In the above program we have used the concept of nested loops. The user is prompted to enter the value of variable *num*. This value is used as the maximum range of the outer loop. The inner loop is predefined to run for ten times starting from 1 to 10.

As can be seen in OUTPUT; the user has entered integer value 3 in *num*. Thus the outer loop is executed two times for o_ctr value 1 and 2. For every instance of outer loop the inner loop is executed ten times from i_ctr value changing from 1 to 10.

Let us have a look at another example that shows the use of nested loops. The program given in Example 2.9 shows an example of using a nested loop to draw a pattern.

```
# Number of rows in the triangle
rows = int(input("Enter number of rows: "))
if (rows > 1):
 o_ctr = 1
 while (o_ctr < rows + 1):
    for i_ctr in range(o_ctr):
        print('*', end=' ')
    o_ctr = o_ctr + 1
    print("\n")  # Move to the next line after finishing one row
else:
 print("Rows should be greater than 1\n")
```

OUTPUT – SCENARIO 1:

```
Enter number of rows: 1
Rows should be greater than 1
```

OUTPUT – SCENARIO 2:

```
Enter number of rows: 4
*
* *
* * *
* * * *
```

In the above program we have used both *while* and *for* loops. The user is prompted to enter the value of variable *rows*. If the value of *rows* is greater than 1, then the nested loop is executed. As can be seen in OUTPUT – SCENARIO 1; as the user has entered 1 we get the message “Rows should be greater than 1”. As seen in OUTPUT – SCENARIO 2; when the user enters 4 we get a pattern of stars.

Check Your Progress-2

a) The break statement stops execution of a loop. (True/False)
b) The break statement cannot be used in the for loop. (True/False)
c) The continue statement skips some statements of the loop and comes out of the loop. (True/False)
d) The pass statement when used in a loop does no activity. (True/False)
e) To avoid creating infinite loops it is better to use continue keyword along with if or if..else statements. (True/False)
f) It is possible to use a for loop within a while loop and vice versa. (True/False)

2.6 LET US SUM UP

In this unit we have discussed the concept of loops. You have got a detailed understanding of how to use loops like *for* and *while*. We further learnt how to use keywords like break, continue and pass if some statements in the loop need to be skipped. We also understood the concept of using loop within a loop.

2.7 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

1-a False

1-b True

1-c False

1-d True

1-e False

1-f True

1-g True

2-a True

2-b False

2-c False

2-d True

2-e True

2-f True

2.8 ASSIGNMENTS

- Explain the purpose of using a loop in your programs.
- What is the difference between entry controlled and exit controlled loop?
- Explain how we can skip a part of a loop using break and continue statements.
- Explain the concept of nested loop with appropriate examples.
- Write a program to perform following activities:
  - Find the factorial of a number using a loop.
  - Find the sum of the first 50 even numbers.
  - Print a Fibonacci series till given position. A Fibonacci series starts with 0. If the user enters 4, the program should print 0, 1, 1, 2. If the user enters 7, the program should print 0, 1, 1, 2, 3, 5, 8.
  - Find the sum of the first 5 square numbers.
  - Print the following patterns for a given number of rows as mentioned:

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

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

Unit-3: Functions in Python 3

Unit Structure

3.0 LEARNING OBJECTIVE

After studying this unit student should be able to:

- Understand what is function
- Use a function in a program
- Understand different types of arguments used in function
- Understand what is anonymous function
- Understand and use recursive function in a program
- Understand the scope of variables

3.1 INTRODUCTION

Till now we worked with small python programs that allowed us to perform a specific task. The real life applications though have a very high number of lines. In such cases it is better to break the code into smaller parts.

Python language provides us a concept called functions to assist converting long codes into smaller parts. In this chapter we will learn what a function is, which are the different types of functions. We will also learn how to use a function in a Python program.

3.2 FUNCTION AND ITS TYPE

A function in Python is a block of code that contains a set of statements that can be reused to perform a specific task. A function once defined helps us to organize a program, make it more readable, and allow reuse of code. There are two categories of functions in Python; Built-in and User defined.

Built-in Functions

A built-in function in Python is a function that is readily available for use without the need for any additional imports or definitions. These functions are part of Python's standard library and provide essential functionalities that help programmers perform common tasks efficiently. As these functions are part of Python's standard library they are also known as library functions. In previous chapters we have already used a few built-in functions like *print* (to print on screen), *input* (to take data from user), *int* (to convert entered data to integer

value). Python has a rich collection of built-in functions. To get the details of these functions we need to explore the builtin module. We will be using various built-in functions in the chapters to come.

User defined Functions

A function created by the programmer (user) is known as a *user defined* function. These functions are created to help programmers perform some specific tasks efficiently. Once a function has been created it can be reused multiple times within a program.

A function after its execution is completed may return a value or may return nothing. The general syntax of a function is as follows:

def function-name(P1 : datatype,…,Pn : datatype):

statement 1

:

statement n

return expression

Here *def* is a keyword that is used to define a function. The *function-name* here represents a unique name of the function. *P1… Pn* are known as input parameters, each parameter needs to have a data type. Specifying parameters is not compulsory, hence we may have functions without parameters also. Statement 1 to Statement n combined is known as *function body*. The last line *return expression* is optional. If specified it states that after execution of the function body, the value stored in the expression will be returned back by the function. The Python program given in Example 3.1 shows how to create a function that does not return anything.

Example 3.1: Program to check whether a number is even or odd

```
#Program using a function to check whether a number is even or odd
# Function EvenOdd
def EvenOdd(num:int):
    if (num % 2 == 0):
        print(f"{num} is even.")
```

```
    else:
        print(f"{num} is odd.")


# Program Logic
number = int(input("Enter a number: "))
if (number > 0):
  EvenOdd(number)    #Function call statement
else:
  print(f"{number} is negative.")
print("Welcome to the world of Python functions.")
```

OUTPUT– SCENARIO 1:

Enter a number: 2

2 is even.

Welcome to the world of Python functions.

OUTPUT– SCENARIO 2:

Enter a number: 45

45 is odd.

Welcome to the world of Python functions.

OUTPUT– SCENARIO 3:

Enter a number: -10

-10 is negative.

Welcome to the world of Python functions.

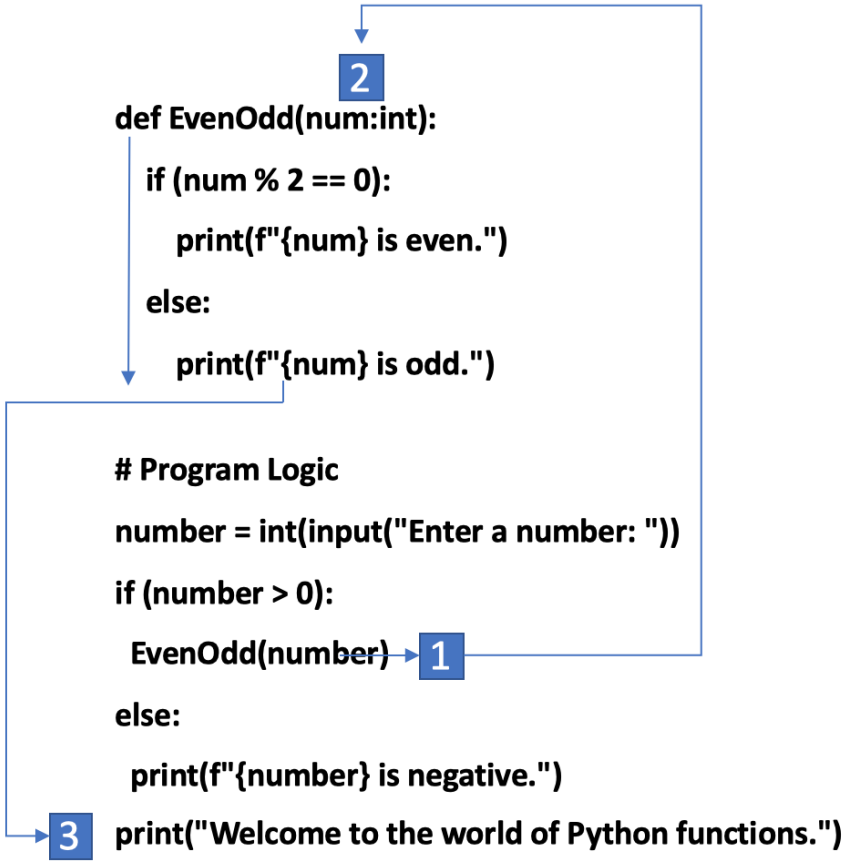
In the above program we have defined a function *EvenOdd* that accepts an integer parameter *num*. The function is used to check whether the remainder of num%2 is zero or not. If the remainder is zero we print a message indicating that the number is even. Otherwise we print a message indicating that the number is odd.

The function *EvenOdd* on its own will not be able to perform any operation. To execute the instructions defined in the function, it needs to be called. Observe that under the line Program Logic we have accepted a value of variable *number* from the user. We have then checked whether the value is greater than zero, if

so we have called the function and passed the value of the number in it. Refer statement *EvenOdd(number)* this line is known as a *function call*.

As can be seen in OUTPUT – SCENARIO 1; the user has entered integer value 2 in variable *number*, as it was greater than zero a function call *EvenOdd(2)* was executed and we got messages “2 is even.” and “Welcome to the world of Python functions.”. When we run the program again as shown in OUTPUT – SCENARIO 2; the user has entered integer value 45 in variable *number*, as it was greater than zero a function call *EvenOdd(45)* was executed and we got messages “45 is odd.” and “Welcome to the world of Python functions.”. In OUTPUT-SCENARIO 3; the user entered integer value -10 in variable *number*, as it is less than zero, function was not called and we got messages “-10 is negative.” and “Welcome to the world of Python functions.”.

Figure 3.1 shows the control and data flow of the program when a function call is made with the value of number.



Observe the sequence of actions in the Figure 3.1:

1. When the function EvenOdd(number) is called, the program's control transfers to the function definition and assigns the value of number to num.
2. All the code inside the function EvenOdd() is executed.
3. The control of the program jumps to the next statement after the function call.

Let us now write a program that uses a function that returns a value back. Example 3.2 shows such a program.

```
#Program using a function to calculate factorial of a number
def factorial(num):
    result = 1
    if (num > 1):
     for ctr in range(1, num + 1):
        result *= ctr
    return result
# Program Logic
number = int(input("Enter a number: "))
if (number > 0):
 fact = factorial(number)    #Function call statement
 print(f"The factorial of {number} is {fact}.")
else:
 print(f"{number} is negative. Factorial is not defined for negative numbers.")
print("Welcome to the world of Python functions.")
```

OUTPUT– SCENARIO 1:

```
Enter a number: 6
The factorial of 6 is 720.
Welcome to the world of Python functions.
```

OUTPUT– SCENARIO 2:

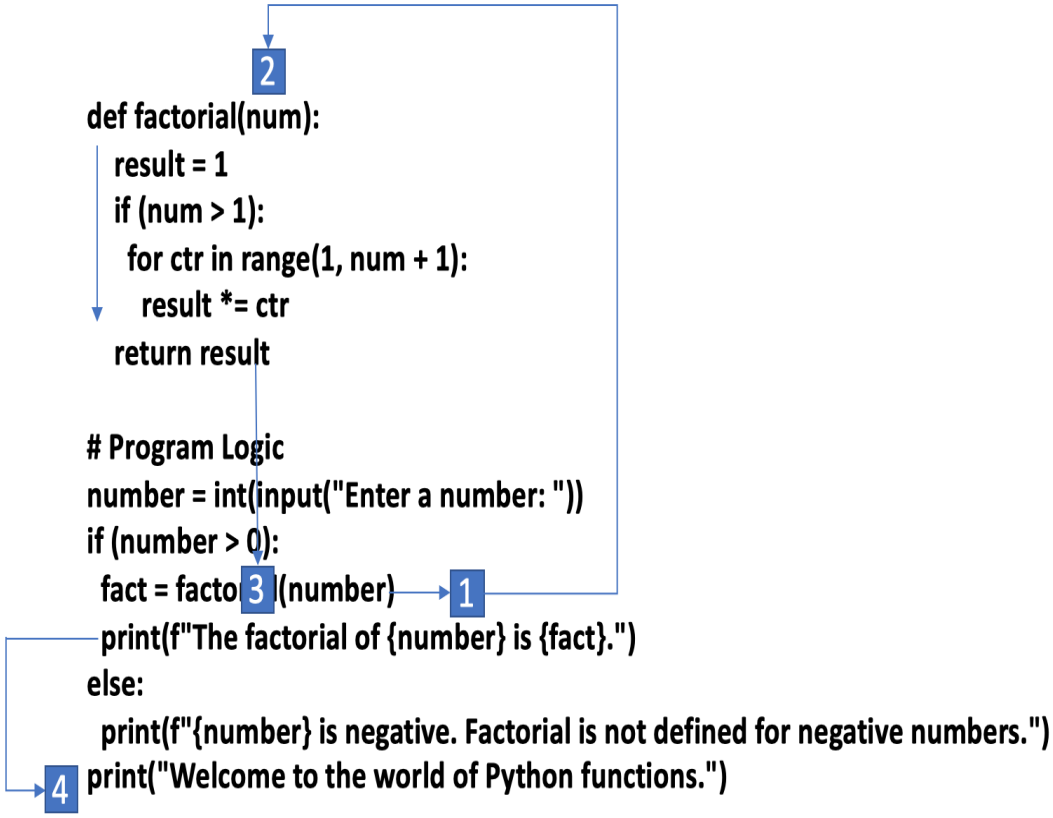
```
Enter a number: -10
-10 is negative. Factorial is not defined for negative numbers.
Welcome to the world of Python functions.
```

In the above program we have defined a function *factorial* that accepts parameter *num*. The function is used to return the factorial of parameter *num*. Initially a variable *result* with value 1 is defined. Then we check whether the value of *num* is greater than 1 or not. If value is 1 we return the default value of result i.e. 1. Otherwise we calculate the factorial of *num* using the *for* loop and return the new value of *result*.

Under the line Program Logic we have accepted a value of variable *number* from the user. We have then checked whether the value is greater than zero, if so we have called the function and passed the value of the *number* in it (refer statement *fact = factorial (number)*). Observe that as the function factorial is returning a value we have called the function and assigned its value to variable *fact*. If required we can now perform further operations with the value stored in variable *fact*.

As can be seen in OUTPUT – SCENARIO 1; the user has entered integer value 6 in variable *number*, as it was greater than zero a function call *factorial(6)* was executed and we got messages “The factorial of 6 is 720.” and “Welcome to the world of Python functions.”. When we run the program again as shown in OUTPUT – SCENARIO 2; the user has entered integer value -10 in variable *number*, as it was less than zero, function was not called and we got messages “-10 is negative. Factorial is not defined for negative numbers.” and “Welcome to the world of Python functions.”.

Figure 3.2 shows the control and data flow of the program when a function call is made with the value of number.



Observe the sequence of actions in the Figure 3.2:

1. When the function factorial(number) is called, the program's control transfers to the function definition and assigns the value of number to num.
2. All the code inside the function factorial() is executed.
3. The value of result is returned back in the statement where the function call is made. The value is then assigned to variable *fact*.
4. The control of the program jumps to the next statement after the function call.

Note:

- **A function needs to be defined before it can be called.**
- **Specifying the data types of parameters is not compulsory.**
- **The return statement is optional.**
- **It is possible to specify the return type of function also.**
- **To specify data type of the return value the def line will become: def function-name(P1 : datatype,…,Pn : datatype) return_type:**

- **A function can be defined in four ways:**
  - **A function with no parameters and no return value**
  - **A function with no parameters and a return value**
  - **A function with parameters and no return value**
  - **A function with parameters and a return value**
- **Example 3.1 was an example of a function with parameters and no return value**
- **Example 3.2 was an example of a function with parameters and a return value**

Check Your Progress-1

a) Function allows us to reuse the code. (True/False)
b) To use a built-in function, we need to first define it. (True/False)
c) To create a user defined function we need to use keyword def. (True/False)
d) A function will always return a value. (True/False)
e) Specifying data type of a return value is compulsory in Python. (True/False)
f) A user defined function can be called even before it is defined. (True/False)

3.3 TYPE OF PARAMETERS

By now we know that there are two aspects of function; first is *function definition* and second is *function call*. Function definition allows us to write a logic to perform a specific task, while function call allows us to use this logic in our program. Functions in Python can accept parameters also known as arguments in various forms, allowing for flexibility and readability in code. The arguments in Python can be classified four types as mentioned:

- Positional Arguments
- Keyword Arguments
- Default Arguments
- Variable length Arguments

Positional Arguments

The positional arguments are the most common type of arguments. They are passed to the function in the order they are defined. The final result of the function will always depend on the order in which the parameters are passed. Example 3.3 shows the use of positional parameters.

Example 3.3: Program to show use of positional parameter

```
#Program to show use of positional parameter
def subtract(num1, num2):
    return num1 - num2
result = subtract(5, 8)  # num1=5, num2=8
print(f"Subtraction of 5 and 8 is {result}.")

result = subtract(8, 5)  # num1=8, num2=5
print(f"Subtraction of 8 and 5 is {result}.")
```

OUTPUT:

```
Subtraction of 5 and 8 is -3.
Subtraction of 8 and 5 is 3.
```

In the above program we have defined a function *subtract* that accepts two numbers.

Observe that in the program we have made two calls to the function;

subtract(5, 8)

and *subtract(8, 5)*. Here when the first function call *subtract(5, 8)* is made then *num1* is assigned value 5 and *num2* is assigned value 8. We thus get the output message “Subtraction of 5 and 8 is -3.” When the second call *subtract(8, 5)* is made then *num1* is assigned value 8 and *num2* is assigned value 5. We thus get the output message “Subtraction of 8 and 5 is 3.”

As can be observed from the output the values passed in the function call are mapped based on the position of the parameters in function definition.

Keyword Arguments

The keyword arguments are passed using the name of the parameter defined in the function. This allows users to specify values without worrying about their order while calling the function. Example 3.4 shows the use of keyword arguments.

Example 3.4: Program to show use of keyword parameter

```
#Program to show use of keyword parameter
def subtract(num1, num2):
    return num1 - num2

a = int(input("Enter first number: "))
b = int(input("Enter second number: "))

result = subtract(num1=a, num2=b)
print(f"Subtraction result is {result}.")

result = subtract(num2=b, num1=a)
print(f"Subtraction result is {result}.")
```

OUTPUT:

```
Enter first number: 8
Enter second number: 3
Subtraction result is 5.
Subtraction result is 5.
```

In the above program we have used the same *subtract* function as that of Example 3.3. Further values of two numbers have been accepted in variable *a* and *b*. When the first function call *subtract(num1=a, num2=b)* is made then *num1* is assigned value of variable *a* (8 in this case) and *num2* is assigned value of variable *b* (3 in this case). We thus get the output message “Subtraction result is 5.” When the second call *subtract(num2=b, num1=a)* is made we still get the same output message “Subtraction result is 5.”

The above output shows that the position of an argument does not matter if the keyword of the argument is specified. The keyword ensures that the value passed will go to the same parameter in function irrespective of its position when function is called.

Default Arguments

While defining a function it is possible to assign a default value to a parameter. If the user does not provide a value for such a parameter during the function call then the default value is used. If the user provides a value then the new value is used. Example 3.5 shows the use of default arguments.

Example 3.5: Program to show use of default parameter

```
# Program to show use of default parameter
def multiply(num1, num2=10):
    return num1 * num2

a = int(input("Enter first number: "))
b = int(input("Enter second number: "))

result = multiply(a). # call with only one parameter
print(f"Multiplication result is {result}.")
result = multiply(a, b)
print(f"Multiplication result is {result}.")
```

OUTPUT:

```
Enter first number: 25
Enter second number: 5
Multiplication result is 250.
Multiplication result is 125.
```

In the above program we have used a function *multiply*, that accepts two parameters *num1* and *num2*. The parameter *num2* has been assigned a default value 10. Further values of two numbers have been accepted in variable *a* and *b*. When the first function call *multiply(a)* is made then num1 is assigned value of a (25 in this case), as the value of num2 is not specified it is assigned default

value (10 in this case). We thus get the output message “Multiplication result is 250.” When the second call *multiply(a,b)* is made then *num1* is assigned value of variable *a* (25 in this case), as the value of *num2* is assigned value of variable *b* (5 in this case). We thus get the output message “Multiplication result is 125.” Observe that when a new value has been specified the default value is overwritten.

Variable length Arguments

Many times it may happen that we do not know in advance how many arguments will be passed to the function. To handle such situations Python provides use of *args for non-keyword arguments and **kwargs for keyword arguments. The parameter *args, also known as arbitrary positional arguments, allows us to accept multiple positional arguments as a tuple. The parameter **kwargs also known as arbitrary keyword arguments allows us to accept multiple keyword arguments as a dictionary. Example 3.6 shows the use of arbitrary positional arguments.

Example 3.6: Program to show use of arbitrary positional arguments

```
# Program to show use of arbitrary positional arguments
def Add_Numbers(*args):
    return sum(args)

print(f"Addition = {Add_Numbers(1, 2, 3, 4, 5)}")
print(f"Addition = {Add_Numbers(10, 20, 30, 40, 50, 60)}")
```

OUTPUT:

```
Addition = 15
Addition = 210
```

In the above program we have used a function *Add_Numbers*, that accepts arbitrary positional arguments using **args*. The function returns the addition of the values passed to it as tuple using built-in library function *sum*. We have made two calls to the function. First call *Add_Numbers(1, 2, 3, 4, 5)* gives us output 15, while second call *Add_Numbers(10, 20, 30, 40, 50, 60)* gives us output 210. Note that the number of parameters and values passed in both the

function calls are different. Example 3.7 shows the use of arbitrary keyword arguments.

Example 3.7: Program to show use of arbitrary keyword arguments

```
# Program to show use of arbitrary keyword arguments
def print_student_details(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

name = input("Enter name of student: ")
age = int(input("Enter age of student: "))
city = input("Enter city of student: ")

print_student_details(Name=name, Age=age, City=city)
```

OUTPUT:

```
Enter name of student: Vidita
Enter age of student: 20
Enter city of student: Ahmedabad
Name: Vidita
Age: 20
City: Ahmedabad
```

In the above program we have defined a function *print_student_details* that takes variable-length keyword arguments (**kwargs). The **kwargs syntax allows the function to accept any number of keyword arguments as a dictionary. Each keyword argument is accessible as a key-value pair inside the function. The *for* loop iterates over each key-value pair in the kwargs dictionary (data passed by user). The kwargs.items() returns a view of the dictionary's items, which is then unpacked into key and value. The print statement formats and outputs each key and its corresponding value.

The program prompts the user to enter the details about a student (name, age, and city) and passes these details to the function. For the input seen in the example the kwargs dictionary will look like {"Name": "Vidita", "Age": 20, "City": "Ahmedabad"}.

Note:

Understanding how to use different types of function arguments will enable us to write more flexible and readable Python programs.

Check Your Progress-2

a) Python arguments can be classified in _____ types.
b) The _____ arguments are initialized with a predefined value.
c) The _____ arguments can be passed without worrying about its position during function call.
d) The _____ parameter is used when arbitrary positional arguments are to be passed.
e) The **kwargs parameter is used when arbitrary _____ arguments are to be passed.
f) Parameters are not _____ in function.

3.4 ANONYMOUS FUNCTION

Anonymous functions, or lambda functions, allows us to create small, unnamed functions in Python. They are particularly useful in functional programming paradigms where functions are passed as arguments to other functions. The syntax of lambda function is:

variable_name = lambda argument_list: expression

A lambda function is defined using the *lambda* keyword instead of *def* keyword. It is followed by a list of parameters separated by comma, a colon, and an expression (logic to be executed). The lambda functions are limited to a single expression and cannot contain multiple statements. Example 3.8 shows the use of lambda function.

Example 3.8: Program to show use of lambda function

```
# Program to show use of lambda function
square = lambda x: x * x

number = int(input("Enter a number: "))
result = square(number)
print(f"Square of {number} is {result}")
```

OUTPUT:

Enter a number: 6

Square of 6 is 36

In the above program we have created an anonymous function (lambda function) and have assigned it to the variable *square*. The function takes one parameter x. The expression x * x is the body of the function. It computes the square of x. When the function is called *result = square(number)*, the lambda function stored in *square* is called with the arguments as value of variable *number* (6 in this case). This triggers the evaluation of the expression x * x. The result of this multiplication, 36, is stored in the variable *result*. A message "Square of 6 is 36" is then printed.

Example 3.9 gives an example of another lambda function that uses if..else as its body.

Example 3.9: Program to show use of lambda function

```
# Program to show use of lambda function
greater = lambda x, y: x if x > y else y

num1 = int(input("Enter first number: "))
num2 = int(input("Enter second number: "))

result = greater(num1, num2)
print("Greater number is:", result)
```

OUTPUT – SCENARIO 1:
Enter first number: 8 Enter second number: 19 Greater number is: 19
OUTPUT – SCENARIO 2:
Enter first number: -10 Enter second number: -20 Greater number is: -10

In the above program we have created an anonymous function (lambda function) and have assigned it to the variable *greater*. The function takes two parameters x and y. The expression *x if x > y else y* is the body of the function. It checks which value is greater. When the function is called *result = greater(num1, num2)*, the lambda function stored in *greater* is called with the arguments as values of variable *num1* and *num2* (8 and 19 in this case). This triggers the evaluation of the expression *x if x > y else y*. The result of this evaluation is stored in the variable *result*. A message “Greater number is: 19 “ is then printed in OUTPUT – SCENARIO 1.

3.5 RECURSIVE FUNCTIONS

Till now, we have written only a single function in our program. In real life, it is very common to have multiple functions within a program. It is also possible for one function to call another function.

A recursive function can be defined as a function that calls itself. The recursive functions are made of the base case and the recursive case. The base case is used to stop the recursion, while the recursive part is where the function calls itself again and again. The program given in Example 3.10 shows an example of using a recursive function in a Python program.

Example 3.10: Example of using recursive function

```
# Program showing example of recursive function
def GCD(num1, num2):
    # Base case
    if num2 == 0:
        return num1
    else:
        # Recursive case with updated values
        new_num1 = num2
        new_num2 = num1 % num2
        num1 = new_num1
        num2 = new_num2
        return GCD(num1,num2)

num1 = int(input("Enter first number: "))
num2 = int(input("Enter second number: "))
result = GCD(num1,num2)
if (result < 0):
 result = result * -1
print(f"GCD of {num1} and {num2} is {result}")
```

OUTPUT- SCENARIO 1:

```
Enter first number: 48
Enter second number: 18
GCD of 48 and 18 is 6
```

OUTPUT- SCENARIO 2:

```
Enter first number: -12
Enter second number: 8
GCD of -12 and 8 is 4
```

In the above program we have used the concept of recursive functions. The user is prompted to enter the value of variables *num1* and *num2*. These values are then passed as parameters to function *GCD(num1, num2)* during the function call. When the function GCD() starts execution the base case checks

if *num2* is 0 (zero). If it is, the function returns the value of *num1* (GCD of any number and 0 is the number itself). If *num2* is not 0 (zero), the function calls itself recursively by changing the values of *num1* and *num2*. For every iteration the variable *new_num1* is assigned the value of *num2* and *new_num2* is assigned the remainder of *num1* divided by *num2* (*num1* % *num2*). We then modify the value of *num1* and *num2* by assigning the value of *new_num1* and *new_num2* to them. This process continues till we reach the base case (the value of *num2* becomes 0). When the value of *num2* becomes 0, we return the updated value of *num1*. The variable *result* is assigned this value. If the value of the result is negative we multiply -1 with it to get a positive value. Finally we print the value of *result*.

Let us look at how the answer in OUTPUT-SCENARIO 1 has been obtained. Here the user has entered values 48 and 18 in *num1* and *num2* respectively. Thus the first call to function is GCD(48,18). As the value of *num2* is not zero, a recursive call is made with new values of *num1* and *num2* GCD(18, 48 % 18) which is GCD(18, 12). The function calls are further repeated with values GCD(12, 18 % 12) which is GCD(12, 6) and GCD(6, 12 % 6) which is GCD(6, 0). As the value of *num2* is 0 the function returns value 6, which is the GCD of 48 and 18.

Note:

- In Python, the maximum number of times a function can run recursively is by default set to 1000.
- This means that a function can call itself recursively up to 1000 times.
- After the 1000th iteration we will get an error *"RecursionError: maximum recursion depth exceeded in comparison*".

3.6 SCOPE OF VARIABLE

Observe that in Example 3.10, the function GCD has been recursively called with variables *num1* and *num2*. How is it that two different functions or recursive functions can use the same variable names?. This feature is provided by a property called scope of variable. The scope of a variable refers to the context

within which that variable is accessible (used, modified) or visible. In general the variables are said to have two scopes: global and *local*. Thus we say that we have a global variable or a local variable.

Global Variable:

The global variables are defined outside the function (usually in the beginning of the program) and can be accessed anywhere (including other functions) in the program.

Local Variable:

The local variable is defined inside a function and is local to that function. Such variables cannot be accessed outside of the functions where they are defined. The local variables are created when a function is called and are destroyed when the function exits.

The program given in Example 3.11 shows the use of scope of variables in a Python program.

Example 3.11: Example of scope of variable

```
# Program showing example of scope of variable
num = 15.  # Global variable accessible in entire program
def print_num():
  num = 10 # Local variable scope limited to function print_num
  print(f"num is local variable of print_num and its value is {num}")

def sqr(num):
  print(f"num can be accessed from function sqr and its value is {num}")
  result = num * num
  return result

print(f"num can be accessed from the program and its value is {num}")
print_num()
result = sqr(num)
print(f"Square of {num} is {result}")
```

OUTPUT:

num can be accessed from the program and its value is 15
num is local variable of print_num and its value is 10
num can be accessed from function sqr and its value is 15
Square of 15 is 225

The program given in example 3.11 defines a global variable *num*. This variable is accessible across the entire program including the two functions print_num and sqr. Observe that the function print_num also defines a variable with the name *num*. Thus there exists two variables named *num* in the program at this point. To avoid any confusion between the two variables Python assigns scope to each of these variables.

The function print_num defines a variable *num* and assigns it a value 10. As the name of the variable defined in the function print_num is the same as that of the global variable, the value of global variable *num* will not be used in this function. Thus within the function print_num the variable *num* is known as local variable and its value will be 10.

The function sqr accepts the value of variable *num* and calculates its square and assigns it to the variable *result* (local variable of function sqr). It then returns the value of the variable *result* to the calling function.

As can be seen in OUTPUT; first we get the message “num can be accessed from the program and its value is 15”. Then as function print_num is called we get the message “num is local variable of print_num and its value is 10”. The control then moves to the statement that calls function sqr(num). The value of the global variable *num* is passed to the function. We first get a message “num can be accessed from function sqr and its value is 15”. The function then calculates the square and returns back value 225. Thus we get the final message “Square of 15 is 225”.

Check Your Progress-3

a) Anonymous functions are also known as _____ functions.
b) A function that calls itself is known as _____ function.
c) After the 1000th call the recursive functions will generate an error message. (True/False)
d) The variables defined in Python can have a _____ scope or a global scope.
e) A variable that is accessible in the entire program or different functions within the program is known as a local variable. (True/False)

3.7 LET US SUM UP

In this unit we have discussed the concept of functions. You have got a detailed understanding of what a function is, types of functions, different types of parameters used in function, anonymous function and recursive functions. You have also got an understanding of different scopes of variables. Further you are now aware of how to define and use a function in a program. The use of function makes the program more readable and concise.

3.8 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

1-a True
1-b False
1-c True
1-d False
1-e False
1-f False
2-a four
2-b default
2-c keyword
2-d *args
2-e keyword

2-f compulsory/mandatory

3-a lambda

3-b recursive

3-c True

3-d local

3-e False

3.9 ASSIGNMENTS

- Differentiate between user defined function and built in functions.
- State the advantages of using a function in a program.
- State the advantages of using library functions in a program.
- Explain the concept of recursive functions.
- Define Global variable and local variable.
- What do you mean by scope of a variable?
- Write a program to perform the given activity:
    - Write a user defined function which will display the sum of a series. given a number N the function should print the value of 1 + 2 + 3 + 4 +….. + N.
    - Write a user defined function which will calculate the value of x to the power n.
    - Write a user defined function to find minimum from the given two numbers.
    - Read a number in your program, pass it to the user defined function as an argument and display whether the given number is positive or negative.
    - Write a user defined function to print a Fibonacci series till given position. A Fibonacci series starts with 0. If the user enters 4, the program should print 0, 1, 1, 2. If the user enters 7, the program should print 0, 1, 1, 2, 3, 5, 8.

Unit-4: Modules in Python

4

Unit Structure

4.0 LEARNING OBJECTIVE

After studying this unit student should be able to:

- Understand what is a module
- Use a module in a program
- Understand different types of modules

4.1 INTRODUCTION

In the previous chapter we learnt how to convert small repetitive tasks into a function. The use of functions make our programs modular. This approach is good for small programs. The real life applications have thousands of lines of code. In such cases we need to store a function definition for a longer period of time. Also it is better to divide the code into as many smaller parts as possible to maintain the code properly.

Python language provides us a concept called module to assist us in segmenting the reusable functions and the core logic part of the program. In this chapter we will learn what a module is, which are the different types of modules. We will also learn how to use a module in a Python program.

4.2 MODULES AND ITS TYPE

A module in Python is a file containing Python definitions, executable statements and classes. To create a module in Python we need to create a file name with extension .py similar to a normal Python program. For example if we create a file named mymodule.py then “mymodule” will be used as a module name. A module once created can be reused as many times as we want in different programs. There are two categories of modules in Python; Standard and Custom.

Standard Modules

Python comes with a library of standard modules. Some of these modules are built into the interpreter. These modules provide access to operations that are not part of the core of the language but are nevertheless built in, either for

efficiency or to provide access to operating system primitives such as system calls. Such modules at times also depend on the underlying platform.

Custom Modules

The modules created by the programmer (user) are known as custom modules. These modules help the programmer to segment large pieces of code into multiple files. It also assists in project development by multiple programmers.

To use a module in a Python program or interpreter we need to use the keyword *import*. The general syntax of using a module in a Python program is as follows:

import module_name

Modules can import other modules. The import statement is usually written at the beginning of a module or a Python program. But it is not compulsorily required to do so. A module can contain executable statements, function definitions as well as classes. These statements are intended to initialize the module. They are executed only the first time the module name is encountered in an import statement.

Each module has its own private namespace, which is used as the global namespace by all functions defined in the module. Thus, one can use global variables in the module without worrying about accidental clashes with a user's global variables. The import statement brings all the contents from a given module into a requested environment.

If the user does not want to import all the definitions from the module but only needs a selected few definitions, then an alternate form of import shown herewith can be used.

from module_name import def_name1, def_name2,…., def_nameN

here def_name1, def_name2,…., def_nameN are the definitions that are imported in the requested environment.

The from..import statement can also be used to import all definitions from a module. To do so the syntax is as mentioned:

from module_name import *

Rather than duplicating definitions into several programs, we may define the frequently used functions in a separate module and then import the complete module or required definition. This makes the program more readable and problems if any can be found easily.

Check Your Progress-1

a) A module is a single program with multiple definitions. (True/False)
b) To maintain definitions for a long duration, we have _____ types of modules.
c) The keyword include is used to add a module in a Python program. (True/False)
d) A module file has extension .mod in Python. (True/False)
e) It is possible to include some required definitions in a Python program. (True/False)

4.3 STANDARD MODULES

Python has a huge list of standard modules that can be used by a programmer. Though discussing all of them is out of scope of this book. We will try to look into one or two commonly used modules.

Assume that the user wants to write a program to calculate the circumference of a circle. We know that the circumference of circle is calculated as 2 * π * r, The value of π in our case will be taken from the *math* module. Example 4.1 shows the use of the math module.

Example 4.1: Program to show use of standard module

```
#Program to show use of standard module
import math

radius = float(input("Enter the radius of the circle: "))
if (radius < 0):
    print("Radius cannot be negative.")
else:
```

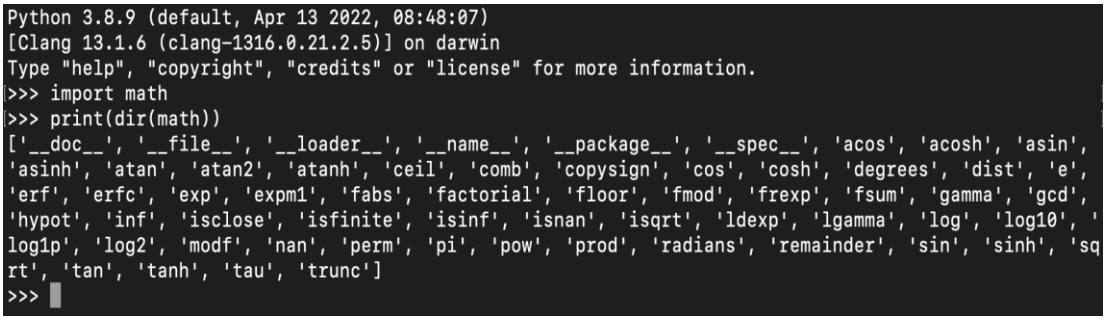
*circum = 2 * math.pi * radius* *print(f"The circumference of the circle with radius {radius} is {circum:.2f}")*
OUTPUT-SCENARIO 1: Enter the radius of the circle: 5.5 The circumference of the circle with radius 5.5 is 34.56
OUTPUT-SCENARIO 2: Enter the radius of the circle: -10 Radius cannot be negative.

In the above program we have imported the *math* module to access the value of constant π. The program then prompts the user for the value of radius. If the value greater than zero (0) the circumference of circle is calculated using equation *circum = 2 * math.pi * radius*. Observe the use of *math.pi* here. The term here indicates that we are trying to access a value of *pi* from a module named *math*. We thus get the output message “The circumference of the circle with radius 5.5 is 34.56” as seen in OUTPUT-SCENARIO 1. The use of 2f in *{circum:.2f}* limits the output of the circumference to two digit decimal only. When a value less than zero (0) is entered we get the output message “Radius cannot be negative.” as seen in OUTPUT-SCENARIO 2.

In the above example we directly used the term math.pi, what if the user does not know which value or definition to use from the module? In such cases the user can take help of the following built in function:

print(dir(module_name))

The *dir* function when passed with the name of the module will list all the function and constant names in a module. Figure 4.1 shows how the *dir* function can be used to get the required information.



Here, we have first gone to the Python interpreter in the terminal by typing *python3*. Then we have imported the *math* module (>>> import math). We have then used the statement print(dir(math)) to look into the values and definitions of the *math* module. As can be observed we can see a sorted list of names. The names that begin with an underscore are default Python attributes associated with the module. For example, the __name__ attribute contains the name of the module, other terms enclosed in single quotes are either constants or functions.

A user can check whether the term is a constant or function by using print function as shown:

>>> print(math.radians)

<built-in function radians>

>>> print(math.pi)

3.141592653589793

In the interpreter when we type **print(math.radians)** we get a message **<built-in function radians>**, thus indicating that *radians* is a function. On the other hand when we type **print(math.pi)** we get a value **3.141592653589793**, indicating that pi is a constant. Observe that the output in Figure 4.1 has many terms including factorial and gcd in it. Let us now try to use the functions factorial and gcd in our program. Example 4.2 shows the Python program that uses both these functions.

Example 4.2: Program to show use of selective functions from a module

```
#Program to show use of selective functions from a module
from math import factorial,gcd
number = int(input("Enter a number: "))
if (number < 0):
    print("Number is negative.")
else:
    fact = factorial(number)
    print(f"The factorial of {number} is {fact}")
    result = gcd(6,number)
    print(f"The GCD of 6 and {number} is {result}")
```

OUTPUT-SCENARIO 1:

Enter a number: 4

The factorial of 4 is 24

The GCD of 6 and 4 is 2

OUTPUT-SCENARIO 2:

Enter a number: -15

Number is negative.

In the above program we have imported two functions *factorial* and *gcd* from the *math* module. The program then prompts the user for the value of the number. If the value is greater than zero (0) the factorial of the number is calculated. We also calculate the greatest common divisor of 6 and the number. Observe the use of *factorial(number)* and *gcd(6,number).* Here we have not preceded the usage of both the functions with “math.”, as we have only included the functions.

When the user enters value 4 in number, we get the output message “The factorial of 4 is 24” and “The GCD of 6 and 4 is 2” as seen in OUTPUT-SCENARIO 1. When a value less than zero (0) is entered we get the output message “Number is negative.” as seen in OUTPUT-SCENARIO 2.

Note:
The user can get the list and details of the standard modules at https://docs.python.org/3/py-modindex.html

4.4 CUSTOM MODULES

We saw examples of how to use standard modules in our program. During creation of a big project at times it so happens that users need to create their own modules. The modules created by users are known as custom modules.

Let us now try and create a custom module. Assume that the user wants to look at different operations of a circle like calculate the area, circumference and diameter. We will first create a custom module called *circle.py* and create all the required functions in it. The code of the module will look as shown in Example 4.3

Example 4.3: Contents of circle.py

```
# Creation of custom module circle
import math as m
def Area(radius):
    return m.pi * radius * radius

def Circumference(radius):
    return 2 * m.pi * radius

def Diameter(radius):
    return 2 * radius
```

In the above code we have imported the standard module *math* with a short name *m*. Using the short name can save us typing time in some cases. Observe the use of *m.pi* instead of *math.pi* here. We have also defined three functions that calculate area, circumference and diameter of a circle if radius is given as input to them. This file will be treated as a custom module when used in any Python program. Example 4.4 shows how to use a custom module.

Example 4.4: Program to show use of custom module

```
#Program to show use of custom module
import circle as C
radius = float(input("Enter a radius: "))
```

```
if (radius < 0):
    print("Radius cannot be negative.")
else:
    area = C.Area(radius)
    print(f"The area of circle with radius {radius} is {area:.2f}")
    circum = C.Circumference(radius)
    print(f"The circumference of circle with radius {radius} is {circum:.2f}")
    dia = C.Diameter(radius)
    print(f"The diameter of circle with radius {radius} is {dia:.2f}")
```

OUTPUT – SCENARIO 1:

```
Enter a radius: 6.5
The area of circle with radius 6.5 is 132.73
The circumference of circle with radius 6.5 is 40.84
The diameter of circle with radius 6.5 is 13.00
```

OUTPUT – SCENARIO 2:

```
Enter a radius: -5
Radius cannot be negative.
```

In the above program we have imported the custom module *circle* as *C*. The program then prompts the user for the value of radius. If the value greater than zero (0) then area, circumference and diameter of the circle is calculated using functions defined in circle.py. Observe the use of *C.Area(radius)*, *C.Circumference(radius)* and *C.Diameter(radius)*. When user enters 6.5 as value of radius we get the output message “The area of circle with radius 6.5 is 132.73”, “The circumference of circle with radius 6.5 is 40.84” and “The diameter of circle with radius 6.5 is 13.00” as seen in OUTPUT-SCENARIO 1. When a value less than zero (0) is entered we get the output message “Radius cannot be negative.” as seen in OUTPUT-SCENARIO 2.

It is also possible to get the help of the circle module also. Figure 4.2 shows the output of the *dir* function that gives us the required information from the circle module.

```
Python 3.8.9 (default, Apr 13 2022, 08:48:07)
[Clang 13.1.6 (clang-1316.0.21.2.5)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import circle
>>> print(dir(circle))
['Area', 'Circumference', 'Diameter', '__builtins__', '__cached__', '__doc__', '
__file__', '__loader__', '__name__', '__package__', '__spec__', 'm']
>>>
```

Here, we have gone to the Python interpreter in the terminal by typing *python3*. Then we have imported the *circle* module. We have then used the statement print(dir(circle)) to look into the values and definitions of the *circle* module. As can be observed we can see a sorted list of names.

Let us have a look at another example that uses both the standard as well as custom module. We will now use a standard module named *sys*, this module is used to access system specific parameters and functions. We will make use of the module circle that we have previously created. Example 4.5 shows a Python program that passes command line arguments to the functions and performs the required operations.

Example 4.5: Program to show use of command line arguments

```
#Program to show use of command line arguments
import sys
from circle import Area as A
print("First command line argument: ",sys.argv[0])
print("Second command line argument: ",sys.argv[1])
radius = float(sys.argv[1])
if (radius < 0):
    print("Radius cannot be negative.")
else:
    area = A(radius)
    print(f"The area of circle with radius {radius} is {area:.2f}")
```

OUTPUT – SCENARIO 1:

```
First command line argument:  4_5.py
Second command line argument:  6.5
The area of circle with radius 6.5 is 132.73
```

OUTPUT – SCENARIO 2:

First command line argument: 4_5.py

Second command line argument: -15

Radius cannot be negative.

In the above program we have imported a standard module *sys* and a function *Area* as *A* from the custom module circle. Once the sys module is imported we get access to the *argv[ ]* array that is used to store or represent the command line arguments. The program then prints the value of the first and second command line argument that is passed by the user. The next statement converts the value of *argv[1]* to float and assigns it to variable *radius*. If the value is greater than zero (0) then the area of the circle is calculated. Observe the use of *area = A(radius)*, Here we have not preceded the usage of the function with "circle.", as we have only included the function. We thus get the output message "First command line argument: 4_5.py", "Second command line argument: 6.5" and "The area of circle with radius 6.5 is 132.73" as seen in OUTPUT-SCENARIO 1.

Here the first output "First command line argument: 4_5.py" represents the name of the Python program. The second output "Second command line argument: 6.5" represents the value of radius the user wants to pass. The third output shows the area of the circle. When a value less than zero (0) is entered we get the output message "First command line argument: 4_5.py", "Second command line argument: -15" and "Radius cannot be negative." as seen in OUTPUT-SCENARIO 2.

Note:

To execute the Python program shown in Example 4.5 we need to type the following commands in the terminal to get output 1 and 2 respectively:

- **python3 4_5.py 6.5**
- **python3 4_5.py -15**

Check Your Progress-2

a) The functions print(dir(module_name)) can be used to find details about a specific module. (True/False)
b) The modules available in the Python library are known as _____ modules.
c) The __name__ attribute contains the name of the module. (True/False)
d) To use command line arguments in a Python program we need to import the _____ module.
e) It is possible to give a short name to the module when using it in a Python program. (True/False)
f) Both the standard and custom module can be used together in a Python program. (True/False)

4.5 LET US SUM UP

In this unit we have discussed the concept of modules. You have got a detailed understanding of what a module is, types of modules. You have also got an understanding of how to look for help when looking into standard modules. Further you are now aware of how to define and use a module in a program. The use of modules allows the user to segregate core logic of the program and the functions that are reused again and again. The use of module also makes the program more readable and concise.

4.6 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

1-a True
1-b two
1-c False
1-d False
1-e True
2-a True
2-b standard
2-c True

2-d sys

2-e True

2-f True

4.7 ASSIGNMENTS

- What is a module? How are modules different from functions?
- Explain how modules can be imported in a Python program or another module.
- How can we get the details of the contents written in the module file?
- State the advantages of using modules in a program.
- Create modules that perform the given activity and use them in Python programs:
    - A module that defines function **Series** and **Fibonacci**.
        - The **Series** function will display the sum of a series. given a number N the function should print the value of 1 + 2 + 3 + 4 +….. + N.
        - The **Fibonacci** function will print a Fibonacci series till given position N. If the user enters N=4, the program should print 0, 1, 1, 2.
    - A module that defines functions **Square** and **Cube**.
        - The **Square** function returns the square of the given number.
        - The **Cube** function returns the cube of the given number.
    - A module that defines function **Max** and **Min**
        - The **Max** function returns the maximum of the given two numbers.
        - The **Min** function returns the minimum of the given two numbers.

Block-3
Data Structures of Python

Unit-1: Lists and Tuples

1

Unit Structure

1.0 LEARNING OBJECTIVE

After studying this unit student should be able to:

- Understand the concepts of lists and tuples in Python.
- Learn how to create, access, and update lists.
- Apply common list operations and functions.
- Understand tuples as immutable sequences and learn how to use them effectively.
- Use slicing and indexing techniques for both lists and tuples.

1.1 INTRODUCTION

A data structure in a programming language is used to organize, store, process and manage data. The sequence is Python's most basic data structure. Each component of a sequence is assigned a number that denotes its index or location. The first index is 0, the second is 1, and so on. You can do specific things with each type of sequence. Some of these operations include indexing, slicing, adding, multiplying, and performing membership checks.

Lists and tuples are essential sequential or linear data structures in Python for storing collections of elements. Despite their initial similarities, each has unique characteristics and applications. Tuples are immutable and cannot be altered once defined, whereas lists are mutable, which means they can be changed after creation.

In this chapter we will create, access, and manipulate lists and tuples using Python's built-in functions and methods. We will also look at other operations that can be performed on these data structures.

1.2 CREATING, ACCESSING AND UPDATING THE LIST

As mentioned, a List is a sequential data structure in Python. It is used for storing collection of elements. In Python, a list is created by placing items inside square brackets [ ], separated by commas. A list can hold items of different data types, such as integer, string, float or even other list. The general syntax used to create a list is as mentioned:

list_name = [value1, value2, ..., valueN]

The example of how to create a list in Python is given in the program of Example 1.1.

Example 1.1: Creation of List

```
# Program to create a list of integers and strings
num_list = [10, 20, 30, 40, 50]
print(num_list)
# list of string values
fruits_list = ["Apple", "Banana", "Cherry", "Grapes"]
print(fruits_list)
```

OUTPUT:

```
[10, 20, 30, 40, 50]
['Apple', 'Banana', 'Cherry', 'Grapes']
```

In the above program the two lists *num_list* and *fruits_list* have been created and initialized with a set of integer and string values respectively. The print function is used to display the contents stored in both the lists as can be seen in the output.

Accessing Elements in a List:

Once the list has been created it is possible to access the elements of the list and perform operations on it. Each element of a list is stored at a specific index. In Python lists are zero-indexed, it means that the first element of the list has index value = 0. You can access elements of the list using their index positions. The general syntax to access a list element is as mentioned:

list_variable[index]

Once an element is accessed, we can use it to perform operations or print the value. The Python program in Example 1.2 shows how to access and print the value of an element at a specific index.

Example 1.2: Accessing elements of a List

```
# Program to access elements from a list
fruits_list = ["Apple", "Banana", "Cherry", "Grapes"]
print(fruits_list[2])
print(fruits_list[-1])
```

OUTPUT:

Cherry

Grapes

In the above program a list *fruits_list* has been initialized with values "Apple", "Banana", "Cherry" and "Grapes" respectively. The print function is used to print the value of the element at a specific index. Here as Apple is stored at index value 0, Banana is stored at index value 1. Thus, when we use the statement *print(fruits_list[2])*, we get output as Cherry. Observe that we can also use a negative index to access elements from the end of the list. Thus, the statement *print(fruits_list[-1])* results into Grapes as output.

Updating values in the lists

The lists in Python are mutable, hence we can modify individual elements of a list or even entire slices of a list. Once a list is created, one or more elements of a list can be easily updated by giving the element index or the slice on the left-hand side of the assignment operator. The general syntax to update is as mentioned:

For updating an element: *list[index] = new value*

For updating a slice: ***list[start:end] = new values***

Example 1.3 shows how to update an element at a specific index as well as how to update a slice.

Example 1.3: Updating a List

```
# Program to update elements from a list

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

print("Original List: ",num_list)
```

```
#Updating an element
num_list[5] = 100
print("Updated list: ",num_list)
#Updating a slice
num_list[7:9] = [45, 55, 65]
print("List after slice update: ", num_list)
```

OUTPUT:

Original List: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Updated list: [1, 2, 3, 4, 5, 100, 7, 8, 9, 10]

List after slice update: [1, 2, 3, 4, 5, 100, 7, 45, 55, 65, 10]

The above program demonstrates how to update a specific element in a Python list by using its index. As lists are mutable the statement *num_list[5] = 100* updates the value of the 6th element stored at index 5 from 6 to 100. Thus, the updated list now becomes [1, 2, 3, 4, 5, 100, 7, 8, 9, 10]. The program also shows an example of updating a slice. The statement *num_list[7:9] = [45, 55, 65]* updates the values stored at index 7, 8 and 9, hence the final list becomes : [1, 2, 3, 4, 5, 100, 7, 45, 55, 65, 10].

Check Your Progress-1

a) A list is a grouping of indexed, changeable, and ordered items. (True/False)
b) To create a list in python { } bracket is used. (True/False)
c) In python list, the first element of the list has index value 1.(True/False)
d) Python lists are capable of storing elements of several data types, such as texts, floats, and integers. (True/False)
e) List_name(value) is the syntax used to access a list element. (True/False)

1.3 LIST OPERATIONS AND FUNCTIONS

The list, in Python is very versatile and powerful. The list supports many operations like indexing, slicing, updating, adding/removing elements and others. Knowing how to manipulate lists is crucial for efficient Python programming, regardless of the size of the dataset one is dealing with. Let us now have a look at some basic list operations that can be performed using +, * and 'in' operators. Table 1.1 gives the details of the operation that can be performed.

Table 1.1: Basic operations on List

Operator	Operation	Description
+	Concatenation	Creates new list by merging two lists
*	Repetition	Makes a list's items appear a predetermined amount of times.
in	Membership	Determines if an element is present in the list. It returns a Boolean value 'true' if the element is present otherwise returns 'false'.

Let us now write a program that performs all the operations mentioned in Table 1.1. Example 1.4 shows how to perform basic operations on a list.

Example 1.4: Basic operations on List

```
# Program to perform basic operation on a list

list1 = [1, 3, 5]

list2 = [2, 4, 6]

#Concatenate a list

combined_list = list1 + list2

print("List 1: ",list1)

print("List 2: ",list2)

print("Concatenated List: ",combined_list)
```

```
#Repetition of list

repeated_list = list1 * 2

print("Repeated List: ",repeated_list)

#Membership in list

print(3 in repeated_list)

print("3" in repeated_list)
```

OUTPUT:

List 1: [1, 3, 5]

List 2: [2, 4, 6]

Concatenated List: [1, 3, 5, 2, 4, 6]

Repeated List: [1, 3, 5, 1, 3, 5]

True

False

The above program defines two lists *list1* and *list2* with values 1, 3, 5 and 2, 4, 6 respectively. The statement *combined_list = list1 + list2* concatenates the lists and creates a new list with values 1, 3, 5, 2, 4, 6. The statement *repeated_list = list1 * 2* creates a new list named repeated_list and assigns it values 1, 3, 5, 1, 3, 5. (Here the contents of list1 are repeated twice). Observe the output of statements *print(3 in repeated_list)* and *print("3" in repeated_list)*. The first statement results in the output as True (as integer 3 is part of the list) while the second statement results in the output False (as character 3 is not part of the list).

List Functions

Python provides many inbuilt methods that can be used with lists to manipulate the data quickly. These methods work only on lists. They do not work on the other sequence data types that are not mutable. Table 1.2 gives the list of methods along with its description.

Table 1.2: List Functions

Method	Description	Syntax
append()	To add item at the end of the list	list.append(element)
insert()	To a specific item at a given index.	list.insert(index,element)
extend()	To add all of the elements from another list to the existing list.	list1.extend(list2)
pop()	To remove and return an element from a given index or the last element (in absence of index)	list.pop() list.pop(index)
sort()	To sort the elements of the list in ascending order. You can sort a list in descending order using the sort() method with the reverse=True parameter, or the sorted() function.	list.sort() list.sort(reverse=True)
reverse()	To place the list in reverse order	list.reverse()
len()	Returns the number of items in the list.	len(list)
min()	Returns the smallest item in the list.	min(list)
max()	Returns the largest item in the list.	max(list)
sum()	Returns the total of all the list's numerical elements.	sum(list)

Let us now write a program that uses all the methods in Table 1.2. Example 1.5 shows how to use these methods with lists.

Example 1.5: Using methods with List

```
# Program to use methods and list

list1 = [1, 2, 3, 4, 5]

list2 = [7, 8, 9]

print("Original List1: ",list1)

list1.append(6)

print("Appended List1: ",list1)
```

```
list1.extend(list2)
print("Extended List1: ",list1)
list1.insert(10,0)
print("List after insert at index 10: ",list1)
list1.pop()
print("List after pop operation: ",list1)
list1.pop(0)
print("List after pop operation at index 0: ",list1)
```

OUTPUT:

```
Original List1:  [1, 2, 3, 4, 5]
Appended List1:  [1, 2, 3, 4, 5, 6]
Extended List1:  [1, 2, 3, 4, 5, 6, 7, 8, 9]
List after insert at index 10:  [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
List after pop operation:  [1, 2, 3, 4, 5, 6, 7, 8, 9]
List after pop operation at index 0:  [2, 3, 4, 5, 6, 7, 8, 9]
```

The above program defines two lists *list1* and *list2* with values 1, 2, 3, 4, 5 and 7, 8, 9 respectively. The statement *list1.append(6)* adds a new element at the end of the list1. Thus, the new list has values 1, 2, 3, 4, 5, 6. The statement *list1.extend(list2)* extends list1 and adds elements of list2 in it. The list1 now becomes 1, 2, 3, 4, 5, 6, 7, 8, 9. The statement *list1.insert(10,0)* inserts a new element with value 0 at index 10 of the list. The list1 now becomes 1, 2, 3, 4, 5, 6, 7, 8, 9, 0. We then use the statement *list1.pop()*, this modifies the list1 back to 1, 2, 3, 4, 5, 6, 7, 8, 9. The last statement *list1.pop(0)* removes the element at index 0 and the final value of list1 is 2, 3, 4, 5, 6, 7, 8, 9,

Let us now write a program that uses the sorting and reversing methods. Example 1.6 shows how to use these methods with list.

Example 1.6: Sort and reverse methods with List

```
# Program to use methods for sorting list
list1 = [1, 2, 3, 4, 5]
list2 = ["Grapes", "Banana", "Apple", "Kiwi", "Avocado"]
print("Original List1: ",list1)
print("Original List2: ",list2)
list1.sort(reverse=True)
print("List in descending order: ",list1)
list2.sort()
print("List2 in ascending order: ",list2)
```

OUTPUT:

```
Original List1:  [1, 2, 3, 4, 5]
Original List2:  ['Grapes', 'Banana', 'Apple', 'Kiwi', 'Avocado']
List in descending order:  [5, 4, 3, 2, 1]
List2 in ascending order:  ['Apple', 'Avocado', 'Banana', 'Grapes', 'Kiwi']
```

The above program defines two lists; *list1* that contains integer values and list2 that contains strings. The statement *list1.sort(reverse=True)* rearranges the values of list1 in reverse order. Thus, the new values of list1 are 5, 4, 3, 2, 1. The next statement *list2.sort()* sorts the list2 in ascending order based on the alphabets used in the string. Thus, the output of the sorted list is 'Apple', 'Avocado', 'Banana', 'Grapes', 'Kiwi'

Let us now write a program that uses the aggregate methods. Example 1.7 shows how to use these methods with list.

Example 1.7: Aggregate methods with List

```
# Program to use aggregate methods and list
list = [10, 2, 30, 4, 50]
print("Original List: ",list)
print("Elements in list: ",len(list))
print("Minimum element in the list: ",min(list))
print("Maximum element in the list: ",max(list))
print("Sum of elements in the list: ",sum(list))
```

OUTPUT:

Original List: [10, 2, 30, 4, 50]

Elements in list: 5

Minimum element in the list: 2

Maximum element in the list: 50

Sum of elements in the list: 96

The above program defines a list that contains integer values. We have performed operations to find the number of elements, minimum and maximum elements and sum of all the elements in the list.

Check Your Progress-2

a) The insert() method is used to add an element to the end of a list in Python. (True/False)

b) Consider following python code:

```
list1 = [1, 2, 3, 4]
list1.pop(2)
print(list1)
```

The output of the given code is: [1, 2, 4] (True/False)

c) By default, the pop() method removes a list's final element unless an index is supplied. (True/False)

d) The len() method can only be used on lists which contain numeric values. (True/False)

e) The + operator can be used to concatenate two lists, creating a new list by merging their elements. (True/False)

1.4 TUPLES: IMMUTABLE SEQUENCES

A tuple is an immutable linear data structure. In contrast to lists, once a tuple is defined, the elements in it cannot be altered. In all other ways the tuple and list are essentially the same. Tuples are typically used when you want a collection of items that should not change during the execution of the program.

A tuple is created by placing the value of elements inside a parenthesis ( ) or by simply separating each element by using a comma.

The general syntax of using tuple is as mentioned:

Tuple = (value 1,value 2,….,value n)

Tuple = "value 1", "value 2", value 3, …. , "value n"

here value 1, value 2, …, value n, can be an integer value, a floating number, a character, or a string.

The example of how to create a tuple in a Python program is given in the program of Example 1.8.

Example 1.8: Creation of Tuple

```
# Program to create a tuple of integers and strings
#Tuple of integers
tuple1=(1,2,3,4,5)
print("Tuple 1: ",tuple1)
#Tuple of strings
tuple2=("abc","def","pqr")
print("Tuple 2: ",tuple2)
#Mix value tuple
tuple3 = "abc", 1, "def", 5
print("Tuple 3: ",tuple3)
```

OUTPUT:

```
Tuple 1:  (1, 2, 3, 4, 5)
Tuple 2:  ('abc', 'def', 'pqr')
Tuple 3:  ('abc', 1, 'def', 5)
```

In the above program the three tuples; tuple1, tuple2 and tuple3 have been created and initialized. The values of tuple1 are a set of integers, tuple2 contains string values. The tuple3 is a tuple that contains mixed values of both strings and integers. The print function is used to display the contents stored in all the tuples as can be seen in the output.

Accessing Elements in a Tuple:

Similar to list once a tuple has been created it is possible to access its elements and perform operations on it. Each element in the tuple is also stored at a specific index. The general syntax to access a tuple element is as mentioned:

tuple_variable[index]

Once an element is accessed, we can use it to perform operations or print the value. Example 1.9 shows how to access and print the value of an element at a specific index.

Example 1.9: Accessing elements of a tuple

```
# Program to access elements from a tuple
tuple = (1, 2, 3, 4, 5)
print("Element at index 2 in tuple: ",tuple[2])
tuple1 = "abc", 1, "def", 5
print("Element at index 0 in tuple1: ",tuple1[0])
print("Element at last index in tuple1: ",tuple1[-1])
```

OUTPUT:

```
Element at index 2 in tuple:  3
Element at index 0 in tuple1:  abc
Element at last index in tuple1:  5
```

In the above program the variable tuple has been initialized with integer values. The tuple1 is initialized with mixed data. The print function is used to print the value of the element at a specific index. Thus when we use the statement *print("Element at index 2 in tuple: ",tuple[2])*, we get output as 3. Similarly, when we use the statement *print("Element at index 0 in tuple1: ",tuple1[0])* we get output as 'abc'. Observe that we can also use a negative index to access elements from the end of the tuple. Thus, the statement *print("Element at last index in tuple1: ",tuple1[-1])* results into 5 as output.

Immutability in Tuple

The tuple, in Python are immutable, hence we cannot modify elements of a tuple. Any attempt to change the elements of tuple will result in an error. Example 1.10 shows what happens if we try to update an element in a tuple.

Example 1.10: Updating a Tuple

```
# Program to update elements from a tuple
tuple = (1, 2, 3, 4, 5)
tuple[0] = 30
print (tuple)
```

OUTPUT:

```
Traceback (most recent call last):
  File "31_10.py", line 3, in <module>
    tuple[0] = 30
TypeError: 'tuple' object does not support item assignment
```

The above program demonstrates what happens if we try to update an element of a tuple. Observe that we have not been able to execute the program as we get an error. The statement “TypeError: 'tuple' object does not support item assignment” makes it very clear that a tuple element cannot be assigned.

Check Your Progress-3

a) A tuple is a mutable linear data structure. (True/False)

b) Consider the following code:

```
tuple1 = (1, 2, 3, 4, 5)
print(tuple1[-1])
```

The output of the given code is: 5 (True/False)

c) Tuples cannot hold a mix of data types such as integers, strings, and floats. (True/False)

d) The immutability of tuples means that any attempt to change one of their elements will result in a TypeError. (True/False)

1.5 SLICING AND INDEXING IN LISTS AND TUPLES

We have already looked at the concept of index and slicing in the previous sections. In Python, slicing and indexing are techniques used to access elements in sequences like lists and tuples. Let us have a relook at these techniques.

Indexing

Indexing in Python is used to retrieve a single element from a list or a tuple based on its position. The index values in Python start at 0 for the first element, 1 for the second, and so on. It is also possible to use negative indexes in Python. Here -1 refers to the last element, -2 refers to the second-last element, and so on. Let us have a look at a simple example as shown in Example 1.11.

Example 1.11: Use of indexing in a List and Tuple

```
# Program to show use of indexing in list and tuple
# List Example
list = [10, 20, 30, 40, 50]
print("Element at index 0: ",list[0])
print("Last element of list: ",list[-1])
# Tuple Example
tuple = ('a', 'b', 'c', 'd', 'e', 'f')
print("Element at index 1: ",tuple[1])
print("Element at second last index: ",tuple[-2])
```

OUTPUT:

```
Element at index 0:  10
Last element of list:  50
Element at index 1:  b
Element at second last index:  e
```

In the above program we have initialized a variable *list* with integer values and the variable *tuple* is initialized with characters. The print function is used to print the value of the element at a specific index. Thus, when we use the statements; *print("Element at index 0: ",list[0])* and *print("Last element of list: ",list[-1]),* we get output 10 and 50. Similarly, when we use the statements; *print("Element at index 1: ",tuple[1])* and *print("Element at second last index: ",tuple[-2])*, we get output as ‘b’ and ‘e’.

Slicing

At times when we work with lists and tuples, we need to extract a subset of the data that is given to us. The concept of slicing allows us to extract a subset of elements from a list or tuple. The general syntax to use the slicing is as mentioned:

sequence[start : end : step]

Where:

start is the index to begin slicing (default is 0)

end is the index to stop slicing (exclusive).

Step is the interval (default is 1)

Table 1.3 shows the different ways in which the start, end and step values are used in list or tuple. The value of var in the below table refers to a list or tuple variable.

Table 1.3: Slicing Options

Usage	Description
var[start:end]	Extracts a subset of elements starting from index **start** to **end-1** i.e inclusive of start, exclusive of end.
Var[start:end:step]	Extracts a subset of elements starting from index **start** to **end-1** with a step size of **step**.
Var[:end]	Extracts a subset of elements from the **beginning** (index 0) up to **end-1**.
Var[start:]	Extracts a subset of elements from **start** to the **end of the list**.
Var[::-1]	Reverses the list by using a negative step.
Var[:end:step]	Extracts a subset of elements from the **beginning** to **end-1** with a step size of **step**.
Var[start::step]	Extracts a subset of elements from **start** to the **end** of the list with a step size of **step**.
Var[-n:]	Extracts the **last n** elements of the list.
Var[:-n]	Extracts the list **excluding** the last n elements.

Let us have a look at an example of slicing given in Example 1.12.

Example 1.12: Use of slicing in a List and Tuple

```
# Program to show use of slicing in list and tuple

#List Example

list1 = [10, 20, 30, 40, 50]

print("Elements from index 1 to 3: ",list1[1:4])

print("Elements till index 2: ",list1[:3])

print("Alternate Elements : ",list1[::2])

print("All Elements from end to index 0: ",list1[::-1])


# Tuple example

tuple1 = ('a', 'b', 'c', 'd', 'e')

print("Elements from index 1 to 3: ",tuple1[1:4])

print("Elements till index 1: ",tuple1[:2])

print("All Elements from end to index 0: ",tuple1[::-1])
```

OUTPUT:

```
Elements from index 1 to 3:  [20, 30, 40]

Elements till index 2:  [10, 20, 30]

Alternate Elements :  [10, 30, 50]

All Elements from end to index 0:  [50, 40, 30, 20, 10]

Elements from index 1 to 3:  ('b', 'c', 'd')

Elements till index 1:  ('a', 'b')

All Elements from end to index 0:  ('e', 'd', 'c', 'b', 'a')
```

Observe that the output of the program is as per the description mentioned in Table 1.3.

Check Your Progress-4

a) Consider the following code:

```
tuple1 = ('a', 'b', 'c', 'd', 'e')
print(tuple1[::-1])
```

The output of the given code is: ('e', 'd', 'c', 'b', 'a') (True/False)

b) The concept of indexing allows us to extract a subset of elements from a list or tuple. (True/False)

c) In Python, Index begins with the first element at 1 and finishes with the last element at -1. (True/False)

d) The slicing syntax var[start:] is used to extracts a subset of elements from start to the end of the list. (True/False)

e) In Python, negative indices can be used for both indexing and slicing. (True/False)

1.6 LET US SUM UP

In this unit, we learned about two of Python's essential data structures: lists and tuples. Lists are mutable, allowing modifications after creation, while tuples are immutable, providing a way to store unchangeable data. We explored various operations, functions, and methods that allow you to manipulate lists, and we saw how tuples offer a stable, unchanging sequence. Slicing and indexing techniques were discussed as ways to access specific parts of these data structures.

1.7 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

1-a True
1-b False
1-c False
1-d True
1-e False
2-a False
2-b True

2-c True
2-d False
2-e True
3-a False
3-b True
3-c False
3-d True
4-a True
4-b False
4-c False
4-d True
4-e True

1.8 ASSIGNMENTS

1. What is the key difference between a list and a tuple?
2. How do you access the second element in the list items = ["item1", "item2", "item3", "item4"]?
3. What does the method append() do in a list?
4. Give an example for a nested list.
5. Write a python program to print the list elements using a for loop.
6. Write difference between indexing and slicing.
7. How to access each tuple element in reverse order using the help of slicing?
8. Create a list of five student names and perform the following:
   - Add two more names to the list.
   - Remove the first name from the list.
   - Replace the third name with another name of your choice.
   - Print the final list.
9. Create a list of ten numbers and perform the following operations:
   - Slice the list to get the first three elements.
   - Slice the list to get the last three elements.
   - Slice the list with a step of 2.
10. Write a program to find the sum of all even numbers in a list.

11. Write a program that reverses a list using a loop.
12. Write a Python program that create given tuple and perform following:

    tuple1 = (10,50,20,40,30)

    - To display the elements 10 and 50 from tuple1
    - To display the length of a tuple1.
    - To find the minimum element from tuple1.
    - To add all elements in the tuple1.
    - To display the same tuple1 multiple times.

Unit-2: Dictionaries 2

Unit Structure

2.0 LEARNING OBJECTIVE

After studying this unit student should be able to:

- Understand what dictionaries are and how they differ from other data structures.
- Understand creating and accessing elements in dictionaries.
- Learn common dictionary use cases and operations.
- Use built-in dictionary methods effectively.
- Implement programs using dictionaries.

2.1 INTRODUCTION

Python has a rich collection of built-in data structures, dictionary is one of them. The dictionary holds information as key-value pairs. It provides a way to access data by using meaningful association of keys and its corresponding values.

In this chapter we will create, access, and manipulate dictionaries using Python's built-in functions and methods. We will also look into different operations that can be performed in the dictionary.

2.2 DICTIONARY AND ITS OPERATIONS

Dictionary is a data structure in which we store values as a pair of keys and its associated value. Each key is separated from its value by using a colon (:), and consecutive items of the dictionary are separated by using a comma. Keys are unique and immutable. Values can be of any data type and can repeat. The entire items in a dictionary are enclosed in curly brackets { }.

The general syntax of creating a dictionary is as mentioned:

dictionary_name = {key 1 : value 1, key 2 : value 2, ..., key n : value n}

The Python program given in Example 2.1 shows how to create a dictionary.

Example 2.1: Creation of dictionary

```
# Program to create a dictionary
dict = {
                "Rollno": "A01",
                "Name": "Advika",
                "Course": "MScIT"
        }
print("Dictionary:", dict)
```

OUTPUT:

Dictionary: {'Rollno': 'A01', 'Name': 'Advika', 'Course': 'MScIT'}

In the above program we have created a dictionary with name *dict*. The dictionary 'dict' contains three key-value pairs. Each key i.e "Rollno", “Name” and “Course” is unique, and its corresponding values are provided after colon.

The dictionary has been defined and initialized in multiple lines for better understanding.

Alternatively we can write the definition in a single line as mentioned:
dict = {"Rollno": "A01", "Name": "Advika", "Course": "MScIT"}

The print function is used to display the contents stored in the dictionary as can be seen in the output.

The dictionary can be used in many applications. Their use is ideal in following few cases:

- Mapping relationships (e.g., student roll numbers to names).
- Storing configuration data.
- Representing structured data (e.g., JSON objects).

Let us now look at some common operations that can be performed using a dictionary.

Accessing Values

The first and foremost operation many users perform is to access the values within a dictionary once it has been defined. The values in the dictionary are accessed with the help of a key. The Python program given in Example 2.2 shows how to access values from a dictionary.

Example 2.2: Accessing values from dictionary

```
# Program to access value from a dictionary
dict = {"Rollno": "A01", "Name": "Advika", "Course": "MScIT"}
print("Rollno: ",dict['Rollno'])
print("Name: ",dict['Name'])
print("Course: ",dict['Course'])
```

OUTPUT:

```
Rollno:  A01
Name:  Advika
Course:  MScIT
```

In the above program we have used the contents of the dictionary 'dict' used in Example 2.1. As can be seen in the program the values of the key's Rollno, Name and Course is accessed by using square brackets as shown in statements, *print("Rollno: ",dict['Rollno']), print("Name: ",dict['Name'])* and *print("Course: ",dict['Course'])*. It returns corresponding values A01, Advika and MScIT as seen in the output.

Adding a pair in a Dictionary

To add a new entry or a key-value pair in a dictionary, we need to specify the key-value pair as we had done for the existing pairs at the time of definition. The general syntax to add an element in a dictionary is as mentioned:

dictionary_ variable[new_key]= new_val

The Python program given in Example 2.3 shows how to add a new pair value in a dictionary.

Example 2.3: Adding new value pair in dictionary

```
# Program to access value from a dictionary
dict = {"Rollno": "A01", "Name": "Advika", "Course": "MScIT"}
print("Rollno: ",dict['Rollno'])
print("Name: ",dict['Name'])
print("Course: ",dict['Course'])
#new entry
dict["Semester"]=3
print("Semester added later: ",dict['Semester'])
```

OUTPUT:

```
Rollno:  A01
Name:  Advika
Course:  MScIT
Semester added later:  3
```

The program shown in Example 2.3 is similar to that of Example 2.2. Here we have added a new key value pair using statement *dict["Semester"]=3*. The key Semester thus becomes the fourth key of the dictionary and is assigned value 3.

Modifying a value associated with key in a Dictionary

To modify a value associated with a particular key, we can just overwrite the existing value. It is similar to adding a new pair, except that the key name remains the same.

The Python program given in Example 2.4 shows how to modify a value associated with a particular key in a dictionary.

Example 2.4: Modifying value associated to a key in dictionary

```
# Program to modify value associated to a key in dictionary
dict = {"Rollno": "A01", "Name": "Advika", "Course": "MScIT"}
print("Rollno: ",dict['Rollno'])
print("Name: ",dict['Name'])
print("Course: ",dict['Course'])
```

```
#Updated entry
dict["Course"]="BScIT"
print("Course after update: ",dict['Course'])
```

OUTPUT:

Rollno: A01

Name: Advika

Course: MScIT

Course after update: BScIT

The program shown in Example 2.3 is similar to that of Example 2.2. Here we have modified the value associated with the key Course by using statement *dict["Course"]="BScIT".* Thus, the initial output of the key Course is MScIT which later gets modified to BScIT.

Note:

A key in a dictionary itself cannot be modified as it is immutable. But it is possible to delete the key.

Deleting Items in dictionary

There are multiple ways that can be used to delete a key value pair in a dictionary. It is possible to delete a specific key or an entire dictionary.

Using del

We can delete one or more key value pairs or an entire dictionary using the ***del*** keyword. The general syntax to delete a particular key is as mentioned:

del dictionary_name[key]

Similarly, an entire dictionary can be deleted using the syntax:

del dictionary_name

The Python program given in Example 2.5 shows how to delete a particular key in a dictionary.

Example 2.5: Delete operation in dictionary

```
# Program to show delete operation in dictionary
dict = {"Rollno": "A01", "Name": "Advika", "Course": "MScIT"}
print(dict)
#Delete course
del dict["Course"]
print("After deleting Course")
print("Rollno: ",dict['Rollno'])
print("Name: ",dict['Name'])
print("Course: ",dict['Course'])
```

OUTPUT:

```
{'Rollno': 'A01', 'Name': 'Advika', 'Course': 'MScIT'}
After deleting Course
Rollno:  A01
Name:  Advika
Traceback (most recent call last):
  File "32_5.py", line 11, in <module>
    print("Course: ",dict['Course'])
KeyError: 'Course'
```

The program shown in Example 2.5 has a dictionary with three key value pairs. Initially when we print the contents of each key the associated value gets printed. Then we have deleted the key Course by using statement *del dict["Course"]*. Now when we try to print the values again, we get the associated values of the first two keys. The statement *print("Course: ",dict['Course'])* at line 13, though generates an error KeyError: 'Course' indicating that the said key is not present in the dictionary.

Let us also see an example of how to delete an entire dictionary. The Python program given in Example 2.6 shows how to delete a dictionary.

Example 2.6: Delete a dictionary

```
# Program to delete a dictionary
dict = {"Rollno": "B01", "Name": "Dhyey", "Course": "MScIT"}
print(dict)
#Delete dictionary
del dict
print(dict)
```

OUTPUT:

```
{'Rollno': 'B01', 'Name': 'Dhyey', 'Course': 'MScIT'}
Traceback (most recent call last):
  File "32_6.py", line 7, in <module>
    print(dict)
NameError: name 'dict' is not defined
```

The program shown in Example 2.6 has a dictionary with three key value pairs. Initially when we print the contents of each key the associated value gets printed. Then we have deleted the dictionary by using statement *del dict*. Now when we try to print the values again, print statement at line 7, though generates an error “NameError: name 'dict' is not defined” indicating that the said dictionary has been removed from the current scope. After this, the dict variable no longer exists in memory.

pop( ) method

Another method to remove or delete key-value pairs from a dictionary is the pop( ) method. The general syntax of using pop( ) method is as mentioned:

dict_name.pop('key')

The Python program given in Example 2.7 shows how to use a pop( ) method in a dictionary.

Example 2.7: Use of pop() method in a dictionary

```
# Program to pop an element from a dictionary
dict = {"Rollno": "B01", "Name": "Dhyey", "Course": "MScIT", "Sem": 2}
print("Before Removal:", dict)
removed_value = dict.pop("Course")
print("Removed 'Course':", removed_value)
print("After Removal:", dict)
```

OUTPUT:

```
Before Removal: {'Rollno': 'B01', 'Name': 'Dhyey', 'Course': 'MScIT', 'Sem': 2}
Removed 'Course': MScIT
After Removal: {'Rollno': 'B01', 'Name': 'Dhyey', 'Sem': 2}
```

The program shown in Example 2.7 has a dictionary with four key value pairs. Initially when we print the contents of each key the associated value gets printed. Then we have popped the key ‘Course’ from the dictionary by using statement *removed_value = dict.pop("Course")*. The value that has been removed using pop is stored in the variable removed_value. Now when we try to print the dictionary values again, we get the output After Removal: {'Rollno': 'B01', 'Name': 'Dhyey', 'Sem': 2}.

in operator

The *in* operator checks whether the specified key exists in the given dictionary or not. It returns a boolean value true if the key is present in a dictionary otherwise it returns false. The general syntax of using the in operator is as mentioned:

key in dictionary_name

The Python program given in Example 2.8 shows how to use the *in* operator in a dictionary.

Example 2.8: Use of in operator within a dictionary

```
# Program to use in operator within a dictionary

dict = {"Rollno": "A01", "Name": "Advika", "Course": "MScIT"}

print("Is 'Name' a key?", "Name" in dict)

print("Is 'Sem' a key?", "Sem" in dict)
```

OUTPUT:

```
Is 'Name' a key? True

Is 'Sem' a key? False
```

The program shown in Example 2.8 has a dictionary with three key value pairs. The statement *print("Is 'Name' a key?", "Name" in dict)* checks if the key called Name exists in the dictionary, it returns value ‘True’ as the key is part of the dictionary. The next statement *print("Is 'Sem' a key?", "Sem" in dict)* checks if the key called Sem exists in the dictionary, it returns value ‘False’ as the key is not part of the dictionary.

Iterating through a dictionary

One of the basic operations performed on the dictionary would be to iterate through the entire dictionary key value pairs one by one. It is possible to iterate through a dictionary using key, value or key-value pairs. The general syntax to use this feature is as mentioned:

Iteration using	**Syntax**
Key	***for key in dictionary_name.keys( ):***
Value	***for value in dictionary_name.values( ):***
Pair	***for key, value in dictionary_name.items( ):***

The Python program given in Example 2.9 shows how to iterate through the dictionary using all the above mentioned options.

Example 2.9: Iteration within a dictionary

```
# Program to show iteration within a dictionary
dict = {"Rollno": "A01", "Name": "Advika", "Course": "MScIT", "Sem":2}
print("*** Iteration using key ***")
for key in dict.keys():
    print(key)
print("*** Iteration using value ***")
for value in dict.values():
    print(value)
print("*** Iteration using pair ***")
for key, value in dict.items():
    print(key,": ",value)
```

OUTPUT:

```
*** Iteration using key ***
Rollno
Name
Course
Sem
*** Iteration using value ***
A01
Advika
MScIT
2
*** Iteration using pair ***
Rollno :  A01
Name :  Advika
Course :  MScIT
Sem :  2
```

The program shown in Example 2.9 has a dictionary with four key value pairs. Observe the output, here for the statement *for key in dict.keys():* we are getting in output only the keys. The print statement after *for value in dict.values():* displays only values pertaining to each key. Lastly we get a list of all key value pairs.

Check Your Progress-1

a) A dictionary's keys are immutable and distinct. (True/False)
b) A key-value pair is deleted from a dictionary using the pop() method, but the deleted value is not returned. (True/False)
c) The in operator can be used to check if a key exists in a dictionary. (True/False)
d) Both the entire dictionary and a particular key-value pair can be removed with the del keyword. (True/False)
e) The items() method in a dictionary returns only the keys. (True/False)

2.3 DICTIONARY METHODS

Python has a rich collection of built-in methods that can be used with a dictionary. We already saw the use of methods like keys(), values() and items(). Let us explore some more methods in this section.

len()

The len() method is used to find the length of the dictionary. It provides the total number of items (key-value pairs) within a dictionary.

get(key, default)

The get() method is used to retrieve the value for the specified key. It can also be used to give a default value if the key doesn't exist in the dictionary.

clear()

The clear() method is used to remove all elements from the dictionary. Thus after performing this operation the dictionary will not contain any key value pair.

update(other_dict)

To merge two dictionaries, we use the update() method. It updates the values of any keys that are already present in the original dictionary by appending key-value pairs from one dictionary to another. The key is inserted as a new key-value pair if it is not present in the original dictionary.

The Python program given in Example 2.10 demonstrates the use of all the above methods.

Example 2.10: Use of different methods

```
# Program to use different methods
dict = {"Fruit": "Apple", "Vegetable": "Onion", "Flower": "Rose", "Animal":
"Dog"}
print("Fruit:", dict.get("Fruit"))
print("Tree :", dict.get("Tree", "Not Found"))
print("Length: ", len(dict))
dict.update({"Tree": "Banyan", "Fruit": "Grapes"})
print("After Update:", dict)
dict.clear()
print("After clear:", dict)
```

OUTPUT:

```
Fruit: Apple
Tree : Not Found
Length:  4
After Update: {'Fruit': 'Grapes', 'Vegetable': 'Onion', 'Flower': 'Rose', 'Animal':
'Dog', 'Tree': 'Banyan'}
After clear: {}
```

The program shown in Example 2.10 has a dictionary with four key value pairs. The statement *print("Fruit:", dict.get("Fruit"))* checks if the key called Fruit exists in the dictionary, as can be seen in the dictionary keys we have the said key. Thus we get the value of Apple as output. The next statement *print("Tree :", dict.get("Tree", "Not Found"))* checks if the key called Tree exists in the dictionary, as the key is not part of the dictionary it returns value 'Not Found' specified as default value. The statement *print("Length: ", len(dict))* prints 4 as output as there are four key value pairs in the dictionary. The statement *dict.update({"Tree": "Banyan", "Fruit": "Grapes"})* updates the current dictionary. The operation adds a key Tree with value Banyan and updates the value of key Fruit to Grapes. The statement *dict.clear()* empties the dictionary and thus when we try to print the dictionary we get an empty set {} as output.

Check Your Progress-2

a) The len() method returns the total number of key-value pairs in a dictionary. (True/False)
b) The get() method always raises an error if the specified key does not exist in the dictionary. (True/False)
c) After using the clear() method, the dictionary will be represented as an empty dictionary {}. (True/False)
d) The update() method merges another dictionary into the existing one, updating values for matching keys and adding new key-value pairs. (True/False)

2.4 LIST vs DICTIONARY

We have learnt about lists in the previous chapter. List is a sequence. In this section we will discuss the difference between a list and a dictionary.

List vs Dictionaries

Having seen what a dictionary is and how to create it, let us now look at the difference between a list and a dictionary. Table 2.1 shows the differences.

Table 2.1: difference between List and Dictionary

List	Dictionary
An ordered collection of elements accessed by index.	An unordered collection of key-value pairs accessed by key.
Access elements using integer index.	Access elements using keys which can be descriptive labels.
Elements can repeat.	Keys must be unique, but values can repeat.
Best for ordered collections (e.g., sequences).	Best for mapping relationships (e.g., data lookups).

Check Your Progress-3

a) A list is an ordered collection of elements accessed by index. (True/False)

b) In a list, each element must be unique. (True/False)

c) A dictionary is best used for ordered collections like sequences. (True/False)

2.5 LET US SUM UP

In this unit, we explored the concept of a dictionary available in Python. Dictionary is a powerful and versatile data structure used to store data as key-value pairs. We looked at the syntax of the dictionary and its distinctive features. Dictionaries are excellent for use cases such as facilitating fast lookups, organizing structured data, and mapping relationships. We also learnt how to use fundamental methods like keys(), values(), items(), get(), and update() that make dictionary management easier. Common actions like adding, updating, accessing, and removing key-value pairs were also demonstrated.

2.6 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

1-a True
1-b False
1-c True
1-d True
1-e False
2-a True
2-b False
2-c True
2-d True
3-a True
3-b False
3-c False

2.7 ASSIGNMENTS

1. Define a dictionary with an appropriate example.
2. What is the main difference between lists and dictionaries?
3. How do you add a new key-value pair to a dictionary?
4. Name three methods of dictionaries and their purposes.
5. How do you remove an element from a dictionary?
6. Write a Python program
   - To create a dictionary
   - To add an element to dictionary
   - To display the length of the dictionary.
   - To update an element in the dictionary.
   - To remove all elements from the dictionary.
7. Create a dictionary to store details about your favorite book (title, author, year, category). Create a menu driven program which allows three operations: Add, update, and remove keys.
8. Write a program to count the occurrences of each character in a string using a dictionary.
9. Implement a phonebook using a dictionary where names are keys and phone numbers are values.
10. Write a program to merge two dictionaries.
11. Write a Python program to get the top three items in a shop. Sample data: {'item1': 45.50, 'item2':35, 'item3': 41.30, 'item4':55, 'item5': 24}
    Expected Output:
    item4: 55
    item1: 45.5
    item3: 41.3

Unit-3: Sets 3

Unit Structure

3.0 LEARNING OBJECTIVE

After studying this unit student should be able to:

- Understand the concept of sets in Python.
- Differentiate between lists and sets.
- Recognize use cases for sets and perform basic set operations.
- Use built-in set methods effectively.

3.1 INTRODUCTION

To store multiple items using a single variable in Python we can use its built-in data type called *set*. Sets are mutable, unordered, and do not allow use of duplicate elements. The sets thus are helpful in situations where membership checks and uniqueness are required. We can add, remove, or modify elements dynamically within a set as they are mutable.

In this chapter we will create, access, and manipulate sets using Python's built-in functions and methods. We will also look at other operations that can be performed on sets.

3.2 SETS USECASES AND OPERATIONS

A set is an iterable, unordered collection data type in Python that has no duplicate elements. Even while sets are changeable, that is, you may add or remove components after they are created, individual things within a set must be immutable and cannot be changed directly. The set is represented by values enclosed in curly braces { }.

The general syntax used to create a set is as mentioned:

set_name = [value1, value2, ..., valueN]

The Python program in Example 3.1 shows how to create and print the values of a set.

Example 3.1: Creation of Set

```
# Program to create a set of integers and strings
num_set = {10, 20, 30, 40, 50, 10, 20}
print(num_set)
# Set of string values
fruits_set = {"Apple", "Banana", "Cherry", "Grapes"}
print(fruits_set)
```

OUTPUT-SCENARIO 1:

```
{40, 10, 50, 20, 30}
{'Grapes', 'Apple', 'Cherry', 'Banana'}
```

OUTPUT-SCENARIO 2:

```
{40, 10, 50, 20, 30}
{'Cherry', 'Apple', 'Grapes', 'Banana'}
```

In the above program the two sets *num_set* and *fruits_sets* have been created and initialized with a set of integer and string values respectively. The print function is used to display the contents stored in both the sets as can be seen in the output.

The set *num_set* is initialized with seven elements and the set *fruits_set* is initialized with four elements. Observe that both the outputs show only five elements when we try to print the values of the set *num_set*. Here the duplicate values are ignored. Further we can see that the order in which the values are initialized in both sets and the order in which they are displayed has changed. The set element thus cannot be referred to by an index or a key.

Note:

It is possible that the users may get different output then the ones shown in OUTPUT-SCENARIO 1 and OUTPUT-SCENARIO 2.

Use cases of Sets

Sets in Python are commonly used for removing duplicates or data filtering, membership testing and performing mathematical operations like unions, intersections, and differences on sets. Let us now see how these operations are performed.

Removing Duplicates or data filtering

Eliminating duplicate values from a collection is one of the most popular applications for sets. Duplicate values are automatically eliminated when we convert a list to a set because sets can only contain unique objects. The Python program in Example 3.2 shows how to create a list that contains duplicate values and generate a set with unique values.

Example 3.2: Removing duplicates from list and creating a set

 # Program to remove duplicates from list and create a set *fruits_list = ["Apple", "Banana", "Cherry", "Grapes", "Banana", "Mango"]* *unique_data = set(fruits_list)* *print(unique_data)*
OUTPUT-1: {'Banana', 'Mango', 'Grapes', 'Cherry', 'Apple'}
OUTPUT-2: {'Cherry', 'Mango', 'Grapes', 'Apple', 'Banana'}

In the above program a list *fruits_list* has been created and initialized with six elements, where the term “Banana” is duplicated. The statement *unique_data = set(fruits_list)* first converts the list into a set and then assigns it a name *unique_data*. Thus, when we print the value in the set *unique_data*, both the outputs show only five elements as the duplicate values have been removed. Further we can see that in both the outputs the order of display of elements is not the same.

Membership Testing

When it comes to determining if an element is a member of the collection, sets are highly effective. As sets employ hash tables internally, membership testing (both *in* and *not in*) with sets is quicker than with lists. The Python program in Example 3.3 shows how to perform membership testing.

Example 3.3: Example of membership testing in a set

```
# Program to show membership testing in a set
fruits = {"Apple", "Banana", "Cherry", "Grapes", "Mango"}
print("Apple" in fruits)
print("Guava" not in fruits)
print("Guava" in fruits)
```

OUTPUT:

```
True
True
False
```

In the above program a set named *fruits* has been created and initialized with five elements. The statement *print("Apple" in fruits)* gives output as True, as the element is part of the set. The statement *print("Guava" not in fruits)* gives output as True, as the element is not part of the set. Similarly, when the *in* membership is tested for "*Guava"* the statement *print("Guava" in fruits)* gives output as False, as the element is not part of the set.

Mathematical Operations on Sets

In mathematics the use of operations like union, intersection, difference, and symmetric difference are widely used. These mathematical operations can be performed easily using sets in Python. These operations are performed with sets using the symbols as mentioned:

- **Union (|) :** Combines elements from two sets.
- **Intersection (&) :** Finds common elements from two sets.
- **Difference (-):** Finds elements in one set but not the other.
- **Symmetric Difference (^):** Finds elements in either set but not in both.

The Python program in Example 3.4 shows how to perform mathematical operations using sets.

Example 3.4: Example of mathematical operations on set

```
# Program to show example of mathematical operations on set
set1 = {1, 2, 3, 5, 8, 13}
set2 = {3, 4, 5, 6, 7, 8}
u_set = set1 | set2
print("Union: ",u_set)
i_set = set1 & set2
print("Intersection: ",i_set)
d_set = set1 - set2
print("Difference: ", d_set)
sd_set = set1 ^ set2
print("Symmetric Difference: ", sd_set)
```

OUTPUT:

```
Union:  {1, 2, 3, 4, 5, 6, 7, 8, 13}
Intersection:  {8, 3, 5}
Difference:  {1, 2, 13}
Symmetric Difference:  {1, 2, 4, 6, 7, 13}
```

In the above program two sets; *set1* and *set2* have been created and initialized with six integer elements each. The statement *u_set = set1 | set2* creates a new set u_set as a union of the two sets. When we print the value of u_set, we get a set with nine elements after removal of duplicate elements from both sets.

The statement *i_set = set1 & set2* creates a new set i_set as an intersection of the two sets. When we print the value of i_set, we get a set with three elements that are common in set1 and set2.

The statement *d_set = set1 - set2* creates a new set d_set as a difference of the two sets. When we print the value of d_set, we get a set with three elements that are present in set1 but not present in set2.

Similarly, the statement *sd_set = set1 ^ set2* creates a new set sd_set as a symmetric difference of the two sets. When we print the value of sd_set, we get a set with six elements that are present in either set1 or set2 but not in both.

Check Your Progress-1

a) A set is an unordered collection of unique elements in Python. (True/False)
b) A key or index can be used to access the elements in a set. (True/False)
c) When a list is converted into a set, duplicate values are automatically removed. (True/False)
d) The mathematical operation Symmetric Difference (^) finds elements in either set but not in both. (True/False)
e) The union of two sets includes only the common elements between them. (True/False)

3.3 SETS AND METHODS

Python has a rich collection of built-in methods that can be used with sets. These methods can be used to add, remove, update or perform different operations on set. Let us explore some of these methods in this section.

add()

The add method is used to add a single element to an existing set. If the element we are trying to add already exists then no change occurs.

remove()

The remove method is used to remove the specified element from an existing set. While trying to remove the element if the element is not found then the program raises a KeyError and terminates.

discard()

The discard method is similar to remove method i.e. it removes the specified element from an existing set, but it does not raise an error if the element is not found.

update(list)

The update method is used to add multiple elements to the set from an iterable like list or tuple.

pop()

The pop method is used to remove and return an arbitrary element from the set. It raises a KeyError if the set is empty.

clear()

The clear method is used to remove all elements from the given set. If the set the user is trying to clear does not exist then the program raises a NameError and terminates. The Python program given in Example 3.5 demonstrates the use of all the above methods.

Example 3.5: Use of different methods

```
# Program to use different methods with set
set1 = {1, 2, 3, 5, 7}
print("Initial Set: ",set1)
set1.add(4)
print("Set after adding element: ",set1)
set1.remove(3)
print("Set after removing element: ",set1)
set1.discard(9)
set1.update([2, 3, 13, 14, 15])
print("Set after updating element: ",set1)
popped_element = set1.pop()
```

```
print("Popped Element: ",popped_element)
print("After pop set1: ",set1)
set1.clear()
print(set1)
```

OUTPUT:

```
Initial Set:  {1, 2, 3, 5, 7}
Set after adding element:  {1, 2, 3, 4, 5, 7}
Set after removing element:  {1, 2, 4, 5, 7}
Set after updating element:  {1, 2, 3, 4, 5, 7, 13, 14, 15}
Popped Element:  1
After pop set1:  {2, 3, 4, 5, 7, 13, 14, 15}
set()
```

The program shown in Example 3.5 has a set with five elements {1, 2, 3, 5, 7}. The statement *set1.add(4)* adds a new element in the set. Thus, when we print the set, we get six elements {1, 2, 3, 4, 5, 7} as output. The statement *set1.remove(3)* removes the element 3 from the set. When we print this new set, we get only five elements {1, 2, 4, 5, 7} as output. The statement *set1.discard(9)* tries to remove an element having value 9, though it is not part of the set no error is generated.

The statement *set1.update([2, 3, 13, 14, 15])* updates the set. The operation adds three new values 13, 14 and 15 into the set. The values 2 and 3 are ignored as they are already part of the set. The updated set now contains nine elements {1, 2, 3, 4, 5, 7, 13, 14, 15}. The statement *popped_element = set1.pop()* removes the element with value 1 and assigns it to variable *popped_element*. Thus, we now get a set with eight elements {2, 3, 4, 5, 7, 13, 14, 15}. Lastly, the statement *set1.clear()* removes all the elements from the set and so when we try to print the set, we get an empty set() as output.

Check Your Progress-2

a) A set can have more than one element added at once using the add() method. (True/False)
b) The discard() method raises a KeyError if the element to be removed is not present in the set. (True/False)
c) The update() method can be used to add elements from an iterable like a list or tuple to a set. (True/False)
d) The pop() method removes and returns an arbitrary element from the set, raising a KeyError if the set is empty. (True/False)
e) The set is empty when all of its elements are removed using the clear() method. (True/False)

3.4 LIST vs SET

The set items are unchangeable and are not sorted, also duplicate values are not permitted in the set. The term "unordered" here refers to a set of elements that lack a clear order. Set objects cannot be accessed by index or key, and they may display in a different order each time you use them. Any two items in a set cannot have the same value. Lists and sets are both collections of items, although they differ in many ways. The difference between list and set is shown in Table 3.1.

Table 3.1: Difference between List and Set

Feature	Set	List
Definition	A collection of unordered, unique items.	An ordered collection of items.
Mutability	Mutable (It is possible to add or remove elements)	Mutable (It is possible to add, remove, or modify elements).
Duplicates	Does not allow duplicate elements as a part of a single set. When a set is created or updated duplicates are automatically removed.	Allows duplicate elements.

Indexing	Not supported	Supports indexing
Use Cases	Useful for membership testing, removing duplicates, and mathematical operations like union or intersection.	Useful for maintaining a sequence of items with order and duplicates.
Syntax	Defined using curly braces {}	Defined using square brackets []

3.5 LET US SUM UP

In this chapter, you have learned how to use a set. Sets are unordered collections that do not allow duplicates. We also learnt how to perform different set operations like union, intersection, difference and symmetric difference. We looked into common methods available for manipulating sets in Python. Finally, we compared and looked at differences between set and list.

3.6 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

1-a True
1-b False
1-c True
1-d True
1-e False
2-a False
2-b False
2-c True
2-d True
2-e True

3.7 ASSIGNMENTS

1. What is a set?
2. Write the difference between set and list.
3. What will be the output of the following code?

```
set_x = {1, 2, 3, 3}
print(len(set_x))
```

4. Name two methods to remove elements from a set.
5. List types of basic set operations.
6. Create a Python script that takes a list of numbers and returns a set of unique numbers.
7. Write a program to find the union and intersection of two sets.
8. Write a program to find Unique Words in a Sentence.
9. Write a program to find Elements Present in One Set but Not in Another.
10. Write a program that counts Unique Characters in a String.

Unit-4: Strings 4

Unit Structure

4.0 LEARNING OBJECTIVE

After studying this unit student should be able to:

- Learn how to create, manipulate, and format strings.
- Understand what strings are and their role.
- Explore various string methods and operations.
- Understand use of regular expressions.

4.1 INTRODUCTION

A string is a collection of characters. Strings are important because they contain the majority of the data we use on a regular basis. For instance, if we wish to store student information, such as their name, address, course, email address, etc., all of these are strings. A string is represented by the str datatype in Python. Python doesn't have an individual datatype for character; it handles strings and characters almost similarly.

In this chapter we will learn about what a string is and how to perform different operations on the strings.

4.2 STRING MANIPULATION AND FORMATTING

A string can be created in Python by allocating a collection of characters to a string variable. To create a text string, we need to enclose the set of characters within a single, double, or triple quotation marks. The general syntax used to create a string is as mentioned:

string_variable = 'string value'

string_variable = “string value”

string_variable = '''string value'''

The example of how to create a string in a Python program is given in the program of Example 4.1.

Example 4.1: Creation of String

```
# Program to create a string using single, double and triple quotes
single_quote = 'Hello'
double_quote = "World"
triple_quote = '''Welcome to Core Python Programming!
Here is an example of a string with triple quote...
String declaration demo.... '''
print(single_quote)
print(double_quote)
print(triple_quote)
```

OUTPUT:

```
Hello
World
Welcome to Core Python Programming!
Here is an example of a string with triple quote...
String declaration demo....
```

In the above program the three strings, *single_quote*, *double_quote* and *triple_quote* have been created and initialized with string values respectively. The print function is used to display the contents stored in all string variables as can be seen in the output. There is no difference between the strings defined using a single and double quote, both these strings work in the similar manner. However, triple quotes are usually utilized to define multi-line strings. Observe that when we print the value of the string stored in variable *triple_quote*, we get a display in multiple lines.

String Indexing

Python strings are character sequences, therefore indexing can be used to retrieve individual characters. Strings are indexed starting from 0 and -1 from end. The general syntax to access a string element is as mentioned:

string_variable[index]

By specifying the position number through an index, we can refer to the individual elements of a string. Example 4.2 shows how to access and print the value of an element at a specific index.

Example 4.2: Accessing elements of a String using index

```
# Program to access elements from string
str1 = "Python"
print(str1[0])
print(str1[-1])
```

OUTPUT:

P

n

The string *str1* in the program above has been initialized with the value 'Python'. The print function is used to print the value of the element at a specific index. Here, we have demonstrated how to use an index to access a string element both forward and backward. The statement *print(str1[0])* thus, gives output 'P' from the beginning of the string, while the statement *print(str1[-1])* gives output 'n' from the end of the string based on the negative index.

String Slicing

It is a process of extracting multiple characters from the string. We can give the range of indexes to fetch a substring from the given string. Like indexing, we have to specify the range of characters with the use of colon (:) inside the square bracket. For example, str[0:4] will return the first 5 characters from index 0 to 4 of the string variable *str*. If we omit the first parameter in range, it will automatically fetch characters from the first character (index 0) to a given range character. Hence, the statement str[:4] is equivalent to the statement str[0:4]. If we omit the last number in the range, then it will return all the characters up till the last character within the string, starting from the specified character. The statement str[4:] will return all the characters starting from index 4 till the end of the string.

By providing the start and end indexes, slicing allows us to retrieve a portion of a string. The general syntax to access a string element is as mentioned:

string_variable[start: stop: step]

Where:

start is the index to begin slicing (default is 0)

end is the index to stop slicing (exclusive).

step is the interval (default is 1)

If the values for *start* and *stop* parameters are not specified, then slicing is done from 0^{th} to $n\text{-}1^{th}$ elements. If the value of parameter *step* is not specified, then the default value 1 is considered. Program in Example 4.3 shows how to access and print the value of a string using the concept of slicing.

Example 4.3: Accessing elements of a String using slicing

```
# Program to access elements from a string
str1 = "Python"
print(str1[0:3])        # Substring from index 0 to 2
print(str1[::2])        # Characters at every second position
```

OUTPUT:

```
Pyt
Pto
```

Here, we have demonstrated how to access portions of string using the concept of slicing. The statement *print(str1[0:3])* extracts substring from index 0 to 2. The statement *print(str1[::2])* extracts Pto, that is it extracts characters at every second position of the original string.

String Formatting

At times we need to join the values of a string and an integer variable. It is though not possible to concatenate a string and in integer using '+' operator as learnt in string concatenation operation earlier. In Python the code given in Example 4.4 will generate an error.

```
#Program with error
age = 36
person_detail = "My name is Tejas, I am " + age
print(person_detail)
```

OUTPUT:

```
Traceback (most recent call last):
  File "test.py", line 2, in <module>
    person_detail = "My name is Tejas, I am " + age
TypeError: can only concatenate str (not "int") to str
```

It is possible to combine strings and numbers by using the *f-string* or the format() method. The *f-string* was introduced in Python 3.6, and is now the preferred way of formatting strings. To specify a string as an *f-string*, simply put a character '*f*' as a prefix of the string literal, and add curly brackets {} as placeholders for variables and other operations. The general syntax to use *f-string* is as mentioned:

f'write string here { variable name } with remaining string'

Example 4.5 shows how to access and print the value of a string with the use of f-string.

Example 4.5 Creating string with the use of f-string

```
# Program to use string and variable with f-string
name = "Tejas"
age = 36
city = "Ahmedabad"
personDetail = f"My name is {name}, I am {age} years old and live in {city}."
print(personDetail)
```

OUTPUT:

```
My name is Tejas, I am 36 years old and live in Ahmedabad.
```

Here, the statement, *"My name is {name}, I am {age} years old and live in {city}."* is f-string. We can use single quote, double quote or triple quote with f-string. Using the f-string we are able to concatenate the string and values of variables mentioned within the curly brackets {}. Thus, here name is replaced with a string

“Tejas”, age is replaced with an integer “36” and city is replaced with a string “Ahmedabad” in the output. This formatting syntax is powerful and easy to use.

Check Your Progress-1

a) In Python, multi-line strings are usually created with triple quotes. (True/False)
b) In Python, string indexing starts from 1 for the first character of the string. (True/False)
c) Negative indexing in Python allows access to characters from the end of the string. (True/False)
d) Slicing a string in Python requires specifying all three parameters: start, stop, and step. (True/False)
e) The f-string formatting method in Python requires placing curly brackets {} around variables to include their values in the string. (True/False)

4.3 STRING METHODS AND OPERATIONS

Many operations can be performed with strings, which makes it one of the most used data types in Python. We can perform many operations on strings using many of the available built-in Python methods. All the Python's string methods return a new string with the modified characteristics. These methods do not alter the original string. Let us now have a look at some inbuilt string methods available in Python.

String Methods

Table 4.1 gives the list of methods that can be used with string. It also gives its description and syntax.

Table 4.1: List of String Functions

Method	Description	Syntax
len()	Returns the length of a string.	len(string)

upper()	Converts all characters in a string to uppercase.	string.upper()
lower()	Converts all characters in a string to lowercase.	string.lower()
find()	Returns the lowest index in the string where the substring is found.	string.find(substring)
strip()	Removes the leading and trailing whitespaces.	string.strip()
replace()	Replaces the occurrences of a substring within a string.	string.replace(old, new)
split()	Splits the string at the specified delimiter and returns a list of substrings.	string.split(delimiter)
join()	Concatenates the elements of an iterable with a specified separator.	string.join(iterable)
startswith()	Checks if the string starts with the specified prefix.	string.startswith(prefix)
endswith()	Checks if the string ends with the specified suffix.	string.endswith(suffix)

Let us now write a program that uses all the methods mentioned in Table 4.1. Examples 4.6 and 4.7 show how to use these methods with strings.

Example 4.6: Using inbuilt string methods

```
# Program to use inbuilt string methods
string1="Python"
string2="  Python Programming  "
lenOfstring1 = len(string1)
print("Length of string1: ",lenOfstring1)
lowerCaseString = string1.lower()
```

```
print("Lower Case string1: ",lowerCaseString)
upperCaseString = string1.upper()
print("Upper Case string1: ",upperCaseString)
subString = string2.find(string1)
print("Lowest index at which string1 is found: ",subString)
strippedString = string2.strip()
print("Stripped string2: ",strippedString)
print(f"Length of original string2 = {len(string2)}")
print(f"Length of stripped string2 = {len(strippedString)}")
```

OUTPUT:

```
Length of string1:  6
Lower Case string1:  python
Upper Case string1:  PYTHON
Lowest index at which string1 is found:  2
Stripped string2:  Python Programming
Length of original string2 = 22
Length of stripped string2 = 18
```

The program demonstrates the use of various inbuilt string methods available in Python. The *len()* method returns a length of *string1* which is 6. The *lower()* and *upper()* methods convert the value "Python" of *string1* to its lowercase equivalent ("python") and uppercase equivalent ("PYTHON") respectively. The *find()* method locates the substring "Python" (value of string1) within string "Python Programming" (value of string2), it returns the index 2, as we have added two blank spaces in the beginning of the *string2*. The *strip()* method removes leading and trailing spaces from the string "Python Programming" (value of string2). The difference can be observed when we print the lengths of variables *string2* and *strippedString*. The length of the original string is shown as 22, while that of the stripped string is 18. This indicates that four blank spaces have been removed from the original string.

Example 4.7: Second example of using inbuilt string methods

```
# Program to use inbuilt string methods
string = "Python Programming"
newString = string.replace("Python", "Java")
print("New string after word replacement: ",newString)
string1 = "Python,Java,C++"
list_languages = string1.split(",")
print("Split List: ",list_languages)
print("Joined using |: "," | ".join(list_languages))
print("Starts with word Python: ",newString.startswith("Python"))
print("Ends with word Programming:", newString.endswith("Programming"))
```

OUTPUT:

```
New string after word replacement:  Java Programming
Split List:  ['Python', 'Java', 'C++']
Joined using |:  Python | Java | C++
Starts with word Python:  False
Ends with word Programming:  True
```

In the above program the *replace()* method substitutes the substring "Python" with "Java" in the variable *string* and assigns the new string to variable *newString*. The resulting string now is "Java Programming". The *split()* method divides the contents of *string1* "Python,Java,C++" using ‘,’ as a separator and creates a list named *list_languages*. The contents of the variable *list_languages* thus become ['Python', 'Java', 'C++']. The *join()* method combines this list into the string "Python | Java | C++" using " | " as a separator. The *startswith()* method checks whether the variable newString that contains string "Java Programming" begins with "Python". It returns False. Similarly, the *endswith()* method checks if the variable *newString* ends with "Programming", it returns True.

Check Your Progress-2

a) The total number of characters in a string is returned by the len() function. (True/False)
b) The find() method returns the index of the first occurrence of a substring within a string. (True/False)
c) A string can be divided into a list of substrings using the join() method. (True/False)
d) If a string ends with a particular suffix, it is checked using the startswith() method. (True/False)
e) The split() method returns a list of substrings based on a specified delimiter. (True/False)

4.4 REGULAR EXPRESSIONS

A regular expression is a collection of characters with a very specific syntax that we may use to match or locate other characters or groups of characters. The terms *regex* and *regexp* are frequently used to describe regular expressions. Text manipulation, validation, and search are common uses for them. The built-in *re* module in Python makes it easy to work with regular expressions. The details about how to work with modules will be covered in later chapter. The general syntax used to import the *re* module is as mentioned:

import re

The *re* module in Python gives full support for regular expressions of Pearl style. When a regular expression is implemented or used incorrectly, the *re* module raises the re.error exception.

Regular Expression Functions

The *re* module offers a set of functions that allows us to search a string for a match. Table 4.2 gives a list of RegEx functions along with its description.

Table 4.2: RegEx Functions

Function	Description
findall()	Returns a list containing all matches.
search()	Returns a match object if there is a match anywhere in the string.
split()	Returns a list where the string has been split at each match.
sub()	Replaces one or many matches with a string.

Let us now write a program that uses all the methods in Table 4.2. Example 4.8 shows how to use these methods.

Example 4.8: Using RegEx methods

```
# Program to use methods of RegEx

import re
string1 = "the demo of python program"

print("findall(): ",re.findall("th", string1))
print("search(): ",re.search("python", string1))
print("split(): ",re.split(" ", string1))
print("sub(): ",re.sub(" ", "*", string1))
```

OUTPUT:

```
findall():  ['th', 'th']
search():  <re.Match object; span=(12, 18), match='python'>
split():  ['the', 'demo', 'of', 'python', 'program']
sub():  the*demo*of*python*program
```

The example demonstrates the use of common regex methods in Python. The *findall()* method searches for all occurrences of the substring "th" within string1, it returns ['th', 'th'] as the characters appear twice. The *search()* method locates the first occurrence of the substring "python" in string1, returning a match object indicating its position (span=(12, 18)) and value ('python'). The *split()* method splits the string at every blank space (" "), producing a list of words: ['the', 'demo', 'of', 'python', 'program']. Finally, the *sub()* method replaces all blank spaces with an asterisk ("*"), resulting in the string "the*demo*of*python*program".

Special Characters

As the name suggests, there are some characters with special meanings, also known as Metacharacters. To understand the RE analogy, Metacharacters are useful and important. They will be used in functions of module re. Table 4.3 gives a list of special characters along with its description.

Table 4.3: Special Characters

Characters	Description
.	**Dot** - It matches any characters except the newline character.
^	**Caret** - It is used to match the pattern from the start of the string. (Starts With)
$	**Dollar** - It matches the end of the string before the new line character. (Ends with)
*****	**Asterisk** - It matches zero or more occurrences of a pattern.
+	**Plus** - It is used when we want a pattern to match at least one.
?	**Question mark** - It matches zero or one occurrence of a pattern.
{}	**Curly Braces** - It matches the exactly specified number of occurrences of a pattern
[]	**Bracket** - It defines the set of characters
|	**Pipe** - It matches any of two defined patterns.

The example of how to use RegEx special characters specified in table 4.3 is given in the Python program of Example 4.9.

Example 4.9: Use of special characters

```
# Program to use special characters with regular expression function
import re
string1 = "hello world"
# .Dot - Any character (except newline character)
word = re.findall("he..o", string1)
print(word)
#Check if the string starts with 'hello':
word = re.findall("^hello", string1)
if word:
  print("Yes, the string starts with 'hello'")
else:
  print("No match")
#Check if the string ends with 'world':
word = re.findall("world$", string1)
if word:
  print("Yes, the string ends with 'world'")
else:
  print("No match")
#Search for a sequence that starts with "he", followed by 0 or more any characters, and an "o":
word = re.findall("he.*o", string1)
print(word)
#Search for a sequence that starts with "he", followed by 1 or more  (any) characters, and an "o":
word = re.findall("he.+o", string1)
print(word)
#Search for a sequence that starts with "he", followed by 0 or 1 character, and an "o":
word = re.findall("he.?o", string1)
print(word)
```

```
#Search for a sequence that starts with "he", followed by exactly any 2 characters, and an "o":
word = re.findall("he.{2}o", string1)
print(word)
#Find all lowercase characters alphabetically between "e" and "r":
word = re.findall("[e-r]", string1)
print(word)
#Check if the string contains either "hello" or "hi":
word = re.findall("hello|hi", string1)
print(word)
if word:
  print("Yes, there is at least one match!")
else:
  print("No match")
```

OUTPUT:

```
['hello']
Yes, the string starts with 'hello'
Yes, the string ends with 'world'
['hello wo']
['hello wo']
[]
['hello']
['h', 'e', 'l', 'l', 'o', 'o', 'r', 'l']
['hello']
Yes, there is at least one match!
```

This example demonstrates various regex patterns and their applications using the re.findall() function in Python. The pattern *he..o* matches any sequence of characters starting with ‘he’, followed by any two characters, and ending with character ‘o’, it returns ['hello'] as output. The *^hello* pattern checks if the string starts with the character sequence “hello”, as the string starts with hello we get “Yes, the string starts with 'hello'” as output. The pattern *world$* confirms that

the string ends with the character sequence “world”, resulting in the output “Yes, the string ends with 'world'”. The pattern *he.*o* matches sequences starting with characters “he”, followed by zero or more occurrences of characters, and ending with character “o”, the output we get here is ['hello wo']. The pattern *he.+o* does the same match but it requires at least one character between characters “he” and “o”. The pattern *he.?o* matches character sequence “he”, optionally followed by one character, and ending with character “o”, this match returns an empty list ([]). The pattern *he.{2}o* matches the character sequence “he”, followed by exactly two characters, and character “o”; it returns ['hello'] as output. The character range [e-r] finds all lowercase letters between characters “e” and “r”, it returns a list ['h', 'e', 'l', 'l', 'o', 'o', 'r', 'l']. Finally, the pattern *hello*|*hi* checks for either the string “hello” or string “hi”, in our example we find a confirmed match as ['hello'].

Check Your Progress-3

a) The re module in Python is used for regular expressions. (True/False)
b) The re.findall() method modifies the original string. (True/False)
c) The re.split() function returns a list after splitting a text at each instance of the given pattern. (True/False)
d) Regular expressions use metacharacters such as. (dot), * (asterisk), and + (plus) to define flexible matching rules. (True/False)
e) The caret (^) matches the start of a string, while the dollar sign ($) matches the end of a string. (True/False)

4.5 LET US SUM UP

In this unit, we explored the versatility of strings in Python, which consist of a series of characters. We explored the string manipulation features, such as indexing, slicing, and formatting, which make working with strings quick and easy. Built-in functions like lower(), replace(), split(), and join() further improve string operations by facilitating smooth changes and transformations. We also looked at regular expressions, which offer powerful pattern matching capabilities for complicated string operations.

4.6 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

1-a True
1-b False
1-c True
1-d False
1-e True
2-a True
2-b True
2-c False
2-d False
2-e True
3-a True
3-b False
3-c True
3-d True
3-e True

4.7 ASSIGNMENTS

1. What is a string in Python?
2. Write the output of the following code:

```
text = "Hello Python"
print(text[6:])
```

3. What does the strip() method do?
4. How would you search for a pattern in a string?
5. Write a Python program to count the number of vowels in a string using a loop.
6. Create a program that accepts a string and removes all spaces using string methods.
7. Write a regex pattern to validate an email address.
8. Develop a program to find and replace a word in a string without using the replace method.
9. Given a string of comma-separated numbers, write a program to calculate the sum of all numbers.

Block-4

OOP Concepts, Exception, File Handling and GUI

Unit-1: Introduction to Object Oriented Programming

1

Unit Structure

1.0 LEARNING OBJECTIVE

After studying this unit student should be able to:

- Understand the key concepts of Object-Oriented Programming (OOP).
- Differentiate between classes and objects.
- Use attributes and methods in classes to create programs.
- Apply the self keyword to manage object-specific behavior.
- Recognize the benefits of using OOP in software development.

1.1 INTRODUCTION

Till now, we are using a procedural method to write Python programs, which splits the program into methods and executes a set of codes either in sequence or in response to pre-set conditions. This methodology of Python programming works well for smaller programs, but as programs grow in size and complexity, it becomes harder to keep everything organized.

Think about instances in day-to-day life: we usually come across an item (or objects) with related data and some related processes. Let's use an automobile as an illustration. A car's colour, model, and speed are examples of its qualities. But an automobile can do more than just sit; it can accelerate, start, and stop. Attempting to represent all of these characteristics and functions in terms of processes is extremely difficult.

In order to solve the stated problem, Object-oriented programming has been introduced by Python. It is the method of organizing code into objects, through which we can represent real-world entities in our programs.

An object is like a real-world thing you can interact with, and it’s created based on a class. A class acts like a blueprint, defining what properties (also called attributes) the object will have and what actions (known as functions) it can perform. By using an Object-oriented approach, you can write code that’s cleaner, more organized, and easier to reuse, especially when programs start to grow.

1.2 WHY OBJECT-ORIENTED PROGRAMMING? (OOP)

We have already seen that the OOP approach (classes and objects) allow us to organize our code well, but you may be wondering by now - **why should we use Object-Oriented Programming at all?** Why not stick to the traditional procedural approach?

The answer lies in how we deal with **complexity**. As programs get bigger, keeping the code manageable becomes tougher. One loses perspective on what each part does and how they interact. OOP solves these by simply allowing us to code in a more human-alike way.

Just envision how you interact with things around you. When you use your phone, you are not interested in how its inner circuitry works. You just care about its exciting features like making calls, using social media apps, taking pictures or listening to music/podcasts. This is precisely what OOP performs; it allows us to focus on what an object does and worries less about the internal complexity.

Here are some good reasons why OOP is useful:

- **Better Organization**

  In OOP, related data and actions can be organized into classes. Rather than scattering variables and functions all over the code, everything pertaining to an object-a Car or Phone-is grouped into one unit. Thus, the program cuts down on the confusion and becomes easier to develop.

- **Reusability**

  One of the most important advantages of OOP is that we can reuse the existing code. For example, if we have already created a class to represent a car, we don't have to write that code again when adding another car to your program. We can simply instantiate a new object from class Car.

- **Scalability**

  OOP helps to take programs to a new dimension of scaling with easy adaptability. For example, new classes can be created based on an existing one using inheritance. Hence, the programmer has the opportunity to simply

add new features to the existing code without modifying or breaking old code.

- **Cleaner Code and Maintenance**

  In OOP, code becomes more modular and cleaner; the maintenance is also easier. When a problem occurs, we can simply identify the particular class or object giving rise to the error without bothering about the rest of the program.

- **Closer to Real Life**

  OOP mimics how we view the world. We deal with objects everyday-cars, telephones, people, or books, -all having properties and behaviors. OOP provides a natural way of coding these concepts, rendering it easier to write and understand programs.

In conclusion, Object-Oriented Programming gives us the opportunity to write clean and organized programs while ensuring scalability. It enables users to break down real-world problems into smaller, more manageable problems that can be solved in an efficient manner.

1.3 REAL WORLD ANALOGY

Imagine you are organizing a library. One library might have hundreds, sometimes thousands, of books. Each book contains specific details like its title, author, genre, and the number of pages. These are the details that tell you what the book is like and help locate it, so you may be able to borrow it and take good care of it.

Now, what can you do with a book? You can read it, stick in bookmarks, lend it to a friend, or take it back to the library. Those are the few possible things a book could be acting upon. You don't need to know how a book is printed or bound before you can use one. You just know how to interact with a book.

Managing a library becomes possible and efficient when one line of designation is taken for handling books. Instead of each book building up its system, each stanza applies equally to every other book: that is, their issuing, receiving back,

or finding stanzas work on every item in the library at all times. This ensures sufficient management of the library as the size of books grows.

Just like with programming, OOP allows us to structure management of complexity in programming. Use it as a recommended and systematized style of treating pieces of a program-the bits that perform some work and those that hold some bit of information, without having to mind about how everything else works. In other words, you can manage growth in complexity in programming with the same ease as you organize a library so that you aren't overwhelmed by the fresh growth in its collection.

1.4 UNDERSTANDING CLASSES AND OBJECTS

A Class is a blueprint for creating objects, while an object is the instance of that class. It defines the structures and behaviours that the objects will have. A class may be associated with a recipe, while the objects are the actual dishes prepared as per the recipe. Example 4.1 shows the creation of classes in python.

Example 4.1: Creating Class and Object

```
# Program to create a class and object.
# Creating a class
class Car:
    pass  # Placeholder, meaning the class is currently empty

# Creating objects
carObject1 = Car()
carObject2 = Car()

print(car1)
print(car2)
```

OUTPUT

```
<__main__.Car object at 0x...>
<__main__.Car object at 0x...>
```

So as shown in the above program, Car is the class, while car1 and car2 are two separate objects created from the class Car.

Note:

- **A class defines the structure and behavior of its objects.**
- **Objects are specific instances of a class, each having its own identity.**

1.5 ATTRIBUTES AND METHODS

In OOP an attribute and a method are basic concepts of a class. These concepts define the structure (data) and behaviour (functions) of objects. Let's dive deeper to understand attributes and methods with practical examples and applications.

What are Attributes?

Attributes are the features or properties of an object. They are similar to variables of type data that yield some specific data related to an object.

For example, for a class Car:

Attributes can be name, company, and colour.

In Python, attributes are defined inside a class and accessed with the dot (.) operator.

Types of Attributes:

1. Instance attributes: Specific to an object; each object maintains its own unique set of instance attributes.
2. Class attributes: shared by all objects of the class.

Example 4.2: Attributes in Class

```
# Program to demonstrate usage of attributes in a class.
class Car:
    # Class attribute
    noOfWheels = 4
```

```
    def __init__(self, name, company, colour):
        # Instance attributes
        self.name = name
        self.company = company
        self.colour = colour

# Creating objects
carObject1 = Car("Swift", "Maruti", "White")
carObject2 = Car("i10", "Hyundai", "Black")

# Accessing attributes of first object
print(carObject1.name)
print(carObject1.company)
print(carObject1.colour)
print(carObject1. noOfWheels)

# Accessing attributes of second object
print(carObject2.name)
print(carObject2.company)
print(carObject2.colour)
print(carObject2. noOfWheels)
```

OUTPUT OF PRINTING THE ATTRIBUTES OF FIRST OBJECT:

Swift

Maruti

White

4

OUTPUT OF PRINTING THE ATTRIBUTES OF SECOND OBJECT:

i10

Hyundai

Black

4

The above program in example 4.2 demonstrates the usage of instance attribute and class attribute in a class. The Car class is declared with attributes to signify the various details about a Car. It contains both instance and class attributes.

1. Instance Attributes (name, company, colour):
   These are defined in the __init__ method and are unique to each object. Each individual car will thus have its own values for the name, company, and colour.
2. Class Attributes (noOfWheels):
   The attribute is defined outside of any other methods and shared across all objects of the Car class. All the car objects will have four wheels as their default value.

Two objects, carObject1 and carObject2, have been instantiated with differing instance attribute values. Using the dot (.) operator, the attributes of each object are accessed.

What are Methods?

Methods are the actions or behaviours that an object of a class can perform. They are somewhat similar to functions but are defined in a class and operate on an object's attributes. With methods, we typically would either interact with the object's data or perform specific actions with the object's attributes.

For example, in a class Car, methods can be start(), stop() or accelerate() .

Types of Methods

1. Instance Methods: These are the most commonly used methods.
   They can read and modify instance attributes and perform actions specific to their objects. Instance methods must have as their first parameter the object that calls the method.
2. Class Methods: These methods must work on class attributes, shared across all objects of that class. They are declared on a class either via the classmethod decorator and must admit cls as their first parameter (referring to the class itself).

3. Static Methods: Static methods themselves do not directly operate on instance attributes nor class methods. Instead, they are used at times when no data neither class nor object will need to be made available. They are declared using the staticmethod decorator.

A program in example 4.3 demonstrates how different types of methods can be used in a class:

Example 4.3: Methods in Class

```
# Program to demonstrate usage of methods in a class.
class Car:
    # Class attribute
    noOfWheels = 4

    def __init__(self, name, company, colour):
        # Instance attributes
        self.name = name
        self.company = company
        self.colour = colour

    # Instance method to display details of the car
    def displayCar(self):
        print(f"Car Name: {self.name}")
        print(f"Company: {self.company}")
        print(f"Colour: {self.colour}")

    # Instance method to change the colour of the car
    def changeCarColour (self, newClr):
        self.colour = newClr
        print(f"The color of {self.name} has been changed to {newClr}.")

    # Class method to modify class attribute
    @classmethod
    def updateNoOfWheels(cls, newNumOfWheels):
        cls.noOfWheels = newNumOfWheels
        print(f"Number of wheels has been changed to {newNumOfWheels}.")

    # Static method to calculate mileage of a car
    @staticmethod
    def getMileage(distance, fuel):
        return distance / fuel
# Creating an object of the class Car
carObject1 = Car("Swift", "Maruti", "White")
```

```
# Accessing instance methods of the class Car
carObject1.displayCar()
carObject1.changeCarColour("Blue")
carObject1. displayCar()

# Accessing class method of the class Car
print(f"Original number of wheels: {Car.noOfWheels}")
Car.updateNoOfWheels(6)
print(f"Updated number of wheels: {Car.noOfWheels}")

# Accessing static method
mileageCar = Car.getMileage(300, 20)
print(f"Mileage of the car: {mileageCar} km/l")
```

OUTPUT OF INSTANCE METHOD:

```
Car Name: Swift
Company: Maruti
Colour: White
The color of Swift has been changed to Blue.
Car Name: Swift
Company: Maruti
Colour: Blue
```

OUTPUT OF CLASS METHOD:

```
Original number of wheels: 4
Number of wheels has been changed to 6.
Updated number of wheels: 6
```

OUTPUT OF STATIC METHOD:

```
Mileage of the car: 15.0 km/l
```

The above program in example 4.3 demonstrates the usage of various types of methods in a class. The Car class is declared with attributes and three types (instance, class and static) of methods.

1. Instance Methods:
    - displayCar(): Prints details of a specific car object (name, company, and colour). This method uses self to access instance attributes.
    - changeCarColour(newClr): This shows instance methods that can modify attributes of an object. It changes the colour attribute of a specific car object to a new value (newClr).

2. Class Method:
    - updateNoOfWheels(cls, newNumOfWheels): It modifies the class attribute noOfWheels, shared by all objects of the class. Uses the cls keyword to access and update the class attribute.

3. Static Method:
    - getMileage(distance, fuel): A utility method that accepts distance travelled and fuel consumed and calculates and returns mileage. There are no dependencies on either instance or class attributes.

Table 1.1 gives the comparison of different methods used in the program.

Table 1.1: Comparison of Methods

Method Type	Decorator	First Parameter	Objective
Instance Method	None	self	Operates on instance attributes and specific objects.
Class Method	@classmethod	cls	Operates on class attributes and affects all objects of the class.
Static Method	@staticmethod	None	General-purpose methods; does not use class or instance data.

Note:

- **In a class, attributes describe "what an object has," while methods describe "what an object does."**
- **Attributes and methods, taken together, encapsulate the data and behaviour of objects and support the principles of Object-Oriented Programming in Python.**
- **Attributes and methods are accessed using the dot (.) operator.**

Check your progress - 1

a) An object is a template or class instance that contains data and methods. (True/False)

b) Attributes capture the activities of an object. (True/False)

c) Methods are functions defined by a class to act or apply to the object attributes. (True/False)

d) A class is a blueprint for creating objects. (True/False)

1.6 THE self KEYWORD

In Python, the self keyword is a critical part in defining and working with attributes and instance methods in a class. It allows access to the attributes and methods of the object that calls the method of a class.

What is self?

- Represents the Instance: The self keyword is a reference to the current instance of the class. It is used to access the attributes and methods associated with that specific object.
- Mandatory in Instance Methods: When defining an instance method, the first parameter must always be self. This lets Python know that the method is tied to the object.
- Not a Reserved Keyword: While self is a convention, you can technically use any valid variable name as the first parameter in instance methods. However, sticking to self ensures readability and consistency.

Why do we need self?

- Separation of instance variables from local variables: Without self, the method cannot differentiate between attributes of an instance and local variables.
- Binding data to a particular object: Each object has its own set of attributes, and through self, the method modifies and fetches attributes related to the object.
- Allows for clearer readability: The use of self makes it clear that the attributes and methods belong to the object.

Let’s understand the usage of 'self' with the multiple examples presented in the below section:

Example 4.4: Basic usage of 'self' keyword

```
# Program to demonstrate the usage of self.
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greetings(self):
        print(f"Hello, my name is {self.name} and I am {self.age} years old.")

# Creating objects
personObject1 = Person("Mohan", 15)
personObject2 = Person("Rohan", 20)

# Accessing methods
personObject1.greetings()
personObject2.greetings()
```

OUTPUT

```
Hello, my name is Mohan and I am 15 years old.
Hello, my name is Rohan and I am 20 years old.
```

In the above example, self.name and self.age are attributes of the specific instance of the object that called the method (*personObject1* or *personObject2*).

Example 4.5: Changing the attributes using self

```
# Program to demonstrate the usage of self.
class Account:
    def __init__(self, accHolder, balance):
        self. accHolder = accHolder
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
        print(f"{amount} deposited. Updated balance is {self.balance}.")
```

```
    def withdraw(self, amount):
        if amount > self.balance:
            print(“Can’t withdraw, Not enough balance!")
        else:
            self.balance -= amount
            print(f"{amount} withdrawn. Updated balance is {self.balance}.")

# Creating an object
account = Account("Mohan", 500)

# Performing operations
account.deposit(500)
account.withdraw(900)
account.withdraw(500)
```

OUTPUT

500 deposited. Updated balance is 1000.
900 withdrawn. Updated balance is 100.
Can’t withdraw, Not enough balance!

In the above example, the use of self.balance guarantees that any changes made to the balance are associated with the specific account object.

Example 4.6: Using self with multiple objects

```
# Program to demonstrate the usage of self.
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def calculateArea(self):
        return self.length * self.width
# Creating objects two rectangles
rectObject1 = Rectangle(3, 7)
rectObject2 = Rectangle(9, 15)

print(f"Area of first rectangle is: { rectObject1.calculateArea()}")
print(f"Area of second rectangle is: { rectObject2.calculateArea()}")
```

OUTPUT

Area of first rectangle is: 21

Area of first rectangle is: 135

In the above example, self.length and self.width make sure that every rectangle object defines its own length/width independent of other rectangle objects.

Common mistakes when using self:

1. While defining methods: This is caused by omitting self in method definitions and leads to a TypeError.

Example 4.7: Omitting self: Use case- 1

```
# Program to demonstrate the usage of self.
class Welcome:
    def greetings():
        print("Hello!")
obj = Welcome()
obj. greetings()
```

OUTPUT

```
# TypeError: greetings() takes 0 positional arguments but 1 was given
```

```
#correct version of above mentioned code
#include self in method definition
class Welcome:
    def greetings(self):
        print("Hello!")
obj = Welcome()
obj. greetings()
```

OUTPUT

```
Hello!
```

2. While accessing attributes: As shown in the example 4.8, without the self keyword, python treats the variable as a local variable.

Example 4.8: Omitting self: Use case- 2

```
# Program to demonstrate the usage of self.
class MathmaticalOp:
    def __init__(self, number):
        number = number  # This does not assign to the instance attribute number

    def showNumber(self):
        print(self.value)

obj= MathmaticalOp(10)
obj.showNumber()
```

OUTPUT

```
# AttributeError: MathmaticalOp object has no attribute number
```

```
#correct version of above mentioned code
#include self in while assigning the instance attribute
class MathmaticalOp:
    def __init__(self, number):
        self.number = number

    def showNumber(self):
        print(self.number)

obj= MathmaticalOp(10)
obj.showNumber()
```

OUTPUT

```
Hello!
```

Note:

- **The "self" keyword allows us to bind the instance attributes and methods to the particular object that is calling them.**
- **The "self" keyword enhances the clarity of the code, showing clearly which attributes and methods belong to an object.**
- **Using the "self" keyword properly forms the backbone of writing robust and maintainable object-oriented programs.**

1.7 CONSTRUCTOR METHODS

In Object-Oriented Programming, constructors are special methods used for the initialization of various attributes of an object when an object is created. In general, the constructor can be viewed as a special method that "constructs" or "sets up" any object and prepares it for use.

In Python, constructors are introduced by the method called __init__, which is automatic whenever a new object from the class is instantiated.

Why Do We Need Constructors?

In most cases, an object has to have some attributes or values assigned to it right from the start. Such a process of performing the action manually will consume lots of effort. Assigning values through a constructor will help in avoiding this redundant work by enabling the object to have values defined and assigned upon creation. This shortens the coding, sets consistency, and yields cleaner code.

Major Features of Constructors in Python:

- Special Method: The constructor is defined through the __init__ method which is a reserved method in Python.
- Self-invocation: When an object is created, the __init__ method is automatically invoked.
- Initializing the Object: The constructor is used to set initial values into the attributes of any object.
- Custom Values: We can define some parameters in the constructor to accept values from the user and initialize its attributes with it.

Syntax to Create a Constructor:

Example 4.9 shows the simple syntax of a constructor in Python.

Example 4.9: Constructor Syntax

```
# Program to demonstrate the creation of Constructor.

class ClassName:
    def __init__(self, parameters):
        # Constructor code
```

- self: The first parameter of the constructor, like all methods in Python, is self. This references the instance of the class being created.
- parameters: These are optional. If provided, they permit the user to pass values to initialize the attributes of the object.

Constructor in a Class

Now, Let's look at a simple code in example 4.10 to understand how a constructor works:

Example 4.10: Constructor in a Class

```
# Program to demonstrate the Constructor in a class.
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

# Creating objects of the class Person
personObject1 = Person("Mohan", 20)
personObject2 = Person("Rohan", 15)

# Accessing attributes
print(f"First Person: { personObject1.name}, Age: { personObject1.age}")
print(f"Second Person: { personObject2.name}, Age: { personObject2.age}")
```

OUTPUT

```
First Person: Mohan, Age: 20
Second Person: Rohan, Age: 15
```

In above code:

- The __init__ method(constructor) accepts three parameters, namely, self, name, and age.
- The moment this statement is executed, **personObject1 = Person("Mohan", 20)**, the __init__method gets automatically invoked.
- Similarly, ***personObject2 = Person("Rohan", 15)*** will be executed, it creates another object with its own name and age.

Default Constructor

As shown in the example 4.11, In the case of no parameters passed during object creation, we may have to define a constructor with no arguments. This is known as the default constructor.

Example 4.11: Default Constructor

```
# Program to demonstrate the default constructor in a class.

class Animal:
    def __init__(self):
        self.species = "Not Known!"

# Creating an object
animalObject = Animal()

# Accessing the attribute
print(f"Species of an animal: {animalObject.species}")
```

OUTPUT

Species of an animal: Not known!

Constructor with Default Values

As shown in the example 4.12, We can furthermore assign default values to constructor parameters that let us create the objects with user-defined values.

Example 4.11: Constructor with default values

```
# Program to demonstrate the default values in a constructor.
class Car:
    def __init__(self, company="NotKnown", colour="White"):
        self.company = company
        self.colour = colour

# Creating objects
carObject1 = Car("Swift", "Black")
carObject2 = Car()  # Uses default values

# Accessing attributes
print(f"First Car Object: Company={car1.brand}, Colour={car1.colour}")
print(f"Second Car Object: Company={car2.brand}, Colour={car2.colour}")
```

OUTPUT

First Car Object: Company=Swift, Colour=Black

Second Car Object: Company =NotKnown, Colour=White

Note:

- **Use of constructors plays an important role in the development of scalable and maintainable object-oriented systems.**
- **Constructors automatically configure attributes at object creation time and eliminate redundant codes.**
- **Allow passing parameters to adjust object attributes while allowing for default values.**

Check your progress - 2

a) The self keyword applies to the class itself, not to an instance of it. (True/False)

b) A constructor method is called when an object is created and used for initializing the object attributes. (True/False)

c) An object can only have one constructor method. (True/False)

d) All methods within classes are mandatory to include the self keyword. (True/False)

e) The self keyword must be present in static methods. (True/False)

1.8 LET US SUM UP

In this unit, the fundamentals of object-oriented programming were introduced. You gained insights into classes and objects in a way that classes can be seen as blueprints for creating objects. Attributes and methods assist in defining the properties and behaviour of the objects. You used the self keyword when referring to the instance of the object itself. The unit also covers constructor methods, designed to initialize objects with specific values when they are created.

1.9 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

1-a False
1-b False
1-c True
1-d True
2-a False
2-b True
2-c False
2-d False
2-e False

1.10 ASSIGNMENTS

1. Write the basic concepts of classes and objects with their advantages and disadvantages.
2. Explain the difference between attributes and methods.
3. Explain the self keyword in a class.
4. What is the constructor method and what is its most important role in object creation?

5. Write a Python program implementing the below functionalities using classes and methods:
    - Create a class Number to check whether a given number is positive or negative.
    - Create a class InterestCalculator, which computes the value of I = (P*R*N)/100 if the values of P, R, and N are greater than 0.
    - Create a class Comparison to read two numbers num1 and num2; then, determine whether num1 is less than, greater than or equal to num2.
    - Create a class LargestNumber that takes three numbers num1, num2, and num3 to find the greatest of three numbers.
    - Create a class NameChecker to enter two names, name1 and name2, and check whether they are similar or not.

Unit-2: Inheritance and Polymorphism

2

Unit Structure

2.0 LEARNING OBJECTIVE

After studying this unit student should be able to:

- Understand the principles of inheritance and polymorphism in Python
- Differentiate between types of inheritance
- Implement method overriding and understand its applications
- Describe encapsulation and data hiding, and implement it effectively
- Recognize the importance of abstract classes and interfaces in object-oriented programming

2.1 INTRODUCTION

In the earlier chapter, we learnt the general concepts of OOP in Python, namely classes, objects, attributes, and methods. We investigated how these concepts allow one to organize code into logical structures, making the code cleaner, modular, and reusable. This mimics the real world, where objects have specific properties and behaviors; for example, a car would have attributes such as color and speed and methods such as acceleration and braking.

While the earlier chapter introduced a necessary foundation for understanding the definition and operation of individual objects, it fell short of defining how objects relate to one another. In practical applications, relations between objects are very crucial, such as:

- A car is a type of vehicle but possesses unique features (Inheritance).
- Another vehicle may vary in the way it behaves, and thus the vehicle may start or stop by virtue of the configuration of such a vehicle, be it a car or bicycle (Polymorphism).

In this chapter, we will look into the relationships as modeled via Inheritance and Polymorphism-Legacies which are absolutely vital pillars of OOP. Inheritance is a mechanism that allows one class to inherit the attributes and methods of another class, thereby permitting an overlap and reuse of codes. Polymorphism lets the method of a class behave differently based on which object has called it, thereby adding flexibility and extendibility to the programs.

On completing this chapter, you will understand how these concepts can be harnessed in the design of scalable and efficient systems along with an application of these concepts in Python.

2.2 INHERITANCE AND POLYMORPHISM

Inheritance is one of the key principles of Object-Oriented Programming (OOP). According to its definition, a new class, the child class, can inherit attributes and methods from an existing class, the parent class. This makes it possible for a developer to reuse code while building a hierarchical relationship among the classes, mirroring reality.

For illustration, in a system which manages vehicles:

So, we have:

- **Parent Class:** Vehicle class with attributes such as speed, company and method definitions such as methods start() and stop().
- **Child Class:** Car and Bicycle, which inherit these properties and behaviors and may also define their own attributes and methods like numberOfDoors, an attribute for cars or pedal(), a method for bicycles.

Why Use Inheritance?

- **Code Reusability:** Common attributes and methods would be defined in a parent class for child classes to inherit.
- **Logical Hierarchy:** Develops relationships across many entities, which simplifies understanding of the system and maintains it in the long run.
- **Extensibility:** New features can always be added to child classes with absolutely no effect on the parent class.
- **Polymorphism:** It provides a foundation for overriding child class methods. Thus, different official behavior for each subclass can be provided.

Key features of Inheritance

1. **Inheritance Hierarchy:** The inheriting class acquires the attributes and methods of the base class by extending without modification. For instance:
   - A base class Employee will have generic attributes such as name and salary.
   - A derived class Manager will include department or any other attribute that is specific for it.
2. **super() Function:** When the super() function is used in a derived class, it gives access to the method of the base class in order to modify it in the derived class context. This helps to resolve conflicts when two methods share the same name.
3. **Overriding methods:** Child classes can redefine methods of the parent class to provide specific behavior. This is central to the concept of polymorphism, which is discussed in more detail later in this chapter.

Example 2.1: Program to demonstrate the concept of Inheritance

```
# Program to demonstrate the concept of Inheritance
# Parent class
class Vehicle:
    def __init__(self, company, model):
        self.company = company
        self.model = model
    def printData(self):
        return f"Vehicle company: {self.company}, Model: {self.model}"

# Child class
class Car(Vehicle):
    def __init__(self, company, model, noOfDoors):
        # Use super() to call the parent class's __init__ method
        super().__init__(company, model)
        self.noOfDoors = noOfDoors

    def printData(self):
        # Use super() to include details from the parent class
        parentClassData = super().printData()
        return f"{parentClassData}, Doors: {self.noOfDoors}"
```

```
vehicleObj = Vehicle("Generic Brand", "X")
carObj = Car("Swift", "ZXI", 4)

print(vehicleObj.printData())
print(carObj.printData())
```

OUTPUT:

Vehicle company: Generic Brand, Model: X
Vehicle company: Swift, Model: ZXI, Doors: 4

The above program shows the inheritance of the Car class from the Vehicle class, inheriting attributes and methods, while also adding its own. The parent class, Vehicle, consists of two attributes, company and model, and a method printData which prints these details. Car extends this base functionality by introducing an additional attribute, noOfDoors, which represents the number of doors relative to cars. Using the super() function, the Car class invokes the parent class's __init__ method to initialize the inherited attributes, company and model, to avoid redundancy. The printData method is also overridden in Car to include the parent class's details (retrieved using super().printData()) along with the number of doors, demonstrating how inheritance allows code reuse and extension.

Types of Inheritance

1. **Single Inheritance:** It occurs when a child class derives from just one parent class. Arguably, this is the most basic form of inheritance, whereby a child class has the access to all attributes and methods of the parent class. A parent-child relationship is direct where one class is child to one single class, thus easing code reuse and extensibility. which is discussed in more detail later in this chapter. The Python program in Example 2.2 demonstrates the concept of single inheritance.

Example 2.2: Concept of single inheritance

```
# Program to demonstrate the concept of single Inheritance
# Parent class
class Parent:
    def parentMethod(self):
        print("This is parent class")
```

```
# Child class
class Child(Parent):
    def childMethod(self):
        print("This is child class")

childObj = Child()
childObj.parentMethod()
childObj.childMethod()
```

OUTPUT

```
This is parent class
This is child class
```

The above python code is the demonstration of single inheritance in Python, where the child class is derived from the parent class. The child class introduces a new method called childMethod, which prints a message different from that of the parent class. By virtue of the inheritance from Parent, the child class automatically acquires the ability to call the method parentMethod, which is shown when the childObj invokes the parentMethod by the statement childObj.parentMethod(). The child class can, in addition, call its own method, childMethod, illustrated by childObj.childMethod(). The resulting output displays the messages coming from both parents and children, thus showing that a child class accesses both its methods and those inherited from the parent class.

2. **Multiple Inheritance:** It occurs when a child class gets its properties from two or more parent classes. Thus, the child class garners strength from the combination and extension of the functionalities of several parent classes. However, it will lead to a problem if its parent classes communicate contradictory information through methods or attributes. The resolution of the conflict is ordered through Method Resolution Order (MRO), which brings an element of order in providing the methods and attributes in a predictable chain.

Example 2.3: Concept of multiple inheritance

```
# Program to demonstrate the concept of multiple Inheritance
# Parent class 1
class Father:
    def fatherMethod(self):
        print("This is a father class.")

# Parent class 2
class Mother:
    def motherMethod(self):
        print("This is a mother class.")

# Child class
class Child(Father, Mother):
    def childMethod (self):
        print("This is a child class.")

childObj = Child()
childObj.fatherMethod()
childObj.motherMethod()
childObj.childMethod()
```

OUTPUT

```
This is a father class.
This is a mother class.
This is a child class.
```

The above python code demonstrates multiple inheritance in Python, with the child class inheriting from two parent classes: Father and Mother. The father class has a method fatherMethod that prints a message and the mother class has a method motherMethod that prints something different from that of the father. Inheriting both of these methods, the child class also defines its own method, childMethod, which prints its own unique message. When an object of the child class

(childObj) is created, it can have access to methods inherited from both parent classes (fatherMethod and motherMethod) including its own childMethod. The output confirms that the child class is able to access methods from both parent classes; such capabilities of accessing via multiple inheritance allow the child class to inherit behaviors from more than one parent.

3. **Hierarchical Inheritance:** It occurs when multiple child classes have a single parent class. The modeling is based on reuse of the code since all child classes will share the same behavior and attributes defined in the parent class. It would be helpful in creating a centralized parent class in one area where other extended child classes will branch off.

Example 2.4: Concept of hierarchical inheritance

```
# Program to demonstrate the concept of hierarchical Inheritance
# Parent class 1
class Animal:
    def animalMethod(self):
        print("This is animal method.")

# Child class 1
class Dog(Animal):
    def dogMethod(self):
        print("This is a dog method.")

# Child class 2
class Cat(Animal):
    def catMethod(self):
        print("This is a cat method.")

dogObject = Dog()
catObject= Cat()

dogObject.dogMethod()
catObject.catMethod()
```

OUTPUT

```
This is a dog method.
This is a cat method.
```

In this program, hierarchical inheritance is illustrated in Python, where more than one child class inherits from a single parent class. The Animal class acts as a parent which has a definition of its method, animalMethod. This behavior represents things that are common to all animals. The Dog and Cat classes inherit from Animal; hence, they are referred to as child classes of Animal class. In turn, each child class enacts one from its own unique methods, for the Dog class the dogMethod and for the Cat class the catMethod. Instances of Dog and Cat that are created namely; dogObject and catObject call the methods common for both classes, showing how, in particular, the child class can both have a different implementation of a method and inherit the common behavior from its parent class, Animal. The output shows working methods from both the child classes independently.

4. **Multilevel Inheritance:** It occurs when a child class inherits from another parent class whose own parent class derives from another. That means multiple inheritance exists in a line: the attributes and methods will propagate through many levels. They can help create a more sophisticated hierarchy, but imploding inheritance also requires limited manageability if the whole structure is too deep.

Example 2.5: Concept of multilevel inheritance

```
# Program to demonstrate the concept of multilevel Inheritance
#Grand Parent class
class Grandparent:
    def grandparentMethod (self):
        print("This is a Grandparent class.”)

#Parent class
class Parent(Grandparent):
    def parentMethod(self):
        print("This is a parent class.")

#Child class
class Child(Parent):
    def childMethod(self):
        print("This is a child class.")
```

childObject = Child() *childObject.grandparentMethod()* *childObject.parentMethod()* *childObject.childMethod()*
OUTPUT *This is a Grandparent class.* *This is a parent class.* *This is a child class.*

This program illustrates multilevel inheritance in Python, wherein a class inherits from one class, itself a subclass of another class, forming a chain or hierarchy of inheritance. The Grandparent class is to the top most class with a definition of a method named the grandparentMethod. The Parent class has a link span from Grandparent whereby it even adds in its own method the parentMethod. Then Child, from Parent, provides his method the childMethod. When an object of the Child class is created (childObject), it can access not only its method but also the methods contained in its, Parent class, and Grandparent classes. Hence, this reflects how multilevel inheritance allows the child class to inherit behavior in relation to various levels of a method access. The output demonstrates that the child object invoked the methods that were called on him, those pertaining to the grandparent, parent, and child classes respectively, reflecting their inherited structure.

Polymorphism

Polymorphism, from Greek words which mean "many forms", is a central concept of object-oriented programming, which means that the same function or method might act differently, depending upon the object or class through which it is being called. In essence, polymorphism is a certain quality through which the same interface is allowed to be used by an assortment of differing data types or classes; it allows significant flexibility in invoking methods over various objects. Polymorphism has two major types:

1. **Compile-time Polymorphism (Overloading):** It arises due to the existence of several methods with the same name that vary in their number or type of parameters. However, the notion of method

overloading is quite rare in Python because Python has not supported this concept the way Java or C++ has done. In Python, a method behaves at runtime, and therefore, method overloading is generally avoided by default or by variable-length argument lists.

Example 2.6: Concept of Compile-time Polymorphism

```
# Program to demonstrate the concept of compile-time polymorphism
class Operation:
    def sum(self, no1, no2=0, no3=0):  # Default values simulate overloading
        return no1+ no2 + no3

opObject = Operation()

# Calling a function with one parameter
print(opObject.sum(5))

# Calling a function with two parameters
print(opObject.sum(5, 10))

# Calling a function with three parameters
print(opObject.sum (5, 10, 15))
```

OUTPUT

```
5
15
30
```

The above program shows how to implement compile-time polymorphism (method overloading) in Python. Here, the sum method of the Operation class takes three arguments but uses assigned values of 0 for the other two arguments. This enables the method to be invoked with one, two, or three function arguments, simulating method overloading. In the case of the invocation with one argument, opObject.sum(5), the second and third arguments by default become 0, resulting in answer 5. When the method is invoked with two arguments, opObject.sum(5, 10), the second argument will determine the answer, and the third will be 0, giving an answer 15. Finally, when three arguments are supplied opObject.sum(5, 10, 15), the answer will be 30. This demonstrates how they behave differently with function invocation

based on the number of arguments supplied: that is how compile-time polymorphism using default arguments is achieved.

2. **Run-time Polymorphism (Overriding):** This kind of polymorphism is, on the other hand, a trend in Python. It occurs when a method is defined with the same name in a child class as that of one in the parent class, only the implementation in the child class is much different from that in the parent class. This allows various behaviors to be performed on runtime based on the type of object making a method call. This is also known as method overriding.

Example 2.7: Concept of Run-time Polymorphism

```
# Program to demonstrate the concept of run-time polymorphism
class Animal:
    def makSound(self):
        print("Some generic animal sound.")

class Dog(Animal):
    def makeSound(self):
        print("Bark!")

class Cat(Animal):
    def makeSound(self):
        print("Meow!")

dogObject = Dog()
catObject = Cat()

# Calling the same method on different objects
dogObject.makeSound()
catObject.makeSound()
```

OUTPUT

```
Bark!
Meow!
```

The above program shows how a run-time polymorphism in Python works. Run time polymorphism is also known as method overriding. The Animal class has a makeSound method, which prints a generic animal sound. The subclasses of Animal include Dog and Cat, each of which

overrides the makeSound method to output a specific sound pertaining to that animal, a bark for the Dog and a Meow for the Cat. When the objects of Dog and Cat (dogObject and catObject) are created, the makeSound method is called on both objects. Even though the method is being called on both objects by the same name (makeSound), the actual method that gets executed is determined at runtime based on the type of the object. So, for dogObject, the method from Dog class is executed, which prints "Bark!"; and for catObject it is executed from Cat class, which prints "Meow!" This dynamic dispatch is clearly seen when the method called is not the one defined in the reference but the one actually defined in that class (the object type).

Note:

- **Inheritance provides a way for objects of some classes to inherit properties and behaviors from objects of other classes, providing an easier way to reuse code and reduce redundancy.**
- **Polymorphism allows a single interface to stand for different underlying forms (such as two methods sharing the same name behaving differently according to huge differences in the arena of necessity).**
- **Method overriding is a feature of polymorphism allowing the subclass to implement its own version of the method declared in a parent class.**
- **Inheritance helps in establishing a hierarchical relationship among classes, which eases the organization and maintenance of code.**
- **Polymorphism adds flexibility and scalability to the objects, treating such objects as instances of their parent class or interface and still calling implementations dedicated to subclasses.**

Check Your Progress-1

a) Inheritance allows a subclass to inherit properties and methods from its parent class. (True/False)

b) Polymorphism means that a method must have the same implementation in all classes. (True/False)
c) Method overriding is an example of compile time polymorphism. (True/False)
d) Inheritance reduces code redundancy by allowing code reuse. (True/False)
e) A subclass cannot modify the behavior of a method inherited from its parent class. (True/False)
f) Polymorphism allows objects of different classes to be treated as objects of a common parent class. (True/False)
g) Inheritance is only used to extend the functionality of a class and cannot alter existing functionality. (True/False)
h) Polymorphism and inheritance are completely independent and cannot be used together. (True/False)

2.3 ENCAPSULATION AND DATA HIDING

Encapsulation and data hiding are key principles of Object-Oriented Programming (OOP). They make sure that the inner workings of a class remain hidden from the outside world and that only essential information and functionalities are exposed. All this does not only add to system security but also makes it easier for future maintenance and ongoing projects of scaling and improvement processes.

Encapsulation: Operations on Data and Bindings

Encapsulation refers to the act of binding data (attributes), and methods (functions) that apply to those data into one cohesive unit. Most of the time it is accomplished by defining a class. It black-boxes the class implementation. For example:

- A BankAccount holds balance and account_number as its attributes and deposit() and withdraw() as its methods. The user doesn't need to understand how the methods work in the background but only knows what they can do for the user.

Data Hiding: Protecting Internal State

Since data hiding allows the setter and getter without exposing the public interface, its functionality is very important for:

- Preventing unauthorized access to crucial information by external entities.
- Control over how data is accessed or modified.

In Python, data hiding is applied by the access specifiers:

- **Public:** accessible from anywhere (default).
- **Protected:** this is indicated by a single underscore _ and suggests an intended restricted access but technically, it is still accessible.
- **Private:** this is represented by a double underscore __ which strictly forbids outside access to class attributes.

Advantages of Encapsulation and Data Hiding

1. **Increased Security:** Encapsulation provides a way to safeguard sensitive data through the use of access modifiers such as private, protected, or public. A controlled point of access (such as getter and setter methods) for specific aspects of an object shields it from unauthorized or accidental access. In this manner, accidental or fraudulent manipulation of the data is limited and also protects the object state.

2. **Data Control:** Encapsulation provides you with a way to place rules and validation checks when attributes of an object are being modified. For example, you may check the inputs in the setter function before changing the property. This way, the state of the object is always in accordance with the expected line of thinking. For example, a setAge() function may confirm that age is a positive number before updating the attribute.

3. **Flexibility:** The architecture of the whole object can be changed without making the change to the portion that makes use of this object. For example, you could implement this data into a better algorithm or into a

database or into a file, and not even slightly change the external interface. This layer of abstraction protects the user of the class from being affected by internal changes, thus ensuring long-term flexibility in development and maintenance.

4. **Modularity:** Encapsulation associates related data and methods into self-contained units called classes. This makes codebases more organized and more easily understandable. Each class focuses on one task defined by the principle of single responsibility. Such modularity helps in maintaining, testing, and debugging, since each class can now be developed and maintained independently, without side effects on other parts of the application.

Example 2.8: Implementation of Encapsulation and Data Hiding

```
# Program to showcase implementation of Encapsulation and Data Hiding
# Encapsulation with Public Access

class Car:
    def __init__(self, company, color):
        self.company = company # Public attribute
        self.color = color  # Public attribute

    def displayData(self):
        print(f"Car company: {self. company}, Color: {self.color}")

# Creating an object
carObject = Car("Swift", "white")
car.displayData()
```

OUTPUT:

Car company: Swift, Color: white

Example 2.9: Implementation of Encapsulation and Data Hiding

```
# Program to showcase implementation of Encapsulation and Data Hiding
# Encapsulation with Private Attributes (Data Hiding)
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Private attribute
```

```
    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"{amount} deposited successfully.")

    def withdraw(self, amount):
        if amount > self.__balance:
            print("Insufficient balance!")
        else:
            self.__balance -= amount
            print(f"{amount} withdrawn successfully.")

    def getBalance(self):
        return self.__balance  # Getter method to access private attribute

# Creating an object
accountObject = BankAccount(2000)
accountObject.deposit(100)
print(accountObject.getBalance())
accountObject.withdraw(5000)
```

OUTPUT:

```
100 deposited successfully.
2100
Insufficient balance!
```

The programs shown in above two examples give an example of encapsulation and data hiding. The Car class in Example 2.8 has public attributes, while the BankAccount class in Example 2.9 has private attributes. The Car class encapsulates the properties (company and color) and methods within a single unit. These properties are public and can be accessed and modified both within the class and externally. The output of displayData() exhibits encapsulation by providing a neat interface to access the attributes. However, as the attributes are public, there is no restriction against direct access or constraints against making an invalid value assignment that may later ruin data integrity. For example, to change the color, somebody could set the color attribute to "".

The contrast, then, is drawn with the BankAccount class because its __balance attribute is kept hidden behind the double underscore prefix; direct access to the attribute from outside the class is not possible. It is only controlled through

method invocation: deposit(), withdraw(), and getBalance(). All these methods observe user supplied constraints on the parameters; for example, deposits must be at least positive amounts and withdrawal requests could not exceed the balance available. It ensures that the state of the account is consistent and shouldn't allow direct changes to the attribute that wouldn't typically happen during an accidental or malicious addition. The encapsulated methods form a clear and secure interface for data access of the object-the benefits of encapsulation and data hiding hence becomes apparent in comparison.

2.4 ABSTRACT CLASSES AND INTERFACE

Abstraction is a very powerful concept in object-oriented programming that allows the programmer to specify what the general structure of a class should look like without providing its whole implementation. This is accomplished with the use of abstract classes and interfaces which provide a blueprint for deriving classes.

Both abstract classes and interfaces serve to enforce consistency throughout the different areas of a program by insisting that all derived classes implement certain methods. This comes in very handy for large systems where several developers have to work on different components of the system since it ensures uniformity and thus eliminates implementation errors.

What is an abstract class?

An abstract class is a class that cannot be instantiated directly. Instead, it is meant to be used as a base from which other classes can derive. This includes:

- Abstract methods (methods without any implementation).
- Concrete methods (methods with a complete implementation).

Python supports abstract classes in the abc (Abstract Base Class) module.

Key features of abstract class:

1. **Blue print for the derived classes:** Abstract class defines methods that derived classes should implement, thus ensuring the same behavior.
2. **Partial implementation:** They may include concrete methods in some cases to distribute code among derived classes.

3. **No instantiation:** Abstract classes cannot be instantiated; creating an object of an abstract class leads to TypeError.

As shown in the Example 2.10, to define an abstract class, use the ABC class from the abc module. Abstract methods are defined using the @abstractmethod decorator.

Example 2.10: Program showing use of abstract class

```
# Program to showing use of abstract class
from abc import ABC, abstractmethod

# Abstract class
class Vehicle(ABC):
    @abstractmethod
    def startEngine(self):
        pass

    @abstractmethod
    def stopEngine(self):
        pass

    def honk(self):
        print("Vehicle is honking.")

# Derived class
class Car(Vehicle):
    def startEngine(self):
        print("Car engine started.")

    def stopEngine(self):
        print("Car engine stopped.")

carObject = Car()
carObject.startEngine()
carObject.honk()
```

OUTPUT:

```
Car engine started.
Vehicle is honking.
```

In the above the Vehicle class is defined as an abstract class with the ABC module in conjunction with the @abstractmethod decorator. The abstract methods include startEngine and stopEngine, which are not implemented in the

abstract class but must be implemented by all instantiable subclasses. In addition, the Vehicle class contains a regular method named honk, which can be inherited by its subclasses with a default implementation. The Car class is a concrete subclass of Vehicle: it implements the abstract methods startEngine and stopEngine, thus becoming concrete and available for instantiation. In this program there exists a sample object of the Car class called carObject. When the statement carObject.startEngine() is executed, the overridden method in the Car class runs and produces the output "Car engine started." Afterward, the honk method, defined in the superclass Vehicle, is called with the output "Vehicle is honking."

This opens the door to abstract classes, giving a general way to design subclasses in such a way that they make sure of general specifications of behavior, while letting subclasses provide their particular implementations against each one of the abstract methods.

What is Interface?

An interface is a collection of methods that a class must implement. Unlike an abstract class, an interface does not provide any method implementation but solely focuses on a method signature. In Python, interfaces are implemented using abstract classes where all methods are abstract.

Distinctive Features of Interfaces

- **Contract Enforcement:** Interfaces make sure that all classes implementing the interface adhere to a predefined structure.
- **Multiple Inheritance Support:** Python provides for a class to implement multiple abstract external interfaces which in turn make possible a combined setting.
- **Language-agnostic Behavior:** Interfaces are a common concept across all programming languages, making Python’s implementation compatible with similar constructs in other languages.

Example 2.11: Program showing implementation of interface

```
# Program to showing implementation of interface
from abc import ABC, abstractmethod

# Interface
class PaymentGateway(ABC):
    @abstractmethod
    def processPayment(self, amount):
        pass

    @abstractmethod
    def refundPayment(self, amount):
        pass

class PayPal(PaymentGateway):
    def processPayment(self, amount):
        print(f"Processing payment of {amount} through PayPal.")

    def refundPayment(self, amount):
        print(f"Refunding payment of {amount} through PayPal.")

paypalObject = PayPal()
paypalObject. processPayment (1000)
paypalObject.refundPayment(500)
```

OUTPUT:

```
Processing payment of 1000 through PayPal.
Refunding payment of 500 through PayPal.
```

The above program shows how to implement interfaces in Python using the ABC module and the @abstractmethoddecorator. The PaymentGateway class represents an interface where the abstract methods processPayment and refundPayment are declared but we cannot define their implementations. In this way, any class that inherits from PaymentGateway must provide methods according to a preset contract for consistent behavior across different payment gateway implementations.

The PayPal class implements the PaymentGateway interface by providing concrete definitions for both the processPayment and refundPayment methods. The processPayment method simulates processing a payment in a specified amount through PayPal in the real sense while the refundPayment method

achieves this goal by simulating refunding of some amount from the same payment gateway.

The PayPal class creates an object paypalObject. Methods are called through this object. When the statement *paypalObject.processPayment(1000)* is executed, it returns "Processing payment of 1000 through PayPal." In the same way, the statement *paypalObject.refundPayment(500)* returns "Refunding payment of 500 through PayPal.". This program demonstrates how interfaces support flexible and extendable designs that enforce subclasses to realize a certain set of required methods, thus supporting polymorphism and establishing a common interface.

Abstract Classes Vs Interfaces

Abstract classes and interfaces are powerful tools used in the design of a robust scalable system. They help to organize code, maintain certain standards, and allow for reuse of different components, hence finding such a broad application in object-oriented programming. For programmers who know how to properly employ them, abstract classes and interfaces afford the programmer the creation of systems which will, in the course of time, be simpler to maintain, expand, and troubleshoot. Table 2.1 shows the differences between abstract classes and interface.

Table 2.1: Abstract Classes vs Interfaces

Feature	Abstract Classes	Interfaces
Definition	Can have both concrete and abstract methods.	Focus on defining method signatures only.
Purpose	Used for partial implementation.	Used as a contract for implementation.
Instantiation	Cannot be instantiated directly.	Cannot be instantiated directly.
Inheritance	Single or multiple inheritance allowed.	Multiple inheritance allowed.
Implementation	May include shared code for derived classes.	Does not include any implementation.

Note:

- **Encapsulation is an influential approach for combining the related data and methods within a class, thereby hiding the inner details of implementation from the external world.**
- **Data hiding is a mechanism, where the attributes of an object are made inaccessible for modification.**
- **Encapsulation can improve security for sensitive information from unauthorized access.**
- **Abstract classes provide a template from which other classes can derive, allowing them to implement abstract methods while inheriting the common functionality.**
- **Interfaces are completely abstract specifications that depict a contract to which the class must adhere, that is: an interface should declare member methods that classes must provide specific implementations for.**
- **Both abstract classes and interfaces provide polymorphism, in that objects can nonetheless be treated generically yet maintain their specific behavior.**

Check Your Progress-2

a) Data hiding refers to the practice of restricting the access privilege of some _____ within a class so that it denies unauthorized access to exposed data.
b) Encapsulation is used to ensure the _____ of an object is maintained by validating input prior to changing the values.
c) An abstract class gives a _____ to other classes and thereby enforces their proper implementation.
d) An abstract class as well as an interface allows _____, treating different classes similarly while still acting in different ways.
e) Encapsulation is henceforth a better measure to ensure _____ of sensitive information by allowing access control via getters and setters.
f) Encapsulation is the process of combining attributes and functions into a single unit called a _____.

2.5 LET US SUM UP

This unit covered key object-oriented programming concepts. Inheritance gives you code reuse and hierarchical relationships, while polymorphism allows methods to perform differently in different contexts through overriding and overloading. Encapsulation and data hiding apply to the bundling of data in classes with restricting access to the data by private and public modifiers. Abstract classes and interfaces are blueprints for extending systems, wherein abstract classes open onto implementation while interfaces maintain an equilibrium by requiring certain methods of implementation classes. The principles work in harmony to provide flexible, maintainable designs for systems.

2.6 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

1-a True
1-b False
1-c False
1-d True
1-e False
1-f True
1-g False
1-h False
2-a Attributes
2-b State
2-c Blueprint
2-d Polymorphism
2-e Security
2-f Class

2.7 ASSIGNMENTS

1. Define encapsulation and explain its importance in object-oriented programming.
2. Explain the purpose of inheritance and how it promotes code reuse.
3. What is polymorphism, and how does it improve flexibility in programming?
4. Differentiate between method overriding and method overloading with examples.
5. Compare and contrast the use of abstract classes and interfaces with examples.
6. Write a Python program which implements the following functionalities:
    - Design a system for different types of employees (e.g., Manager, Engineer, Intern). Each employee type should have a work() method, which behaves differently based on the employee role. Use inheritance and method overriding to implement this functionality.
    - Create a class Student with private attributes for name, age, and grade. Use getter and setter methods to access and modify these values while ensuring validation (e.g., grade should be between 0 and 100).
    - Create an interface Vehicle with methods start(), stop(), and fuelEfficiency(). Implement this interface in classes Car and Truck, each with unique behaviors for starting, stopping, and calculating fuel efficiency.
    - Create an abstract class Shape with a method area(). Derive two classes, Circle and Rectangle, where each class implements the area() method to calculate the area specific to its shape.
    - Develop a simple payment system where an abstract class PaymentMethod defines the method processPayment(). Implement this method in subclasses CreditCard and PayPal, with different logic for processing payments.

Unit-3: Exception Handling 3

Unit Structure

3.0 LEARNING OBJECTIVE

After studying this unit student should be able to:

- Define and understand the concept of exceptions.
- Identify some common run-times errors in the Python programming language and learn how to tackle them.
- Effectively use try, except, and finally blocks to handle exceptions.
- Understand the purpose and use of the else clause in exception handling.
- Raise and handle custom exceptions with the raise keyword.
- Design robust Python programs by utilizing advanced exception-handling mechanisms.

3.1 INTRODUCTION

Errors and exceptions are an indispensable part of programming. Errors tell us that our program is not functioning as expected. Scenarios that can stop program execution are, for example, if a number is being divided by zero or if one tries to open a non-existent file. Such situations in Python bring about exceptions.

Exception handling is a powerful tool that is very useful to the programmer to gracefully manage unexpected conditions without aborting the program. The built-in mechanisms in Python allow the programmer to catch the exception, check on its cause, and possibly recover from it.

3.2 UNDERSTANDING EXCEPTIONS IN PYTHON

What are exceptions?

They are the runtime errors which cause program interruption. They do not belong to the group of errors caused by incorrect syntax, but are caused by program execution. For example, consider these exceptions:

- **ZeroDivisionError**: Division by zero.
- **ValueError**: The value provided to a function is not within its expected range/type.
- **FileNotFoundError**: Attempting to read a file that does not exist.

Key Characteristics of Exceptions

- Unpredictable: Exceptions arise from unexpected events, such as invalid input, unavailable resources, or logical errors within the program.
- Interruptive: An exception will cause the normal flow of the program to end, immediately unless the exception is handled.
- Recoverable: By suitably putting up an exception-handling interface, the application continues ordinarily or returns to execution.

Common Causes of Exceptions

- Invalid Input: Providing data that is either of the wrong type or that has an incorrect value. Example: Supplying a string rather than a number to a mathematical operation.
- File Operations: Operations involving non-existent files or files with no permission to access them. Example: Attempting to access a non-existing file.
- Arithmetic Operation: When operations of an undefined kind are performed between two mathematical entities.
- Out-of-Bound Indexing: When a user accesses an item of a list or dictionary incorrectly using an invalid index or key.
- Another wicked source of exceptions are network errors: all sorts of issues associated with the loss of connectivity or lack of availability of the server in networked applications.

Error V/s exception

Though the terms "errors" and "exceptions" are sometimes used interchangeably, they have different meanings:

- **Errors**: Issues caused by the programmer's own syntax or logic that may be unresolvable. Examples include errors like:
    - Syntax errors (missing colons or parentheses).
    - Indentation errors.
- **Exceptions**: These are run-time events that interfere with the continuity of a program execution but are manageable through the exception-handling mechanisms.

The Python program given in Example 3.1 shows the difference between error v/s exception.

Example 3.1: Program to differentiate between Error and Exception

```
#Program to differentiate between Error and Exception

# Example of Error
if True    # Omission of  colon
   print("Above statement will generate syntax error.")

# Example of Exception
divOutput = 50/0  # Generates ZeroDivisionError
print("divOutput.")
```

In above code difference between Error and Exception is demonstrated:

- **Error**: In Python, every if statement must be followed by a ':' to indicate the start of the block of code. By not including the colon after if True, you get a SyntaxError. The code could not be executed by the interpreter of Python, which reported a SyntaxError because of the missing colon.
- **Exception**: In the try block, the value 50 is attempted to be divided by 0. Since division by zero is not permissible in mathematics, it raises a ZeroDivisionError in Python. Instead of crashing the program, if the exception was handled correctly in the except block, an appropriate message to the user could have been displayed"

Various types of Exceptions in Python

Python has many built-in exceptions for a variety of error types. These exceptions are structured in a hierarchy where the BaseException is positioned at the top. Table 2.1 lists exception, their reason and an example.

Table 3.1: List of Exception

Name	Reason	Example
ZeroDivisionError	Divide by zero.	50 / 0
ValueError	Invalid value provided for a function or method.	float("abc")
IndexError	Fetching a value from invalid index	lstNums [10] for a list of size 5
KeyError	Accessing an invalid key from a dictionary.	dictData['invalid_key']
FileNotFoundError	Trying to open a non-existent file.	open(missingFile.txt')
TypeError	Using a mathematical operation on incompatible types.	"textualdata" + 10

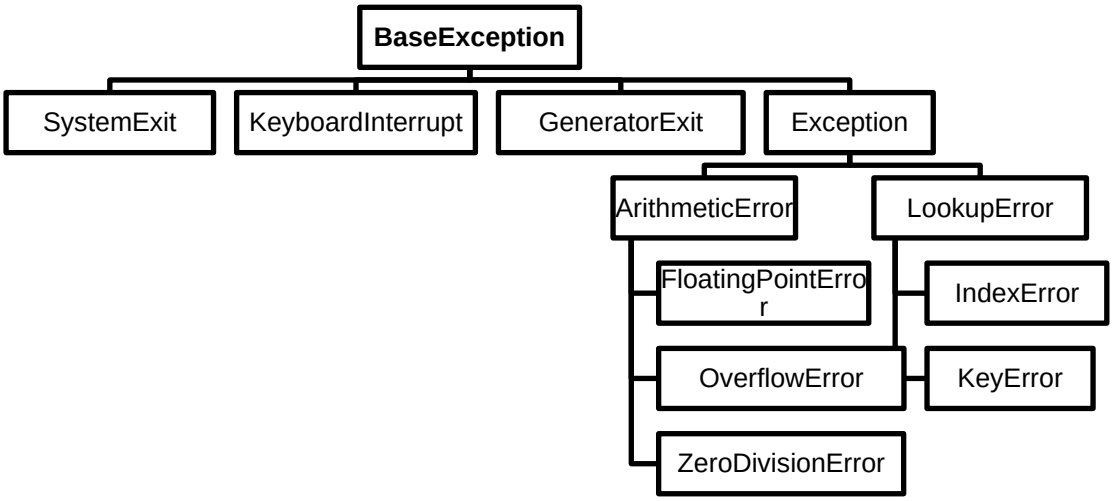
Exceptions Hierarchy

In Python, exceptions are arranged in a hierarchy starting from subclass BaseException.

- BaseException: It is the top-level class for all exceptions.
    - Exception: It is the base class from which most runtime exceptions are derived.
        - ArithmeticError would raise exceptions including the ZeroDivisionError.
        - A LookupError would raise exceptions including the IndexError and KeyError.

Let us Visualize the hierarchy of some of the classes of Exception as shown in the Figure 3.1

Figure 3.1: Hierarchy of Exceptions



Benefits of using exception handling

- Better User Experience Betterment: Allows the program not to terminate suddenly and provide appropriate error messages.
- Program Continuity: Ensures execution of important functions like writing back files or cleaning resources.
- Debug Support: Helps identify errors and log them for further improvement.

3.3 THE *try*, *except*, AND *finally* BLOCKS

Python's try, except, and finally constructs provide the basis for the exception-handling mechanism. These blocks enable python programmers to distinguish those parts of codes that are prone to errors, grant ability to control exception conditions gracefully, and guarantee the executing of cleanup operations irrespective of an error.

Structure and Purpose of *try*, *except* and *finally* blocks

1. try block:
   - Code that might throw an exception.
   - If an exception is raised, the rest of the try block is skipped, and control goes to the corresponding except block.
   - If no exception occurs, the try block is completed normally, and the except block is passed over.

2. except block:
    - Handles specific or general exceptions raised in the try block.
    - You can define multiple except blocks to handle different exceptions differently.
3. finally block:
    - Contains clean-up code that is guaranteed to run after the try block, regardless of whether an exception occurred.
    - Commonly used to release resources like file handles, database connections, or network sockets.
4. Else block(Optional):
    - An else block may only be executed if the try block has no exceptions.
    - This will usually contain code that should be executed if no exceptions happen.

Code snippet in example 3.2 demonstrates the basic structure of try, catch and finally block.

Example 3.2: Try, Catch and Finally block

```
#Code snippet to demonstrate try, catch and finally block
try:
    # Code that may generate an exception
except SpecificException as e:
    # Code to handle the exception
except AnotherException:
    # Handle another exception
else:
    # Code to run if no exception occurs
finally:
    # Cleanup code that always executes
```

Let us explore the structure of exception handling in detail with examples and use cases.

1. **The *try* block:** It contains the code that might throw exceptions. It isolates the error-prone logic from the rest of the program. If an exception occurs, the rest of the statements in the try block will not be executed and the interpreter will jump to the appropriate except block.

Example 3.3 demonstrates the usage of try block for exception handling.

Example 3.3: Try block

```
#program to demonstrate try block
try:
    no = int(input("Input a valid no: "))
    print(f"Output: {100 / no}")
except ZeroDivisionError:
    print("Error: Divide by zero")
except ValueError:
    print("Error: Not a valid no")
```

OUTPUT– SCENARIO 1:

Input: 5

Output: 20

OUTPUT– SCENARIO 2:

Input: 0

Output: **Error: Divide by zero**

OUTPUT– SCENARIO 3:

Input: “Number”

Output: **Error: Not a valid no**

The above python program demonstrates how to handle exceptions with a try block containing multiple exceptions. The processes in this program illustrate cases where user input may cause different types of runtime errors.

If the user inputs 0, then a ZeroDivisionError is raised and "Error: Divide by zero" is shown. If non-numeric input is provided, then the ValueError is raised, and "Error: Not a valid no" is shown. Thus, the program in such a situation will handle the errors without crashing.

2. **The *except* block:** This block catches and handles exceptions raised in the try block. It allows:
    - Handling a specific exception type
    - Handling multiple exception types in one block
    - Catch-all with a generic except

    Example 3.4 demonstrates the different ways of handling exceptions in except block.

Example 3.4: Except block

```
#program to demonstrate except block
# Handling Specific Exceptions
try:
    no = int(input("Input a valid no: "))
    print(f"Output: {100 / no}")
except ZeroDivisionError:
    print("Error: Divide by zero")
except ValueError:
    print("Error: Not a valid no")
```

```
# Handling Multiple Exceptions
try:
    no = int(input("Input a valid no: "))
    print(f"Output: {100 / no}")
except (ZeroDivisionError, ValueError):
    print("Either division by zero  or the input was not valid.")
```

```
# Handling All Exceptions
try:
    no = int(input("Input a valid no: "))
    print(f"Output: {100 / no}")
except Exception as ex:
    print(f"An error occurred: {ex}")
```

The above python program covers use of except block in three different ways: It makes use of separate except clauses to handle ZeroDivisionError and ValueError, displaying specific messages; it demonstrates how you can group exceptions in a tuple and handle multiple exceptions together in a single except

block by outputting a common message; and lastly it shows the catch-all except Exception block which catches any error and prints the error message dynamically for better debugging.

3. **The *else* block:** This block in exception handling is optional and runs only when the code in the try block has no exceptions. This can be useful for code depending on the successful completion of the try block. Example 3.5 demonstrates the usage of the else block in exception handling mechanism.

Example 3.5: Else block

```
#program to demonstrate else block
try:
    no = int(input("Input a valid no: "))
    print(f"Output: {100 / no}")
except ZeroDivisionError:
    print("Error: Divide by zero")
else:
    print("Code in try block executed without any errors!")
```

OUTPUT– SCENARIO 1:

Input: 5

Output: 20

Code in try block executed without any errors!

OUTPUT– SCENARIO 2:

Input: 0

Output: **Error: Divide by zero**

The above python program demonstrates the usage of else clause along with exception handling. The else part executes as long as there is no exception raised in the try part. If a number is entered that is valid, the computation result is displayed, along with the message "Code in try block executed without any errors!". If a ZeroDivisionError is raised, where the user inputs 0, the exception block handles the exception and the else block is not executed.

4. **The *finally* block:** In this block the clean-up code will run after the try and except blocks, no matter whether or not an exception was raised. Usually, it is used for resource releases like file or database connection closures. Example 3.6 demonstrates the usage of finally block in exception handling mechanism.

Example 3.6: Finally block

```
#program to demonstrate finally block
try:
    demoFile = open("demo.txt", "r")
    txtData = demoFile.read()
    print(txtData)
except FileNotFoundError:
    print("Demo.txt file not found.")
finally:
    print("Closing the file in finally block")
    demoFile.close()
```

The above mentioned code showcases exception handling using the `finally` block. Here, efforts are made to open and read the file `demo.txt`. If the specified file does not exist, a FileNotFoundError exception is generated, displaying the message: "Demo.txt file not found." Either way, in order to ensure that the file gets closed and proper resource management is done, a `finally` block will always execute.

Putting All Elements Together

By including all the components of exception handling mechanism (try, except, else, finally), we can create fairly powerful exception handler code that manages the errors gracefully and guarantees the cleanup of the resources used. Example 3.7 showcases the usage of all four components of exception handling.

Example 3.7: All blocks of Exception Handling

```
#program to demonstrate the usage of all blocks
try:
    no = int(input("Input a valid no: "))
except ZeroDivisionError:
    print("Error: Divide by zero!")
except ValueError:
    print("Error: Not a valid no!")
else:
    print(f"Output: {100 / no}")
finally:
    print("Execution completed Successfully!")
```

The above program demonstrates how to work with all blocks of exception handling, that is: `try`, `except`, `else`, and `finally`. It tries to read the user input and casts it into the integer type. If that input is not valid (non-numeric), then a ValueError exception is being caught and the error message "Error: Not a valid no!" will be displayed. This will prompt a ZeroDivisionError exception in case the user enters `0` because in that case `100 / no` will not be possible, and the negative exception message will be displayed. If no errors occur, the `else` block executes the division and prints the result of `100 / no`. The `finally` block always executes and prints "Execution completed Successfully!" which guarantees a normal completion of the program, if any exceptions were thrown or not.

Note:

- Exceptions are errors that arise when running a program and can be handled using exception handling mechanisms.
- The try block contains the code that one anticipates might raise an exception while the except block deals with it.
- The else block will only occur if the code within try did not throw any errors.
- The finally block will always get executed, letting you do some cleanup work, such as closing files or releasing resources.

Check Your Progress-1

a) A Python program terminates if an exception is not handled.
b) The try block is used to test a block of code for exceptions.
c) The except block is executed only when the try block does not raise an exception.
d) The else block is mandatory in exception handling.
e) You can handle multiple exceptions in a single except block using a tuple.
f) If a file operation raises an exception, the finally block ensures the file is closed properly.

3.4 RAISING EXCEPTIONS

The raise statement in Python is used where one needs to raise an exception on purpose. This is useful to enforce rules, validate inputs, and provide specific signalization of error in certain cases. The raising of an exception will ensure effective communication of errors while ensuring the arbitration at the appropriate level of your program.

When should we raise exceptions?

- Input Validation: Input data does or does not conform to set standards.
- Flow Control: An alternative is to interrupt a thread of execution when something unexpected occurs.
- Custom Messages: Provide error messages that have meaning specific to the application.

The basic syntax for raising an exception is as follows:

raise ExceptionType("Error message")

- ExceptionType: Denotes the type of exception to raise (e.g., ValueError, TypeError, RuntimeError).
- Error message: A string value which provides additional details about the exception.

Raising built-in exceptions

Python offers a variety of built-in exceptions that may be raised when warranted. Common examples include:

- ValueError: When a value is invalid.
- TypeError: For an invalid type.
- KeyError: As a result of a missing key in the dictionary.
- FileNotFoundError: For files that cannot be accessed.

Example 3.8 shows how to raise built-in exceptions in python.

Example 3.8: Program to raise built-in exception

```
#Program to raise built-in exeption
def getSquareRoot(no):
    if no < 0:
        raise ValueError("It's not possible to find the square root of a negative number.")
    return no ** 0.5


try:
    output = getSquareRoot(-9)
    print(f"Square root is: {result}")
except ValueError as e:
    print(f"Error is: {e}")
```

OUTPUT

Error is: It's not possible to find the square root of a negative number.

The above example shows how to indicate and catch a built-in exception in Python. The function getSquareRoot(no) calculates the square root of a number. If the input is negative, this function raises a ValueError with the message:"One cannot find the square root of negative numbers." The try block here calls the function with -9, which raises the exception. The except block catches the error and prints the message.

3.5 CUSTOM EXCEPTION

Python allows developers to create custom exceptions tailored to the specific needs of an application. This feature is useful when built-in exceptions don't sufficiently describe the errors that may occur in your program.

Why Use Custom Exceptions?

- User-friendly error messages: Custom exceptions increase the descriptiveness of error messages.
- Program Specific Needs: Address errors specific to the program domain, like business rule validation.
- Custom exceptions allow modular and specific error handling.

How to Create a Custom Exception?

Custom exceptions are created in Python by subclassing the built-in Exception class or any of its subclasses. That way, the new exception will inherit the properties and behavior of the base exception class. Following code snippet creates a simple custom exception class which can be used to raise custom exceptions:

```
class CustomException(Exception):
    """Custom exception with a descriptive error message."""
    pass
```

Example 3.9 shows the python code to implement custom exception:

Example 3.9: Program to implement custom exception

```
# Program to implement custom exception
class NegativeNoError(Exception):
    """Exception raised for invalid input: Negative no."""
    def __init__(self, no):
        self.no = no
        super().__init__(f"Negative number is inputted: {no}")
```

```
# Example usage
try:
    no = int(input("Input a positive number: "))
    if no < 0:
        raise NegativeNoError(no)
    print(f"Number is valid: {no}")
except NegativeNoError as e:
    print(e)
```

OUTPUT – SCENARIO 1:

Input a positive number: 10

Number is valid: 10

OUTPUT – SCENARIO 2:

Input a positive number: -10

Negative number is inputted: -10

The above program illustrates the creation and use of user-defined exceptions in Python. The NegativeNoError class, which is inherited from the built-in Exception, is used to handle exceptions when a negative number is given. This class is initialized with a custom error message. The try block takes user input and converts it into an integer; in case of a negative input number, the NegativeNoError exception is raised. The except block catches this exception and prints the custom error message. The program, for instance, will print, "Number is valid: 10" when the input is 10 and will print, "Negative number is inputted: -10" in case you input -10.

Taking care of best practices while working with custom exception

- Meaningful Names: The class name should clearly convey the nature of the error (e.g., FileFormatError, InvalidInputError).
- Documentation Strings: A descriptive line mentioning the purposes of the exceptions.
- Modular: Put related custom exceptions within a single module.

- Inheritance: Always inherit from Python's built-in Exception class, ensuring you can tap into the whole Python exception-handling.
- Provide Contextual Information: Add characteristics or methods so that it can better explain what went wrong.

Note:

- The raise statement in Python is used to force an exception.
- The raise statement can also have a message to provide a little context about what went wrong.
- Custom exceptions are those exceptions defined by users and derived from the built-in Exception class of Python.
- Custom exceptions can have a constructor (__init__) that accepts extra information when raised.
- Defining meaningful custom exceptions helps in handling specific error scenarios in a program.

Check Your Progress-2

a) A custom exception class should have a meaningful _____ method to describe the exception's details.
b) When creating a custom exception, the class name should follow the _____ naming convention.
c) To raise a custom exception, use the statement: raise _____("Error message").
d) Custom exceptions improve _____ by allowing developers to define meaningful and descriptive errors.
e) To define a custom exception, you need to create a class that inherits from the built-in _____ class.

3.6 LET US SUM UP

In this unit, we learned all about exception handling. You have learned what exceptions are, how they differ from errors, and why it is essential for them to

be handled in Python programs. The four blocks of exception handling (try, except, else, and finally) were discussed, and how together they help in the graceful management of runtime errors. The different method of explicitly raising exceptions was also discussed: the raise statement allows you to generate exceptions under what you specify as the conditions in your program. You have also learned how to define specific error types for your program needs and how to create and use custom exceptions. With this particular knowledge, you can write very solid user-friendly computer programs that can turn the errors and exceptions into opportunities for increasing robustness and maintainability of applications.

3.7 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

1-a True
1-b True
1-c False
1-d False
1-e True
1-f True
2-a __str__
2-b PascalCase
2-c CustomException
2-d Code Readability
2-e Exception

3.8 ASSIGNMENTS

1. Discuss in detail the difference between the compile-time errors and runtime exceptions.
2. Explain the purpose of using exception handling in a program.
3. What is the role of the try block in exception handling?
4. Differentiate between specific and generic exception handling based on examples.
5. What is the importance of the finally block in exception handling?

6. Describe the significance of handling multiple exceptions in one single try block.
7. Write a program to perform the activities given:
    - Write a program to do simple arithmetic operations such as addition, subtraction, multiplication, and division using two user-input numbers and catch exceptions arising out of them.
    - Write a program that handles KeyError by raising a message for unknown keys in a dictionary.
    - Design a program where an exception is defined by the user and should be raised if for the given input the user value is not in the given range (ex: 1 to 100).
    - Develop a program that catches exceptions in typecasting operations such as str to int and provides reasonable explanations accordingly.
    - Design a program responsible to catch ModuleNotFoundError if the user imports some missing module and suggest other alternatives to the user.

Unit-4: File Handling & GUI

4

Unit Structure

4.0 LEARNING OBJECTIVE

After studying this unit student should be able to:

- Understand the importance of file handling in python programming
- Differentiate between various modes ('r', 'w', 'a', etc.) and use them effectively.
- Identify the basic structure and utility of the Tkinter library in GUI applications.
- Integrate file handling with GUI using Tkinter to make user-friendly applications.

4.1 INTRODUCTION

In the previous chapter, we studied a concept called exception handling, which is one of the most important things to keep in mind when creating robust error-resilient programs. Exception handling basically helps the programmer to anticipate potential runtime errors and handle them gracefully, so that the application still functions as expected even when such unexpected things happen. For instance, if a program attempts to open a file that doesn't exist, a FileNotFoundError might occur. With exception handling, this error can be managed by providing the user with a meaningful message or alternative actions, such as prompting for a different file path.

Building on that foundation, this chapter focuses on two other important goals of computer programming: **The data handling** and **the presentation of data in a user-friendly way**. File handling and GUIs make Python very efficient with just such tasks. This section will cover these concepts and explain why they are integral to creating practical, robust applications.

GUI applications, for instance, can ask for user inputs to be saved for future retrieval. If no exception handling is done, the GUI program might crash because of mistakes, such as non-existent file paths or file permission problems. Incorporating exception handling principles (which was covered in the last chapter) together with optimal file handling techniques, programmers

can develop robust applications capable of dealing with files quite readily and also recovering in case of errors.

This chapter takes these ideas further with the introduction of Tkinter, Python's built-in GUI development library, along with a showcase of how file handling and GUI technologies can work hand in hand to produce interactive applications that are robust and error-tolerant.

What is file handling?

File handling refers to interaction with file systems and storing information in order to read, write, or modify data stored on the disk. Unlike data stored in variables during program execution, files offer persistent storage that allows the data to survive the program termination. These are the most notable features of file handling:

- **Data storage**: It allows users to store data permanently on disk; compared to temporary storage in RAM using variables.
- **Portability**: Files can be easily shared and opened in different systems.
- **Organization**: Efficient storage of logs, user preferences, and structured datasets.
- **Reusability**: Read and process information already stored without having to input it again.

The built-in functions and modes for performing file-operation tasks for Python are as follows:

- R (read): To read the content from files.
- W (write): To write the content to files.
- A (append): To append and still preserve the existing content.
- B (binary): For non-text data, such as images and videos.

File handling is very important in many scenarios, like saving user data, writing logs, or reading from a configuration file. For example, an application can save user settings in a file for use the next time the app is run.

What is a GUI?

A Graphical User Interface allows the use of icons and activities such as click, double click etc. for interaction with software. GUI replaces conventional text-based input and considers elements like windows, buttons, text fields, and menus to create a more intuitively navigable, friendly application. Advantages of GUIs:

- **Ease** - Non-technical individuals can run the application directly without having to memorize commands.
- **Visual Feedback**- The GUI offers visual feedback on the actions performed, such as a message for the action of saving a file.
- **User Experience**: Modern software often depends on an interface that is both appealing and highly functional.

Tkinter library of Python is a powerful GUI toolkit in Python. It is pre-installed with Python and hence freely available to all developers. Some of the Tkinter objects usually known as widgets are as mentioned:

- **Labels**: To display static text.
- **Buttons**: Trigger actions.
- **Entry Boxes**: To capture user input.
- **Text Areas**: For multi-line input.

A login form that we have been using for accessing emails is an example of a simple GUI.

4.2 FILE HANDLING IN PYTHON

File handling is one of the core tasks in programming that makes it convenient for applications to interact with the data stored on a computer disk. Python makes file handling simple and powerful, placing tools at the developers' disposal to read, write, modify, and manage files. This section discusses how file handling is done and then shows practical implementation of different operations that can be performed on file.

Understanding File Handling

File handling refers to the creation, reading, writing, and appending of data in files. A file provides a means of persistent storage as opposed to volatile memory like RAM, which clears memory after program termination. Key characteristics of Python file handling features are:

- **Ease of Learning**: File handling in Python is simple and straightforward.
- **Versatile**: Supports both text and binary file operations.
- **Exception handling**: Runtime errors due to certain cases such as absence of files are taken properly into account.

File operations in python

In Python, file handling is a well-structured process that includes aspects of opening, manipulating, and closing files. Let us now look into these operations:

1. **Opening a file:** The first stage of file handling in python is to open a file. This is done by using the built-in function *open()* which returns a file object, acting as an interface to communicate with the file. The general syntax for opening a file in python:

   fileObject = open(nameOfFile, mode)

   - nameOfFile: The name (or path) of the file to be opened. If the file is not in the current directory, you must specify the full path of a file.
   - mode: The mode in which the file will be opened (e.g., read, write, append, etc.).

Table 4.1 shows the different modes in which a file can be opened in Python.

Table 4.1: Modes of opening a file

Mode	Description
'r'	Opens an existing file for reading.
'w'	Opens a file for writing. Creates a new file or overwrites an existing file.
'a'	Opens a file for appending. Adds data at the end of the file.
'x'	Creates a new file. Throws an error if the file already exists.
'b'	Binary mode. Used with 'r', 'w', or 'a' for binary files.
't'	Text mode. Default mode for handling text files.
'+'	Enables simultaneous reading and writing.

2. **Writing to a file:** The write() method allows you to write data into a file. When a file is opened in write mode ("w"), if the file already exists then the contents of that file are erased and new content is written. If the file does not exist it is created.

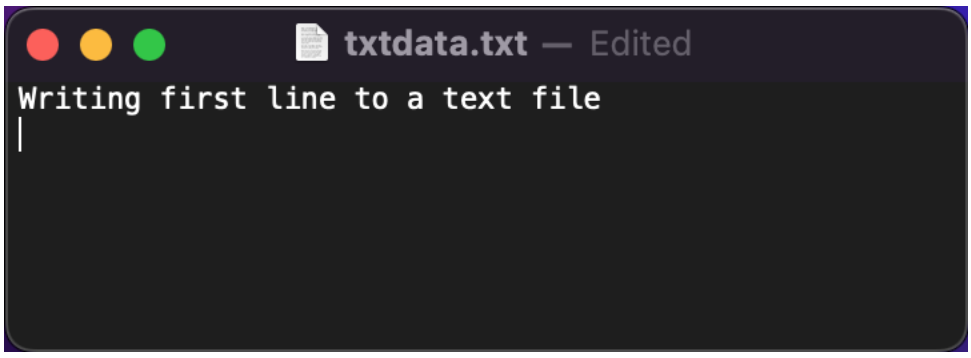
The Python program in Example 4.1 shows how to write the data to a file.

Example 4.1: Program to write the data to a file

```
#Program to write the data
with open("txtdata.txt", "w") as file:
    file.write("Writing first line to a text file")
```

The above Python code demonstrates how to write data to a file using the write() method. The statement *with open("txtdata.txt", "w") as file:* opens a file name txtdata.txt in write mode("w") with the help of the function open(). If this file is not present, it is created, while if it already exists, its contents are erased. The statement *file.write("Writing first line to a text file")* uses the write() method to write the string "Writing first line to a text file" to a file. The with open(...) as file: block helps to automatically close the file once the working with it is over, thus preventing any problem with the handling of the file. This approach will be helpful to store or log your data since, in this way, you can write text to the file with less complication. When we execute this program, we will not see any output on the screen. To see the output, we need to open the file txtdata.txt that would be stored in the directory where your Python program is. The file can be open in any text editor. A sample view of the file is shown in Figure 4.1.

Figure 4.1: Contents of file



3. **Reading from a file:** File reading is perhaps the most common file handling operation in Python. There are many methods that Python provides for reading data, each meant for a specific case. The choice of the method relies on the file structure and the amount of data you need to process at the same time. Methods for reading the data:
    a. **read() method:** This method reads the entire contents of a file in one go. It is useful when the file is small or when you need to read its contents all the way through.
    b. **readline() method:** This method reads one line from the file at a time. It is useful when reading logs or CSV data since log files and CSV files can be usually processed on a per-line basis.
    c. **readlines() method:** This method reads lines from the file and returns a list of strings. In this list, each element constitutes a line in the file.
    d. **Line-by-line reading with a loop:** As the term mentions we use a for loop to read the file object line by line. It works in a memory-efficient way because the whole file is not loaded into memory.

The Python program in Example 4.2 showcases the different ways of reading a file.

Example 4.2: Program to read the data from a file

```
# Program to show multiple ways to use read method

# Part 1 - Read the entire file using read() method
with open("txtfile.txt", "r") as file:
    data = file.read()
    print(data)

# Part 2 - Reading one line at a time
with open("txtfile.txt", "r") as file:
    singleLine = file.readline()
    while singleLine:
        print(singleLine.strip())  # Removes extra newline characters
        singleLine = file.readline()

# Part 3 - Reading all lines at once
with open("txtfile.txt", "r") as file:
    allLines = file.readlines()
```

```
    for singleLine in allLines:
        print(singleLine.strip())

# Part 4 - Reading data line by line using loop
with open("txtfile.txt", "r") as file:
    for singleLine in file:
        print(singleLine.strip())
```

In the above program, four different ways to read the contents of a text file called txtfile.txt are demonstrated. The first part of the code opens the file in the read mode ("r") and utilizes the read() method to read the entire content from the file in one go. The variable *data* stores the whole text from the file and the statement *print(data)* is then used to print the data fetched from the file.

The second part opens the file in the read mode. The program uses the readline() method to read one line at a time. In the while loop, as long as singleLine is of non-zero length, it reads from the file line by line. The statement *print(singleLine.strip())* uses the strip() method to remove any extra newline characters at the end and prints the line. After printing the readline() method is called once again to move to the next line of the input.

In the third part, the readlines() method reads all the lines of the file into a list. Each item in that list is a line from the file. The program cycles through this list, stripping newlines and printing each line.

The fourth part reads a file named txtfile.txt line by line using a for loop and prints each line after stripping the leading and trailing whitespace.

These methods give different ways of reading a file according to the needs of the program. The read() method is used to get the entire file content all at once if required, the readline() method is used for reading a file line by line, whereas the method readlines() is used whenever all

lines need to be handled in a list. The choice to use the method depends again on the file contents and size of the data.

4. **Appending data to a file:** Appending refers to adding data to the end of a file, leaving its previous contents intact. It is performed in append mode ("a").

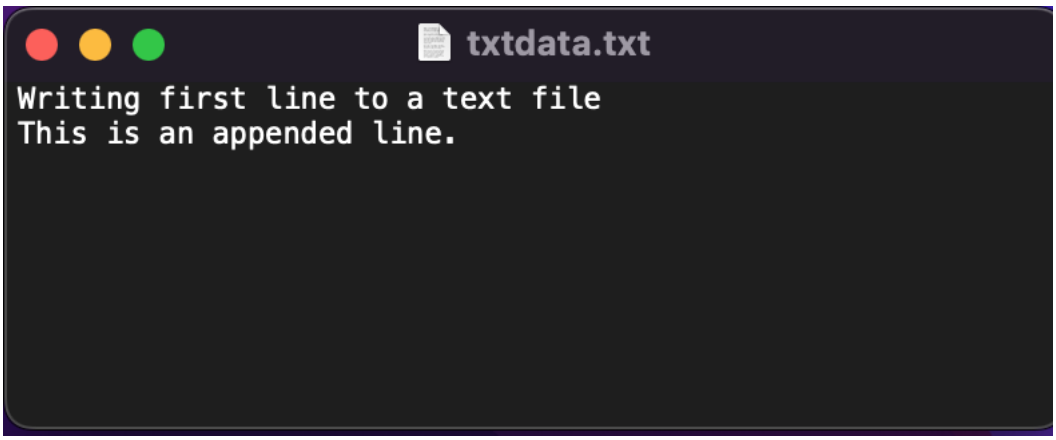
   The Python program in Example 4.3 shows how to append the data to an existing file.

Example 4.3: Program to append the data to a file

```
#Program to append the data
with open("txtdata.txt", "a") as file:
    file.write("\nThis is an appended line.")
```

The above python code shows how to append contents to an existing file using write() method in the append mode. The file is opened using the statement *with open("txtdata.txt", "a") as file:*, If the file does not exist, it will be created; otherwise, new content will be appended at the end of the file, without affecting the already existing data. The statement *file.write("\nThis is an appended line.")* then writes the text "\nThis is an appended line." to the file - here, the newline character(\n) ensures that the new text is written on a new line. This method is good for adding new information to a file without destroying the already existing content. The output will be similar to the one shown in Figure 4.2.

Figure 4.2: Output of an appended file



5. **Closing a file:** Closing a file releases resources and ensures that any buffered data is written to the disk. While Python automatically closes files opened within a with block, files opened manually must be closed using the close() method.

   The Python program in Example 4.4 shows how to explicitly close the file using the close() method.

Example 4.4: Program to close a file

```
#Program to close a file
file = open("txtdata.txt", "r")
data = file.read()
print(data)
file.close()
```

OUTPUT:

```
Writing first line to a text file
This is an appended line.
```

Above python code demonstrates a manual approach to close a file after reading the contents. It opens the file txtdata.txt in read mode ("r") using the open() method and reads the contents into the variable data using the read() method. The contents of variable data are then displayed on the screen using statement *print(data)*. After the reading is over, the statement *file.close()* (program manually calls the close method) closes the file. Closing a file is essential for releasing system resources and making sure that changes made to the file are appropriately saved.

6. **Dealing with binary files:** In Python programming the data of objects such as images, audio, video etc. are stored in binary format. Binary files are used to store data in a binary format. If we try to open the contents of a binary file in. a text editor, it will not be properly readable as in text files. Binary files store data by writing a sequence of raw bytes instead of characters readable for a human eye. Proper set of modes are needed while performing the reading and writing operations on a binary file.

a. **Writing to binary files:** In order to write binary data to the file, the file must be opened in binary write mode using "wb" option with the open() method. The write() method is used to pass data in the form of raw bytes, the string to be stored in binary form is prefixed with the letter "b".
b. **Reading from binary files:** In order to read binary data from the file, the file must be opened in binary read mode using "rb" option with the open() method. Once in this mode, the read() method then allows the reading of raw bytes from the file, it returns a byte object.

Example 4.5 shows the Python program that uses binary write and read mode to perform file operations.

Example 4.5: Program to show use of binary mode

```
# Program to show use of binary mode
with open("binFile.bin", "wb") as file:
    file.write(b"Writing binary data to a file")

with open("binFile.bin", "rb") as file:
    data = file.read()
    print(data)
```

OUTPUT:

```
b'Writing binary data to a file'
```

The purpose of the above program is to demonstrate the file handling operations in Python using the binary file. The statement *with open("binFile.bin", "wb") as file:* creates and opens, a binary file called binFile.bin. The "wb" mode (write binary) passed as a parameter to the open() method ensures that the file is treated as a binary file. The statement *file.write(b"Writing binary data to a file")* writes the string "Writing binary data to a file" into the file using the write() method. The letter "b" here ensures that the string will be stored in a binary format. Similar to text files in this case also when the program is executed the file named binFile.bin will be stored in the directory where the above

program is. Once the operation is over the file will be automatically closed.

The statement *with open("binFile.bin", "rb") as file:* opens the created binary file in binary read mode. The "rb" mode (read binary) passed as a parameter to the open() method ensures that the file is read as a binary file. The statement *data = file.read()* reads the binary data from the file and stores it in the variable data. The last statement *print(data)* then displays the contents on the screen.

Note:

- **File handling allows Python programs to read from and write to files to data storage.**
- **Once stored on the data storage files manipulation operations can be performed.**
- **The open() method is used to open a file. It requires the file name and mode ('r', 'w', 'a', 'b', etc.) as arguments.**
- **The *with* statement is used for file handling to ensure proper closure of the file, even if an exception occurs.**
- **Binary mode ('b') is used to handle binary files, such as images or audio, where data is read or written in bytes.**
- **Errors in file handling, such as trying to access a non-existent file, can be managed using exception handling with try and except blocks.**

Check Your Progress-1

a) The try block is used to handle errors that may occur during file operations. (True/False)

b) The with statement is not compatible with binary file operations. (True/False)

c) The close() method is optional if the with statement is used for file handling. (True/False)

d) The readlines() method reads a file line by line and returns a list of strings. (True/False)
e) The write() method can write multiple lines to a file at once. (True/False)
f) Binary mode ('b') is used to handle files containing text data. (True/False)

4.3 TKINTER FOR GUI DEVELOPMENT

GUIs help make python applications more user friendly by giving visual interaction instead of a sole reliance on text commands. Tkinter provides python programmers with the simplest and most powerful forms of toolkit for building GUI-based applications. It is included with Python installations and is an efficient option for developing interactive programs. This section will discuss the features of Tkinter, its components (widgets), and how to create dynamic GUIs.

What is Tkinter?

Tkinter is an acronym for Tk Interface, it is a Python binding to the Tk GUI Toolkit. Tkinter is one of the most widely used toolkits for creating desktop applications. It allows users to create applications that can have GUI windows. To build windows Tkinter provides the user with different widgets such as buttons, labels, and text boxes etc.

Why Tkinter?

- **Simple to Use:** Simple syntax and easy setup.
- **Cross-Platform:** Works on Windows, Mac OS, and Linux.
- **Wide variety of widgets:** Buttons, text boxes, labels, menus, etc.
- **Event-driven programming:** GUI applications respond to user actions such as clicks and keystrokes.
- **Preinstalled:** It's part of the standard installation of Python. You don't have to install Tkinter separately.

Anatomy of Tkinter Application

A Tkinter program broadly follows the following structure:

1. **Import of the Tkinter Module:** Either the entire Tkinter library or specific components are imported.
2. **Creates a Main Window:** The main window is the root container for all other widgets.
3. **Add Widgets:** Place elements within the window, such as buttons, labels, or entry boxes.
4. **Run the Event Loop:** The mainloop() command begins the program's event loop. This allows the application to be responsive to user events.

Example 4.6 shows the Python program that showcases the basic structure of a GUI program using the Tkinter module.

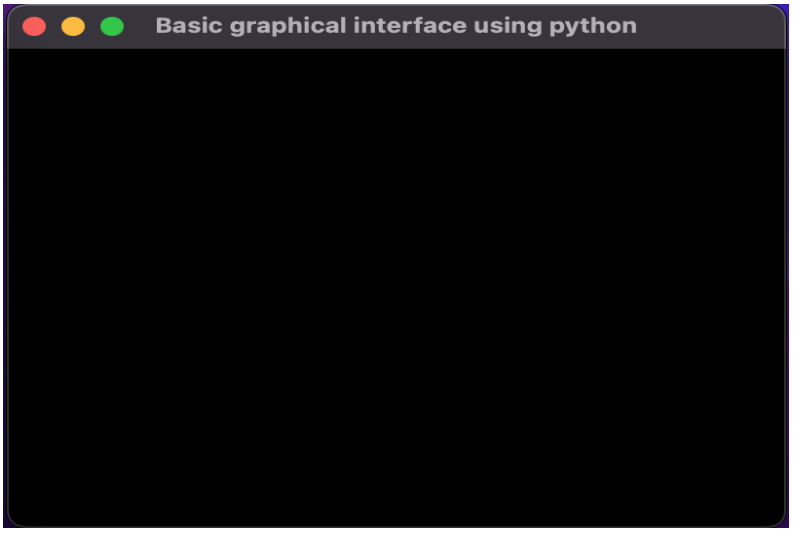
Example 4.6: Basic GUI using Tkinter

```
# Basic GUI using Tkinter
import tkinter as tk

# Create the main application window
root = tk.Tk()
root.title("Basic graphical interface using python")
root.geometry("400x300")  # Sets the size of window (width x height)

# Run the event loop
root.mainloop()
```

The code demonstrates the simplest GUI that can be created in Python, using the Tkinter library. The tkinter module is called and the application main window is created using tk.Tk(), which is used to create a root window for the application. The title() method sets the title of the window to "Basic graphical interface using python," and the geometry() method sets the window size to 400 pixels in width and 300 pixels in height. The mainloop() method at the end starts the Tkinter event loop and continues running it till the application window is closed, and the user interaction is attended to.

This is basic code that serves as a basis for the construction of more complicated GUI applications. The sample output of the code will be as seen in Figure 4.3.

Figure 4.3: Sample window using Tkinter



Common Tkinter Widgets

Widgets are the building blocks of a Tkinter GUI. They allow users to interact with the application visually, some of the commonly used widgets are as mentioned:

Label:

A Label widget enables static text to be displayed in a GUI application. Labels are often used as headings, instructions, or real-time status updates to a user. They can also be customized in terms of fonts, colors, and alignment according to the visual design of the application. Labels cannot be interacted with and are for display purposes only.

Button:

A Button widget causes actions to be performed once clicked. Any button can be configured to trigger a specific function or event, thus being an integral component for interactivity in GUI applications. Buttons may be customized with text, pictures, or icons and specific styles (color changes, resizing) for better usability.

Entry:

An Entry widget gives users a one-line box in which to enter short information such as names and passwords. It is often seen on forms or search bars to capture user information. Some Entry widgets can also have input restrictions, such as accepting only numeric characters or masking the input as the user is typing.

Text:

A Text widget accepts multi-line input and output of texts. This allows the handling of larger chunks of text, such as paragraphs or code snippets. In respect to the Entry widget, a Text widget gives a lot of flexibility. It supports things such as scrolling, text formatting, and editing, making it an advanced text input or output requirement.

Frame:

A Frame widget serves as a container that can be used to organize other widgets in a GUI. A Frame is used to handle grouping of widgets in a logical way, which can serve to structure an application layout. Frames may also be nested to allow for complicated layouts and can have borders, color, or padding for visual separation and clarity. Table 4.2 summarizes the widgets.

Table 4.2: Commonly used widgets in Tkinter

Widget	Description	Example
Label	Displays static text or images.	tk.Label(root, text="Login")
Button	Adds clickable buttons for user actions.	tk.Button(root, text="Go!")
Entry	Accepts single-line user input.	tk.Entry(root)
Text	Provides a multi-line text input area.	tk.Text(root)
Frame	Serves as a container for organizing widgets.	tk.Frame(root)

Example 4.7 shows the code snippets of a Python program which showcases the creation of different widgets using the Tkinter module.

Example 4.7: Code snippets to showcase creation of Widgets

```
#Program to showcase creation of Widgets
#Label
lbl = tk.Label(root, text="This is a Label!")
lbl.pack()  # Adds the label to the window
```

```
#Button
def on_ actionLogin ():
    print("Login Button Clicked!")

btnLogin = tk.Button(root, text="Login", command=actionLogin)
btnLogin.pack()
```

```
#Entry
txtEntry = tk.Entry(root)
txtEntry.pack()

def actionSubmit ():
    print(txtEntry.get())  # Fetches the user's input from entry widget

btnSubmit = tk.Button(root, text="Submit", command= actionSubmit)
btnSubmit.pack()
```

```
#Text
txtAddress = tk.Text(root, height=4, width=40)
txtAddress.pack()
```

```
#Frame
redFrame = tk.Frame(root, bg="red", height=200, width=400)
redFrame.pack()
```

The above code snippets show how to create and use various Tkinter widgets in a simple GUI application. It has a Label widget that shows static text, "This is a Label!" and it is packed into the window by using the pack() method. A "Login" button is added to the window, which when pressed calls the actionLogin function and prints a message to the console. An Entry widget is also included, allowing users to enter single-line text, along with a "Submit"

button to call the get() method in order to print the entry widget's contents. Apart from this, Text widget is also provided for allowing multiline texts such as address input and anything else that is extensive in terms. The pack() method is used consistently throughout for attaching widgets to the application window and organizing them vertically. The above code shows a perfect example of how Tkinter widgets can work together to make a GUI interactive and functional.

A simple application demonstrating the combined use of all widgets

Example 4.8 shows the simple application built using Python which showcases the usage of different widgets using the Tkinter module.

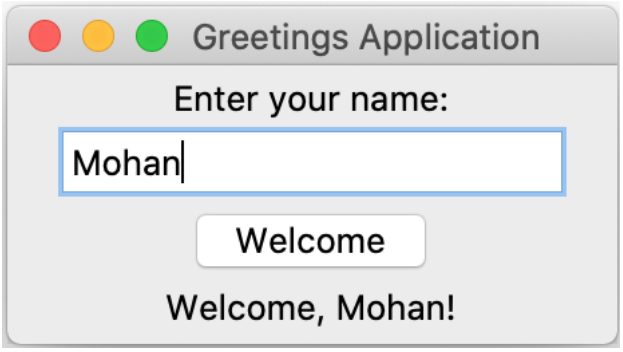
Example 4.8: Program to show combined usage of different widgets

```
#Program to show combined usage of different widgets

import tkinter as tk

def actionWelcome ():
    txtData = entryName.get()
    lblWelcome.config(text=f"Welcome, {txtData }!")

# Create the main window
root = tk.Tk()
root.title("Greetings Application")

# Add widgets
lblName = tk.Label(root, text="Enter your name:")
lblName.pack()

entryName = tk.Entry(root)
entryName.pack()

btnWelcome = tk.Button(root, text="Welcome", command=actionWelcome)
btnWelcome.pack()
```

```
lblWelcome = tk.Label(root, text="")
lblWelcome.pack()

# Run the application
root.mainloop()
```

The code above presents a demonstration showing the combined use of several Tkinter widgets to form a simple "Greetings Application." The main window is created with tk.Tk() and entitled "Greetings Application." This window contains a Label widget (lblName) that asks for the user's name, followed by an Entry widget (entryName) for typing the names. A Button widget (btnWelcome) is configured to call the actionWelcome function when clicked. This function issues the user input from the Entry widget and, thus, updates another Label widget (lblWelcome) to contain a personalized greeting. The pack() method is used to align all widgets vertically within the window. The application executes in a loop waiting for user interaction until the window is closed. When this program is executed you will get a sample GUI as can be seen in Figure 4.4.

Figure 4.4: Out of Example 4.8



Tkinter allows building GUI interfaces to be quite easy with its wide range of widgets and tools aiding user-friendly applications. Mixing the Python event-driven model with Tkinter and layout management provides developers with building dynamic, interactive applications that could be used in a host of real-life cases.

Note:

- **Tkinter is Python's standard library for creating graphical user interfaces (GUIs), allowing users to interact with applications visually.**
- **Tkinter is cross-platform, meaning GUIs built with it will work on Windows, macOS, and Linux without modification.**
- **The Tk() class is used to create the main application window, which serves as the container for all widgets.**
- **The mainloop() method starts the event loop, keeping the application responsive to user actions like clicks or key presses.**
- **Widgets like Labels, Buttons, Entry boxes, and Text areas are the building blocks of a Tkinter GUI.**

Check Your Progress-2

a) The method _____ is responsible for keeping the Tkinter window responsive to user actions.

b) The _____ widget is used to display static text or images in a Tkinter application.

c) For making a clickable button in Tkinter, use the _____ widget.

d) The _____ widget is used to accept multiple lines of text from a user or display long paragraphs.

e) A function assigned to a button in Tkinter, which gets executed upon a click, is called a _____.

f) The _____ widget in Tkinter is useful for collecting single-line user input, such as names or email addresses.

g) In Tkinter, the method _____ positions the widget in the application window.

4.4 LET US SUM UP

This unit discusses the concepts of file handling and GUI development using Tkinter in Python. You are acquainted with file handling's concept and different modes for interacting with files, various operations associated with file handling like reading, writing, appending, and handling binary files were discussed. We

also looked at how to build GUIs with Tkinter. We have discussed the anatomy of a Tkinter application, widely used widgets, like Label, Button, Entry, and so forth.

4.5 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

1-a True
1-b False
1- c True
1-d True
1-e False
1-f False
2-a mainloop()
2-b Label
2-c Button
2-d Text
2-e command
2-f Entry
2-g pack()

4.6 ASSIGNMENTS

1. Explain the concepts of file handling, explaining its importance in Python programming.
2. Write the objective and use of the open() function and its modes ('r', 'w', 'a', etc.).
3. Differentiate between text files and binary files with examples.
4. Explain the fundamental components of a Tkinter application and their roles.
5. Explain the purpose of the following Tkinter widgets: Label, Button, Entry, and Canvas.
6. Write a Python program implementing the below functionalities

- Write a program to count the number of words in a file named words.txt.
- Create a program to copy the contents of a file named source.txt into another file named destination.txt.
- Develop a program to read a binary file and display its content in hexadecimal format.
- Build a GUI that displays the content of a file in a Label widget after selecting the file through a file dialog.
- Create a calculator GUI that performs basic arithmetic operations using Entry widgets for input and Buttons for operations.