PG CA S4-03 (DSC)

TWO YEAR MASTER'S DEGREE COURSE

(CBCS in Accordance with NEP 2020)

SEMESTER-IV

PROGRAMMING IN PYTHON

MSc IT

Hiranya Chandra Bhuyan School of Science and Technology

KRISHNA KANTA HANDIQUI STATE OPEN UNIVERSITY

Subject Experts

Prof. Anjana Kakoti Mahanta, Gauhati University

Prof. Hiren Kumar Deva Sarma, Gauhati University

Dr. Lipi B Mahanta, IASST, Guwahati

Dr. Sanjib Kr. Kalita, Gauhati University

Course Co-ordinator: Dr. Ridip Dev Choudhury, KKHSOU

SLM Preparation Team

UNIT	CONTRIBUTORS
1, 10, 11, 12, 15	Dr. Sruti Sruba Bharali, KKHSOU
2, 3, 8, 13	Dr. Tapashi Kashyap Das, KKHSOU
4, 5, 6, 7, 9, 14, 16	Dr. Ridip Dev Choudhury, KKHSOU

Editorial Team

Content : **Dr. Utpal Barman, Assam Skill University**

Structure, Format & Graphics: **Dr. Sruti Sruba Bharali, KKHSOU**

April, 2025

This Self Learning Material (SLM) of the Krishna Kanta Handiqui State University is made available under a Creative Commons Attribution-Non Commercial-ShareAlike4.0 License (International): http.//creativecommons.org/licenses/by-nc-sa/4.0.

Printed and published by Registrar on behalf of the Krishna Kanta Handiqui State Open University.

Head Office : Patgaon, Rani Gate, Guwahati-781 017;

Sub-Head Office : Resham Nagar, Khanapara, Guwahati-22; Web: www.kkhsou.ac.in

The University acknowledges with thanks the financial support provided by the **Distance Education Bureau, UGC**, for the preparation of this study material.

MScIT

PROGRAMMING IN PYTHON

CONTENTS

Unit 1: Introduction to Python **7-17**

Introduction to Python and its features, Installation of Python and setup of development environment, Running a simple program in Python

Unit 2: Variables and Data Types **18-31**

Variables and their types in Python, Data types in Python, including integers, floats, strings, and Booleans; Working with variables and data types in Python

Unit 3: Operators and Expressions **32-55**

Arithmetic, comparison, and logical operators in Python, Expression evaluation in Python, Order of precedence in Python expressions

Unit 4: Control Structures **56-68**

Conditional statements in Python, including if-else and nested if-else statements; Looping statements in Python, including for and while loops; Control flow statements in Python, including break and continue statements

Unit 5: Functions **69-80**

Introduction to functions in Python, Defining and calling functions in Python, Parameters and return values in Python functions

Unit 6: Lists in Python **81-92**

Introduction to lists in Python, Creating and manipulating lists in Python, Slicing and indexing in Python lists

Unit 7: Tuples **93-102**

Introduction to tuples in Python, Creating and manipulating tuples in Python, Difference between lists and tuples in Python

Unit 8: Dictionaries **103-122**

Introduction to dictionaries in Python, Creating and manipulating dictionaries in Python Using dictionaries to solve problems

Unit 9: Sets **123-135**

Introduction to sets in Python, Creating and manipulating sets in Python, Using sets to solve problems

Unit 10: File Handling **136-147**

Introduction to file handling in Python, Reading and writing text files in Python, Handling errors while working with files

Unit 11: Object-Oriented Programming **148-164**

Introduction to object-oriented programming in Python, Defining classes and objects in Python, Encapsulation, inheritance, and polymorphism in Python

Unit 12: Exception Handling **165-177**

Introduction to exception handling in Python, Types of exceptions in Python, Handling exceptions in Python code

Unit 13: Regular Expressions **178-203**

Introduction to regular expressions in Python, Defining regular expressions in Python, Using regular expressions to search and replace text in Python

Unit 14: Modules and Packages **204-225**

Introduction to modules and packages in Python, Creating and importing modules in Python, Using third-party packages in Python

Unit 15: GUI Programming **226-247**

Introduction to GUI programming in Python, Using Tkinter library for GUI programming in Python, Creating and running GUI programs in Python

Unit 16: Advanced Topics **248-263**

Advanced topics in Python, including multi-threading, networking, and web programming, Using libraries and frameworks for advanced Python development, Best practices for Python programming and development.

COURSE INTRODUCTION

Python is a versatile, high-level programming language that is widely recognized for its simplicity, readability, and power. It is extensively used in web development, data science, artificial intelligence, automation, and software engineering. This course, Programming in Python, is designed to guide learners from the basics of programming to advanced Python concepts, providing a strong foundation to build real-world applications.

The course begins with an introduction to Python, covering its features, installation, and setup of the development environment. Students will learn how to write and run simple Python programs, gradually progressing to core concepts such as variables, data types, operators, expressions, and control structures. These fundamentals enable learners to write clear, efficient, and logical programs.

Learners will then explore essential Python data structures, including lists, tuples, dictionaries, and sets, and gain hands-on experience with functions, file handling, and object-oriented programming principles such as classes, objects, inheritance, and polymorphism. Exception handling and regular expressions are introduced to help students manage errors and perform pattern-based text processing.

The course also emphasizes practical programming skills through modular programming and GUI development using Tkinter, allowing students to create interactive desktop applications. Advanced topics such as multi-threading, networking, and web programming are covered to equip learners with the knowledge required to build high-performance, scalable applications. Students will also learn to use popular Python libraries and frameworks, enhancing productivity and simplifying complex programming tasks.

By the end of the course, students will be confident in designing, writing, testing, and debugging Python programs. They will be able to develop desktop and web applications, handle data efficiently, and apply Python programming skills to solve real-world problems. This course is ideal for beginners as well as those looking to strengthen their programming knowledge and apply Python in professional or academic settings.

UNIT 1: INTRODUCTION TO PYTHON

UNIT STRUCTURE

1.1 Learning Objectives

1.2 Introduction

1.3 Python

1.4 Features of Python

1.5 Installation of Python

1.6 Running Programs in Python

1.7 Let Us Sum Up

1.8 Answers to Check Your Progress

1.9 Further Reading

1.10 Model Questions

1.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- describe Python programming language
- list the various features of Python
- install Python in Windows machine
- run simple programs in Python using Python's command line window
- run simple programs in Python using command prompt
- run simple programs in Python using Python's IDLE

1.2 INTRODUCTION

In this unit, we will learn about Python programming language. Python is widely used across multiple domains and plays a key role in numerous applications, including data science, machine learning, web development, and more. In this unit, we will also discuss the various features of Python programming language that makes it so popular and widely used. We will also explore the steps to install Python in a Windows machine. At the end of this unit, we will see the various options available to run simple Python statements and programs. In the next unit, we will discuss about the vari-

ables and data types in Python programming language.

1.3 PYTHON

Python is one of the most popular programming languages in the world. Python was developed by Guido van Rossum in the year 1991. Python is widely regarded as an excellent choice for scripting and rapid application development due to several of its key features. Its clean, readable, and elegant syntax allows developers to write code quickly and efficiently, while its dynamic typing system enables flexible and adaptive programming without the need for strict variable declarations. Furthermore, as an interpreted language, Python executes code line by line, which not only simplifies debugging but also accelerates the development cycle. These characteristics, combined with Python's cross-platform compatibility, make it an ideal programming language for a wide range of applications across various domains and operating systems.

1.4 FEATURES OF PYTHON

Let us look at the various features of Python:

- Simple: Python is a simple programming language which makes it feel like using a natural language. This make it easy for beginners to write code using Python.
- Open Source: Python software can be freely downloaded from the website "www.python.org.". There is no need to pay for Python software.
- Interpreted Language: The code in Python is interpreted line by line which makes debugging easier. This also makes testing of small code blocks quicker and easier.
- Dynamically Typed: We do not need to declare the types of the variable in Python. The type of the variable is determined at runtime. This adds flexibility as the same variable may be used for multiple purposes by assigning it values of different data types.
- High Level Language: Python is a high level language which helps programmers to focus on solving the problems instead of dealing with low level details such as memory management.
- Platform Independent: The Python code when complied generates a

byte code. The byte code can be run on any machine by using a Python Virtual Machine (PVM). So, the same Python code can be run on Windows, Linux and Mac OS systems. This makes Python a platform independent language.

- Extensive Standard Library: Python has a rich set of built-in modules and libraries that support various tasks like file handling, regular expressions etc.
- Procedure and Object-Oriented: Python has features of both procedure and object-oriented programming paradigms. It supports procedure features like functions as well as object-oriented features like classes and objects.
- Applications in Various Domains: Python has a host of popular libraries like NumPy, Pandas, Django, SciPy which allow Python to be used in various applications in various domains like data science, web development, machine learning etc.

1.5 INSTALLATION OF PYTHON

The latest version of Python is 3.13.4. Depending on the configuration of the computer system, we can choose to download either the 32-bit version or the 64-bit version. Since our configuration is 64-bit, so we will choose to download the executable file for 64-bit Python installer.

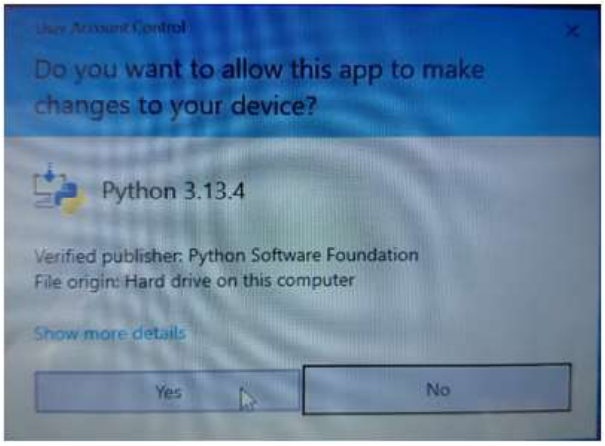
Installing Python for Windows:

The steps for installing Python on Windows 10 machine as listed below:

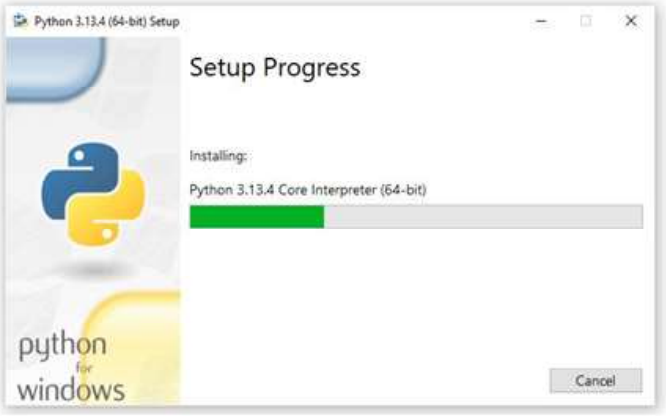
1. Open the website "www.python.org.".
2. Download the application file "python-3.13.4-amd64.exe" from the website.
3. Run the installer file. Select the options for admin privileges when executing py



1. Select the “Install Now” option.
2. Select “Yes” to allow this app to make changes to your device.

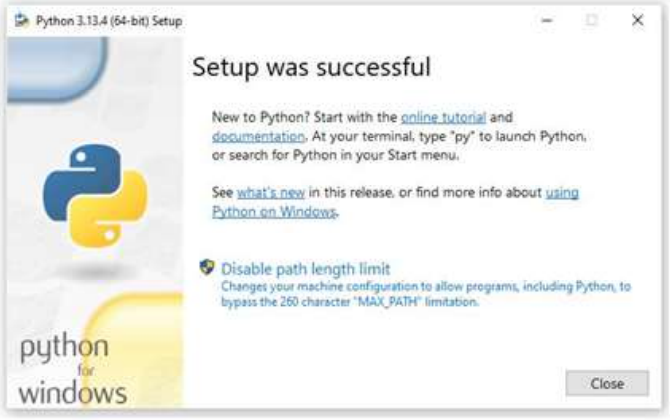


1. The Setup Progress bar will show the progress of the installation process.



Once Installation is complete, we can see the "Setup was successful" message.

7. We click on the "Close" button to finish the installation process.

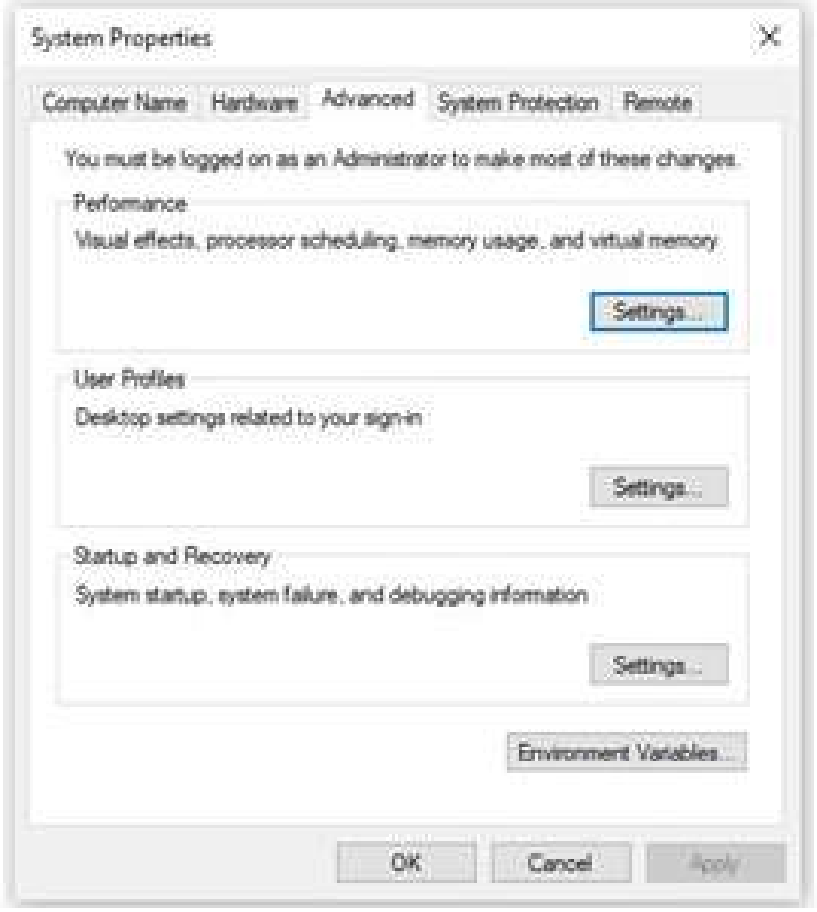


Python 3.13.4 has been installed in the computer system now.

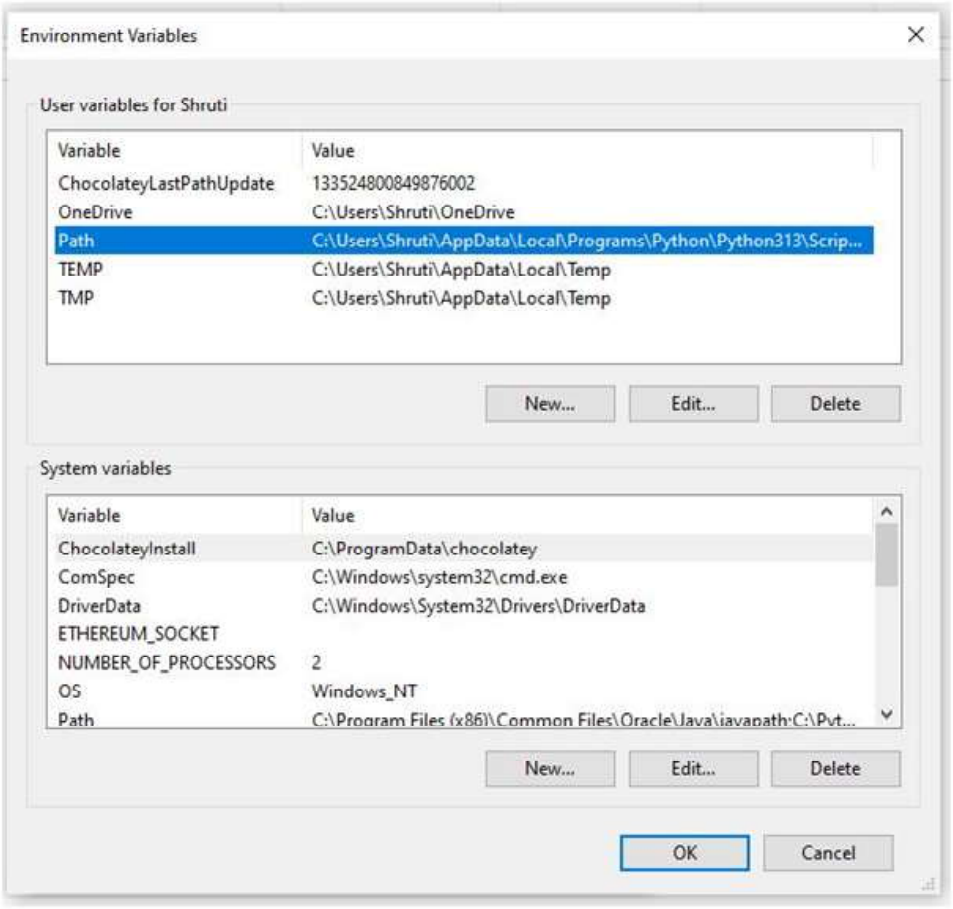
Setting the PATH to Python:

During the installation process of Python, the path is automatically set to Python. We can check whether the path to the Python directory is added in the OS's environment variables. The steps to view the path to the Python directory are:

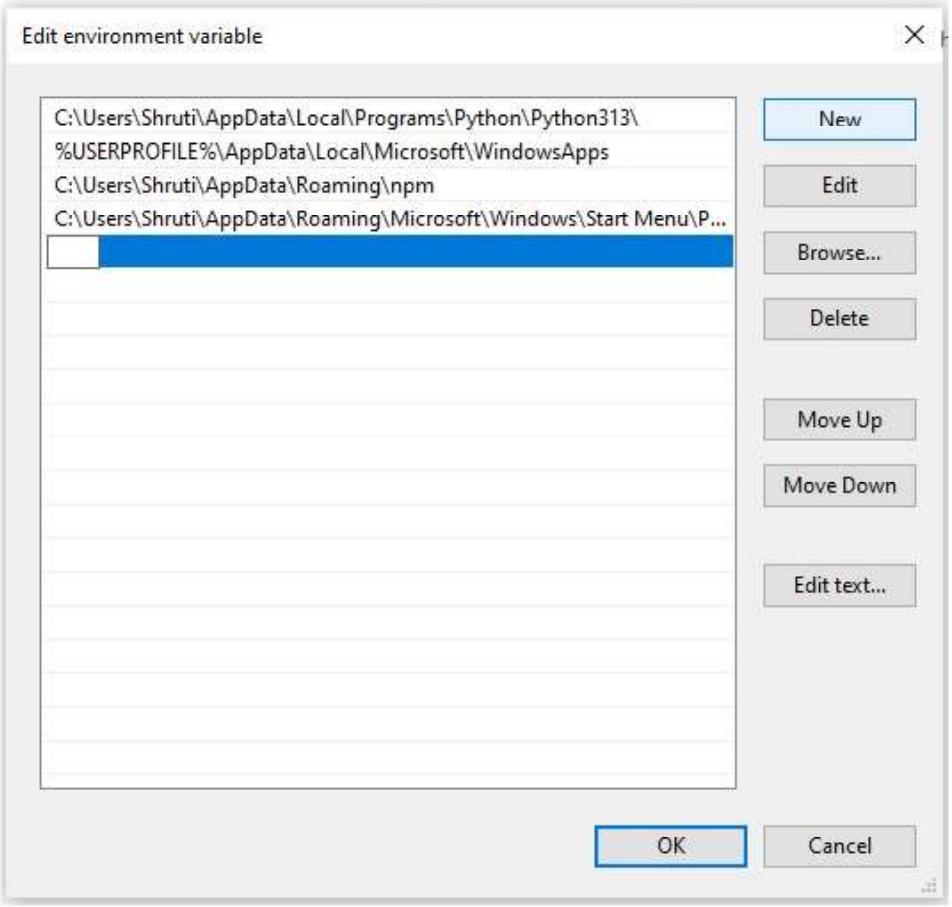
1. We right click on "This PC" in Windows 10 and click on the "Properties" option. This will open the "Settings" menu.
2. We now click on the "Advanced System Settings" option from the Settings menu. This will display a "System Properties" dialog box.



3. We click on the "Environment Variables" button in the dialog box. This will display the User variables and System variables.
4. We select the "PATH" variable.



5. We select on "Edit" button in the User variables for the "Path" variable. This will open the "Edit Environment Variable" dialog box. This will display the directories involved in the Path. To get the location or path to Python file that is to be added, we go the Python executable application in the Start Menu. We right click on this file and select "Open file location" to go the exact location where the Python files are located. We can copy the path of this location and add the path if they not already added, by clicking the "New" option and writing the path in the place where the cursor is positioned. For example:
   - C:\Users\.....\Python 313\ (for the actual Python install folder)
   - C:\Users\.....\Python 313\Scripts\ (for pip and other tools)



6. After adding, we click on "OK" to save the settings.

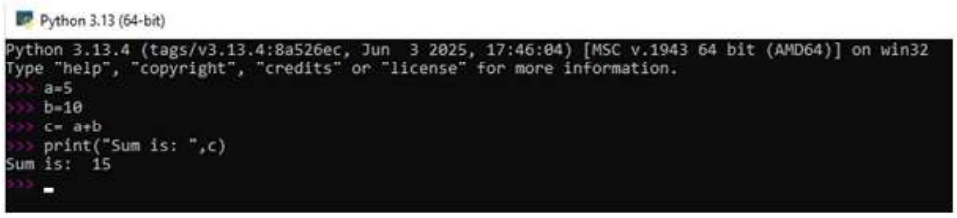
1.6 RUNNING SIMPLE PROGRAMS IN PYTHON

There are multiple ways to run programs in Python. Let us look at two of the simplest ways.

- Using Python's Command Line Window:

To use Python's Command Prompt Window, we can simply click on the Python 3.13 (64 bit) application available in the Start Menu. This will open the Python's command line window. We can see the Python prompt in the window which is "<<<". We can type our statements and program after the Python prompt.

Let us try to execute a simple Python program to add two numbers.



Here in this example, we can see that we have taken two variables "a" and "b" and initialized them with values 5 and 10 respectively. We have then taken a third variable "c to which we assigned the sum of the two variables "a" and "b". Finally we have used the print() function to display the sum value.

- Using Command Prompt:

The same code that has been written as statements in the above example can be written together in a program. We can write these statements in a text file and save it by giving a relatable filename with the extension ".py". For example, we have save the statements as "sum.py". The syntax to run to a python file is:

python filename.py

So, run the file "sum.py" we write the command:

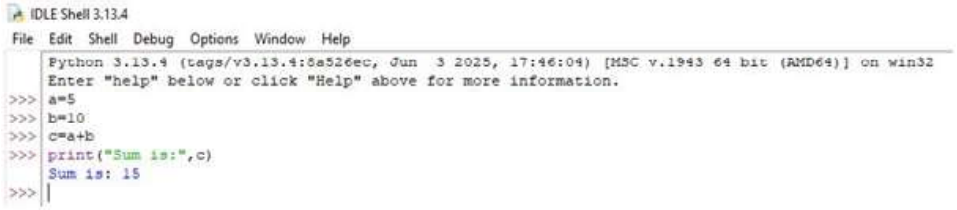
python sum.py

The output can be seen in the figure.

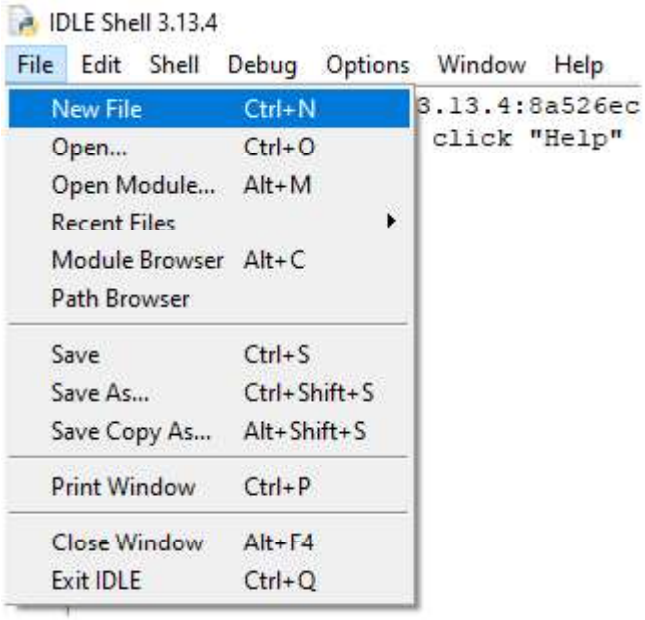


- Using Python's IDLE Graphics Window:

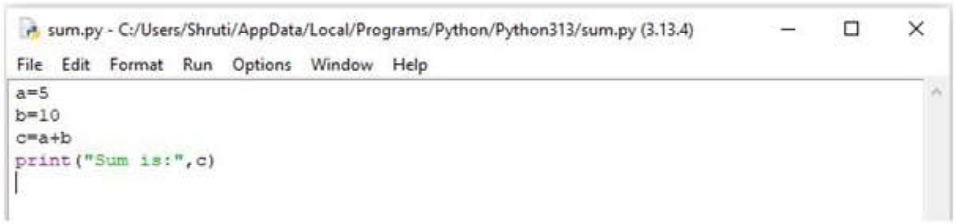
We can also use the Python's IDLE (Integrated Development Environment) which provides a graphical interface. We can use IDLE to write statements or whole programs and also to run our programs. We can open an IDLE window by clicking on the Python's IDLE application. We can then write our statements there just like in case of Python's command line window.



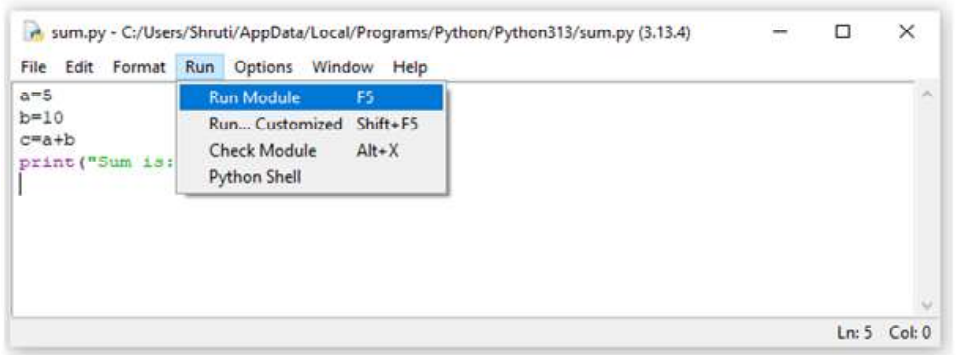
We can also create new programs and run those programs in the IDLE. To do this, we first open a new file by selecting the "File" -> "New File" option available in the Menu Bar.



We can now write our program in the new file and save it.



We can run the program using the command prompt. Another option is to run the file using the "Run"-> "Run Module" option available in the Menu Bar for the program.



The result of the program will be displayed in the IDLE window.

```
>>> ========== RESTART: C:/Users/Shruti/AppData/Local/Programs/Python/Python313/sum.py ==========
Sum is: 15
>>>
```



CHECK YOUR PROGRESS

Q1. What are two features of Python?

Q2. What is the command to run a Python file?



1.7 LET US SUM UP

- Python was developed by Guido van Rossum in the year 1991.
- Python software can be freely downloaded from the website "www.python.org.".
- We do not need to declare the types of the variable in Python.
- The Python code when complied generates a byte code. The byte code can be run on any machine by using a Python Virtual Machine (PVM).
- Python has features of both procedure and object-oriented programming paradigms.
- Python has a host of popular libraries like NumPy, Pandas, Django, SciPy etc.
- The syntax to run to a python file is: python filename.py
- We can also use the Python's IDLE (Integrated Development Environment) which provides a graphical interface to write statements, whole programs and also to run our programs.



1.8 ANSWERS TO CHECK YOUR PROGRESS

Ans to Q1: Two features of Python are it is platform independent and open source.

Ans to Q2: The command to run a Python file is: python filename.py



1.9 FURTHER READING

1) Rao, D. R. N. (2018). Core python programming.
2) Kanetkar, Y., & Kanetkar, A. (2020). Let Us Python: Python Is Future, Embrace It Fast (Vol. 2). BPB Publications.
3) Chun, W. (2001). Core python programming (Vol. 1). Prentice Hall Professional.
4) https://docs.python.org/3/contents.html



1.10 MODEL QUESTIONS

Q1. What makes Python a popular programming language?
Q2. Describe the various features of Python.
Q3. What are the steps to install Python in a Windows machine?
Q4. Describe the different ways in which we can run a simple Python program?
Q5. What does platform independent mean in context with Python language?

****

UNIT 2: VARIABLES AND DATA TYPES

UNIT STRUCTURE

2.1 Learning Objectives

2.2 Introduction

2.3 Understanding Variables in Python

2.4 Datatypes in Python

2.4.1 Basic data types in Python

2.5 Working with Variables and Data Types

2.6 Let Us Sum Up

2.7 Answers to Check Your Progress

2.8 Further Reading

2.9 Model Questions

2.1 LEARNING OBJECTIVES

After going through this tutorial, you will be able to:

- learn what variables are in Python.
- identify and use different basic data types in Python.
- work with variable assignment, type checking, and type casting.
- represent binary, octal, and hexadecimal numbers.

2.2 INTRODUCTION

In the previous unit, we were introduced to Python as a programming language, its key features, and how to set up the Python development environment. We also learned how to write and run a simple Python program.

This unit will help us understand how to store and manage data in Python using variables. We will learn about different datatypes such as integers, floats, complex, strings, and Booleans, and how they are used in programs. Understanding variables and datatypes is essential because they form the basis for writing logical and functional code.

2.3 UNDERSTANDING VARIABLES IN PYTHON

In Python, a variable can be thought of as a labeled storage location in the computer's memory where data is kept temporarily. The name assigned to this storage location is known as an identifier. Identifiers are used to name variables, functions, objects, classes, or other elements within a program. One of the powerful features of Python is its ability to automatically recognize the type of data a variable hold. When we assign a value to a variable, the Python interpreter internally determines the data type and allocates the required amount of memory accordingly. This means we do not need to explicitly declare the data type of a variable, unlike languages such as C or Java where type declaration is mandatory.

For example, in C, we declare an integer type variable as:

```
int a;      // declare a as integer type variable
a = 25;     //  25 is stored in the variable a
```

But, in Python, we can simply write as :

```
a =  25
```

In Python, when we store a string (a sequence of characters) in a variable like 'a', the language automatically treats a as a string-type variable and allocates memory accordingly to store that string.

In traditional programming languages like C or Java, variables are closely linked to memory locations. We can think of a variable as a labelled box that stores a value. For instance, if we write:

a = 5;

a memory location is reserved with the name 'a', and the value '5' is stored in it. We can imagine this as a physical box labelled a containing the number 5.

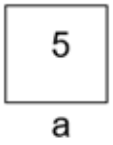


Figure 2.1: Variable 'a' storing value 5

If we later change the value of 'a' to '10', the content inside the box is simply updated:

a = 10;

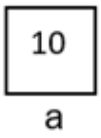


Figure 2.2: Updated value of variable 'a'

Now, if we assign the value of one variable to another:
int b = a;
a new memory space is created for 'b', and the value stored in 'a' (which is 10) is copied into 'b'.

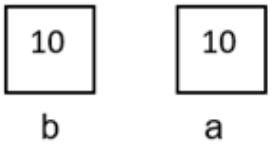


Figure 2.3: Variable 'b' copying value from 'a'

This is the typical behaviour in many programming languages. But Python works a bit differently.
In Python, a variable is more like a name or label that is attached to a value. When we write:
a = 10
Python first creates the value 10 as an object in memory and then attaches the name 'a' to that value.



Figure 2.4: Python variable 'a' referencing value 10

Now, if we change the value of 'a':
a = 15
Python doesn't update the value inside the same memory location. Instead, it creates a new object '15' and reassigns 'a' to point to this new object. The previous value '10' becomes unreferenced and is automatically cleared from memory by Python's garbage collector.



Figure 2.5: 'a' now points to 15

If we now assign the value of 'a; to another variable 'b':

b = a

Python creates another label 'b' that also refers to the same object '15'. Both 'a' and 'b' point to the same value, not copies of the value.

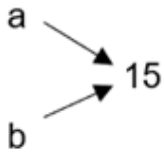


Figure 2.6: 'a' and 'b' both refer to the same value

Let's understand how Python handles variables a bit differently than some other programming languages. In Python, we use names (or tags) to refer to values. For example, in Figure 2.3, two separate memory spaces are used to store two values. But in Figure 2.6, the same memory is shared by two names. This shows that Python uses memory more efficiently. In Python, values are treated as objects, and the variables are just labels or references pointing to those objects.

2.4 DATA TYPES IN PYTHON

A datatype represents the type of data stored into a variable and memory. The datatypes which are already available in any language are called built-in datatypes. The datatypes which can be created by programmers are called user-defined datatypes.

Python is a dynamically typed language. This means we don't need to mention the data type of a variable when we create it. Python automatically figures out the type of data based on the value we assign to the variable.

Example:

```
x = 10          # Python understands x is an integer
y = 3.14        # Python understands y is a float
```

```
name = "Kaberi"        # Python understands name is a string
```

In each case, Python decides the data type on its own, without the need for us to declare it. Data types are important because they help Python use the right amount of memory, decide what operations can be done on the data, and avoid errors in the program.

Python supports three categories of datatypes:

- Basic types: int, float, complex, bool, bytes
- Container types: list, tuple, sets, dict
- User-defined types- class

In this unit, we will learn about the basic data types with the help of examples. The container types and user-defined types will be covered in the upcoming units of this course.

2.4.1 Basic Data Types in Python

Python comes with several basic datatypes that help us store and work with different kinds of data. These data types are grouped into different categories. One of the most important categories is the numeric types, which include:

- Integer (int)
- Floating-point (float)
- Complex (complex)

Integer (int) Type:

An integer is a whole number- it does not have a decimal point. Positive numbers, negative numbers, and even zero are considered integers.

Examples of integers:

```
a = 50
b = -100
c = 0
d = 987654321
```

In Python, there is no fixed size limit for integers. This means Python can handle very large integer values as needed, thanks to its arbitrary-precision integer system.

big_number = 12345678910111213141516171819 20

Python stores such large values efficiently and allows mathematical operations on them just like with smaller numbers.

Floating-Point (float) Type

A float is a number that includes a decimal point. These numbers can also be written in scientific notation, using e or E to represent powers of 10.

Examples of floats:

x = 55.45

y = -4.3596

z = 0.001

a = 2.5e3 # Represents 2.5 × 10^3 = 2500.0

Internally, floats are stored using the IEEE 754 64-bit double-precision format. Because of this binary representation, some decimal numbers cannot be stored exactly.

Python floats can represent numbers up to approximately 1.8 × 10^3??.

Complex (complex) Type: A complex number has two parts:

- A real part
- An imaginary part (indicated by j or J)

The general form is: a + bj

Here, 'a' is the real part and 'b' is the imaginary part.

Examples of complex numbers:

c1 = 4 + 5j

c2 = 3 - 2j

c3 = 0.4 + 8.5J

c4 = -2 - 4.5J

Both parts of a complex number can be integers or floats.

Example:

```
# Python program to add two complex numbers
c1 = 4 + 5j
c2 = 3 - 2j
c3 = c1 + c2
print("Sum = ", c3)
```

Output:

Sum = (7+3j)

This makes complex number handling in Python very straightforward, especially for applications in mathematics, engineering, and scientific computing.

Boolean Type:

In Python, the Boolean type is used to represent two possible values: True or False (both start with a capital letter). The bool datatype in Python represents Boolean values. Booleans are commonly used in conditions and decision-making in programs.

Example:

```
is_passed = True
```

This means the variable is_passed holds the Boolean value True.

Example:

```
# Example of bool datatype in Python
is_raining = True
is_sunny = False
print("Is it raining?", is_raining)
print("Is it sunny?", is_sunny)
# Using bool in a condition
```

```
if is_raining:
    print("Take an umbrella.")
else:
    print("No need for an umbrella.")
```

Output:

```
Is it raining? True
Is it sunny? False
Take an umbrella.
```

This example shows how Boolean values can be stored in variables and used in conditional statements.

The String (str) data type

A string in Python is a sequence of characters used to store and manipulate text data. Strings are one of the most commonly used datatypes in programming because most of the information we encounter daily - like names, addresses, identification numbers, and descriptive text is in string form.

In Python, str represent string datatype. It provides operations and methods to handle text efficiently. Moreover, in Python, there is no separate datatype for a single character. Even a single character like 'A' or '9' is considered a string of length one. In Python, strings can be defined by enclosing characters within

- Single quotes: 'Hello'
- Double quotes: "Hello"
- Triple quotes: '''Hello''' or """Hello"""

Triple quotes are particularly useful for defining multi-line strings or for including both single and double quotes in the string without the need for escape characters.

Example:

```
s1 = 'India'    # string with single quote
```

```
s2 = "My country name is India."
            # String with double quote
s3 = '''This is
a multi-line
string.'''
```

Representing Binary, Octal and Hexadecimal Numbers in Python :
In Python, you can represent binary, octal, and hexadecimal numbers using specific prefixes:

Number System	Prefix	Example	Base
Binary	0b	0b1010	Base 2
Octal	0o	0o12	Base 8
Hexadecimal	0x	0xA	Base 16

Example :

```
# Binary number
binary_num = 0b1010       # = 10 in decimal
print("Binary:", binary_num)

# Octal number
octal_num = 0o12          # = 10 in decimal
print("Octal:", octal_num)

# Hexadecimal number
hex_num = 0xA             # = 10 in decimal
print("Hexadecimal:", hex_num)
```

Output:

```
Binary:10
Octal:10
Hexadecimal:10
```

Example: Converting Decimal to Other Bases:

```
num = 25
print("Binary of 25:", bin(num))       # 0b11001
print("Octal of 25:", oct(num)) # 0o31
print("Hexadecimal of 25:", hex(num))  # 0x19
```

2.5 WORKING WITH VARIABLES AND DATA TYPES

Simple Variable Assignment: In Python, a simple variable assignment means assigning a value to a single variable using the = operator.

Example:

```
x = 10  # x is assigned the integer 10
name = "Niyor"    # name is assigned the string "Niyor"
price = 350.75  # price is assigned the float 350.75
```

Multiple Variable Assignment: Python allows assigning values to multiple variables at the same time in a single line. Multiple Assignment makes the code cleaner and shorter.

Assigning different values to different variables:

```
a, b, c = 1, 2, 3
```

Assigning the same value to multiple variables:

```
x = y = z = 100
```

Here, x, y, and z all get the value 100

```
#Multiple assignment
a=10;b=35.50;name="Ruman"
print(a,b,name)   # Output: 10 35.5 Ruman
```

Here, the semicolon acts as separator.

Type Checking:

In Python, it is possible to check the type of a variable using the built-in function type(). For example,

```
x = 25
print(type(x))      # Output: <class 'int'>
a='KKHSOU'
```

```
print(type(a))    # Output: <class 'str'>
print(type(3.14))   # Output: <class 'float'>
```

Type Conversion (Type Casting):

Sometimes, we may need to convert one type to another. Python provides functions for this:

- int() - Convert to integer
- float() - Convert to float
- str() - Convert to string
- list() - Convert to list
- tuple() - Convert to tuple
- set() - Convert to set

Example:

```
x = "123"
y = int(x)
print(y + 10)  # Output: 133
```

Example: Python program to assign three values to three variables in one line and display them.

```
a, b, c = 10, 20.5, "Python"
print(a)
print(b)
print(c)
```



CHECK YOUR PROGRESS

Q1) What will be the output of type(10.0)?

Q2) Convert the string "25" to an integer and add 10.

Q3) Give the correct answer for the following MCQs:

i) What is the type of value assigned by x = 8.5?

A) int

B) float

C) complex

D) bool

ii) Which of the following is a correct multiple variable assignment?

A) a, b = 1; 2

B) a b c = 1 2 3

C) a, b, c = 1, 2, 3

D) a = b = 1, 2

iii) What function is used to check the type of a variable in Python?

A) datatype()

B) typeof()

C) checktype()

D) type()

iv) Which prefix is used to represent a binary number in Python?

A) 0x

B) 0o

C) 0b

D) 0d

What is the output of int('5') + 10

A) '15'

B) Error

C) "5"

D) 15



2.6 LET US SUM UP

Understanding data types enables us to store, manage, and process different kinds of data effectively. Python's dynamic and flexible approach to data types makes programming easier for beginners while offering powerful tools for advanced users. The key points covered in this chapter are as follows:

- Python variables are labels attached to values.
- Python automatically assigns data types during variable assignment (dynamic typing).

- Major basic data types in Python include: int, float, complex, bool, and str.
  - ❖ **int:** Whole numbers, positive or negative, of unlimited size.
  - ❖ **float:** Numbers with decimal points or in scientific notation.
  - ❖ **complex:** Numbers with real and imaginary parts (a + bj format).
  - ❖ **bool:** Represents True or False.
  - ❖ **str:** Sequence of characters, defined using single, double, or triple quotes.
- Python allows simple and multiple variable assignments in one line.
- type() can be used to check the datatype of a variable.
- Type casting functions (int(), float(), str(), etc.) help in converting data types.
- Binary, octal, and hexadecimal numbers can be represented using 0b, 0o, and 0x prefixes.



2.7 ANSWERS TO CHECK YOUR PROGRESS

Ans.Q.No.1:

<class 'float'>

Explanation: Here, 10.0 is a floating-point number, so its type is float.

Ans.Q.No.2:

int("25") + 10 # This gives 35

Explanation: The string "25" is converted to an integer using int(), and then 10 is added.

Ans.Q.No.3:

i) B) float ii) C) a, b, c = 1, 2, 3 iii) D) type()

iv) C) 0b v) D) 15



2.8 FURTHER READING

- Kanetkar, Y., & Kanetkar, A. (2024). Let us Python (6th ed.). BPB Publications.

- Rao, N. (2024). Core Python programming: PyCharm and Jupyter. Dreamtech Press.
- Python Official Documentation https://docs.python.org/3/tutorial/



2.9 MODEL QUESTIONS

1. What are the different types of numeric data types in Python?
2. What is bool datatype with a simple program example.
3. How are complex numbers handled in Python?
4. Differentiate between int and float datatypes.
5. What is type casting? Give examples.
6. Discuss the concept of variables in Python. How is it different from other languages like C or Java?
7. Explain all basic datatypes in Python with suitable examples.
8. What are strings in Python? Describe different ways of creating strings.
9. Describe type checking and type conversion with sample codes.
10. Describe the representation of binary, octal, and hexadecimal numbers in Python with examples.
11. Create a Python program that reads a string input, converts it to an integer, and performs arithmetic operations.
12. Write a Python program to convert the string "50" to an integer, add 25 to it, and print the result.
13. Write a Python program that assigns different types of values to variables and prints their types.
14. Write a program to store a Boolean value and use it in an if condition to print a message.

*****

UNIT 3: OPERATORS AND EXPRESSIONS

UNIT STRUCTURE

3.1 Learning Objectives

3.2 Introduction

3.3 Understanding Operators in Python

- 3.3.1 Arithmetic Operators
- 3.3.2 Relational (Comparison) Operators
- 3.3.3 Logical (Boolean) Operators
- 3.3.4 Assignment Operators
- 3.3.5 Bitwise Operators
- 3.3.6 Membership Operators
- 3.3.7 Identity Operators

3.4 Operator Precedence and Associativity

3.5 Let Us Sum Up

3.6 Answers to Check Your Progress

3.7 Further Reading

3.8 Model Questions

3.1 LEARNING OBJECTIVES

After going through this tutorial, you will be able to:

- learn the concept of operators and their significance in Python programming.
- identify and apply various types of operators, including arithmetic, relational, logical, assignment, bitwise, membership, and identity operators.
- evaluate Python expressions accurately using appropriate operators.
- apply operator precedence and associativity rules to ensure correct expression evaluation.

3.2 INTRODUCTION

In the previous unit, we were introduced to variables and data types in Python. In this unit, we will learn about Python operators, which will help us

perform operations on data and control program logic. We will learn various types of operators, including arithmetic, relational, logical, assignment, bitwise, identity, and membership operators. This unit will also explain operator precedence, associativity, and how to evaluate expressions which are essential for writing accurate and efficient Python code.

3.3 UNDERSTANDING OPERATORS IN PYTHON

Operators are symbols in Python that carry out operations on variables and values. They play a crucial role in calculations, comparisons, and logical decision-making. Operands are the values that operators work on. For instance, in the expression a = 5 + 3, the + is an operator that adds the operands 5 and 3.

Python provides a variety of operators to perform different kinds of operations in your programs. These operators are essential for calculations, comparisons, assignments, and logical decision-making.

The main types of operators in Python include:

- Arithmetic Operators - For mathematical calculations (e.g., +, -, *, /, etc.)
- Relational (Comparison) Operators - For comparing values (e.g., ==, >, <)
- Logical (Boolean) Operators - For combining conditional statements (e.g., and, or, not)
- Assignment Operators - For assigning values to variables (e.g., =, +=, -=)
- Bitwise Operators - For manipulating binary data (e.g., &, |, <<)
- Identity Operators - For checking object identity (e.g., is, is not)
- Membership Operators - For testing membership in sequences (e.g., in, not in)

Each of these operators serves a specific purpose, and understanding them is key to writing efficient Python code. Let us discuss each of these categories of Python operators.

3.3.1 Arithmetic Operators

We will begin by discussing Arithmetic Operators. These operators are used to perform basic mathematical operations, including addition, subtraction, multiplication, division, modulus, exponentiation, and floor division. In the following table, the arithmetic operators in Python are provided:

Table 1: Python Arithmetic Operators

Operator	Description	Example
+	Addition operator	4 + 6 → 10
-	Subtraction operator	10 - 4 → 6
*	Multiplication operator	2 * 5 → 10
/	Division operator	3 / 2 → 1.5
//	Floor Division operator	9 // 4 → 2
%	Modulus operator	100 % 25 → 0
**	Exponentiation operator	5 ** 2 → 25

In Python, arithmetic operators are used to perform basic numeric calculations. An overview of the common arithmetic operators is given here:

- **Addition operator (+) :** The addition operator adds two numbers. It is denoted by the plus symbol (+).

Example: 5 + 3 results in 8.

- **Subtraction operator (-):** The subtraction operator subtracts the right operand from the left operand. It is denoted by the minus symbol (-). Example: 5 - 3 results in 2.
- **Multiplication operator (*):** The multiplication operator multiplies two numbers. It is denoted by the asterisk symbol (*).

Example: 5 * 3 results in 15.

- **Division operator (/):** The division operator divides the left operand by the right operand, returning a floating-point result. It is denoted by the forward slash (/).

Example: 3 / 2 results in 1.5.

- **Floor Division operator (//):** The floor division operator divides two numbers and rounds the result down to the nearest integer. The floor division // operator returns only the integer part of the division. It is denoted by the double backward slash (//).

Example: For a = 9 and b = 4, a // b results in 2.

- **Modulus operator (%):** The modulus operator returns the remainder of the division of the left operand by the right operand. It is denoted by the percentage symbol (%).

Example: For a = 100 and b = 25, a % b results in 0.

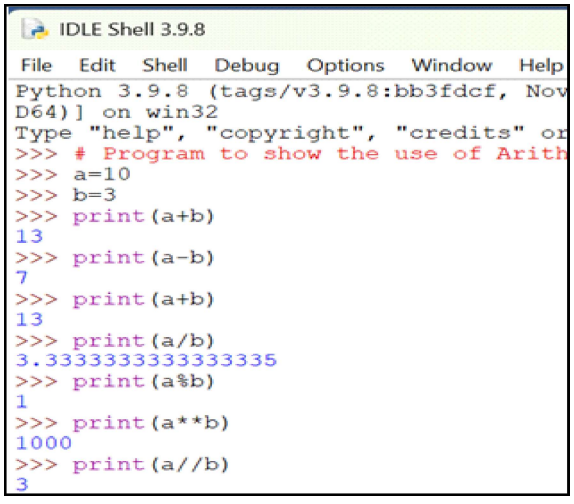
- **Exponentiation operator ():** ** The exponentiation operator raises the left operand to the power of the right operand. It is denoted by the double asterisk (**).

Example: 2 ** 3 results in 8.

These operators enable efficient numeric calculations within Python programs. To demonstrate the use of arithmetic operators, let's write a simple Python program in the Python command line. Upon launching the Python command line, we will see the prompt >>>, where we can enter and execute Python code. An example is given below:

Sample code 1: Use of Arithmetic operators in Python

```
>>> a=10
>>> b=3
>>> print(a+b)
13
>>> print(a-b)
7
>>> print(a+b)
13
>>> print(a/b)
```



```
3.3333333333333335
>>> print(a%b)
1
>>> print(a**b)
1000
>>> print(a//b)
3
>>>
```

Here, we define two variables, a and b. Using the various arithmetic operators, we perform addition, subtraction, multiplication, division, and more.

3.3.2 Relational (Comparison) Operators

Relational operators, also known as comparison operators, are used to compare two values and determine the relationship between them. These operators return a Boolean result-either True or False-based on the outcome of the comparison. They are fundamental in writing conditional statements and controlling the flow of decision-making in a program. For instance, the expression 10 > 5 evaluates to True, while 5 == 10 evaluates to False.

The commonly used relational operators in Python are summarized in the following table (Table 2):

Table 2: Python Relational Operators

Operator	Description	Example
>	Greater than	5 > 2 →True
<	Less than	5 < 2 → False
>=	Greater than or equal	5 >= 5 → True
<=	Less than or equal	4 <= 2 → False
==	Equal to	10 == 5 → False
!=	Not equal to	10! = 5 → True

For example, the following code compares the values of the variables a and b using various relational operators and displays the result of each comparison. The code is written in Notepad and saved with the filename "relational.py". The output shown is obtained by executing the program from the command prompt using the command: python relational.py

Sample code 2:

```
# Using Python relational operators in code
a, b = 12,8
print(a>b)# True as 12 > 8
print(a<b)# False
print(a>=b)# True
print(a<=b)# False
print(a==b)# False
print(a!=b)# True
```

```
C:\pythonscript>python relational.py
True
False
True
False
False
True
```

3.3.3 Logical (Boolean) Operators

Logical operators allow us to combine multiple conditions to create more complex decision-making structures in our code. These operators evaluate individual conditions (which return either True or False) and produce a final Boolean result based on their interaction.

A key aspect of Python's truth evaluation is:

- The value False is represented by 0 (zero)
- Any non-zero value is considered True

This behaviour becomes particularly important when working with different data types and conditional statements. Key logical operators in Python include:

and - Returns True only if both conditions are true

or - Returns True if at least one condition is true

not - Reverses the truth value of a single condition

These operators form the backbone of complex conditional logic in Python programming, enabling developers to create sophisticated decision trees

and program flows. The logical operators are listed in the following table (Table 3)

Table 3: Logical Operators in Python

Operator	Description	Example
and	Returns True if both conditions are true.	True **and** True →True True **and** False →False False **and** True →False False **and** False →False
or	Returns True if at least one condition is true.	True **or** True →True True **or** False →True False **or** True →True False **or** False →False
not	Reverses the truth value of a single condition	**not** True →False **not** False →True

Now, let us write some simple Python programs using logical operators and observe their outputs.

Sample code 3:

```
#Example of Logical operators
x=True
y=False
print(x and y)
print(x or y)
print(not x)
print(not y)
```

Output:



In this example (Sample code 3), we have two Boolean variables: x is set to True and y is set to False. Let's apply logical operators to these variables:

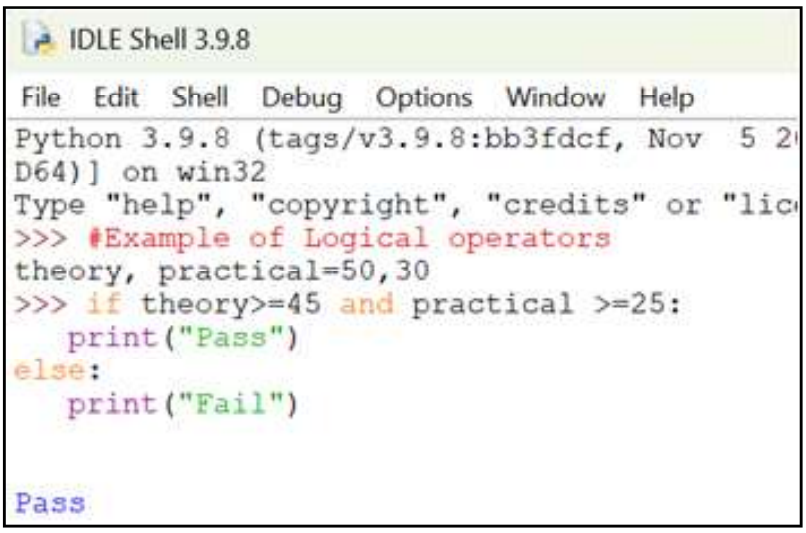
- x and y return False because both conditions must be True for the and operator to return True.
- x or y returns True because at least one condition is True, which satisfies the or operator.
- not x returns False because the not operator inverts the value of x.
- not y returns True because the not operator inverts the value of y.

Logical operators are essential in constructing conditional statements and loops, where decisions are based on multiple conditions. We will explore these concepts further in the upcoming lectures.

Now, let us consider another example involving logical operators. In this case, a student is considered to have passed only if they meet both the theory and practical marks criteria.

Sample code 4:

```
# use of Logical Operators in Python
theory, practical=50,30
if theory>=45 and practical >=25:
  print("Pass")
else:
  print("Fail")
```



In this example, the output is "Pass" because both the theory and practical marks exceed the required thresholds of 45 and 25, respectively. Logical operators such as and, or, and not are fundamental for controlling the flow of a program based on multiple conditions.

3.3.4 Assignment Operators

Assignment operators are used to assign values to variables in an efficient and concise manner. Assignment operators are presented in the following table:

Table 3: Assignment Operators in Python

Operators	Examples	Equivalent to
=	x = 10	10 is assigned to x
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
**=	x **= 5	x = x ** 5
//=	x //= 5	x = x // 5

The following examples demonstrates the various assignment operators available in Python.

Simple Assignment:

First, we assign the value 10 to variable a. When we print a, the output is 10.

```
a = 10
print("a =", a)  # Output: 10
```

Addition Assignment (+=): Next, we use the addition assignment operator.

A += 5 is equivalent to a = a + 5. The new value of a is 15.

```
A += 5
print("a += 5 ->", a)     # Output: 15
```

Subtraction Assignment (-=):

Now, we subtract 3 using a -= 3, which gives 12.

```
A -= 3
print("a -= 3 ->", a)  # Output: 12
```

Multiplication Assignment (*=):

Multiplying a by 2 using a *= 2 results in 24.

```
A *= 2
print("a *= 2 ->", a)     # Output: 24
```

Division Assignment (/=):

Dividing a by 4 using a /= 4 gives 6.0, because division in Python always results in a float.

```
A /= 4
print("a /= 4 ->", a)     # Output: 6.0
```

Floor Division Assignment (//=):

Floor division a //= 2 results in 3.0, as it rounds down the division result.

```
A //= 2
print("a //= 2 ->", a)  # Output: 3.0
```

Modulus Assignment (%=):

Applying a %= 2 gives 1.0, which is the remainder when 3 is divided by 2.

```
A %= 2
print("a %= 2 ->", a)  # Output: 1.0
```

Exponentiation Assignment (=):**

Now, we assign 3 to b. Using b **= 2 raises b to the power of 2, giving 9.

```
B = 3
b **= 2
print("b **= 2 ->", b)  # Output: 9
```

3.3.5 Bitwise Operators

Bitwise operators perform operations directly on the binary (bit-level) representations of integers.

The common bitwise operators in Python are:

- Bitwise AND (&): Performs an AND operation on each pair of corresponding bits.
- Bitwise OR (|): Performs an OR operation on each pair of corresponding bits.
- Bitwise XOR (^): Performs an exclusive OR; returns 1 if the bits are different, otherwise 0.
- Bitwise NOT (~): Inverts all the bits (computes the 1's complement).
- Left Shift (<<): Shifts bits to the left and fills the vacant positions with zeros.
- Right Shift (>>): Shifts bits to the right and discards the rightmost bits.

These operators are useful for low-level programming, cryptography, and when dealing with binary data.

Bitwise Left Shift and Right Shift Operations

Let us consider

```
x = 5  # Binary representation: 0101
```

Left Shift (x << 1)

0101 << 1 ? 1010 (binary) = 10 (decimal)

Right Shift (x >> 1)

0101 >> 1 ? 0010 (binary) = 2 (decimal)

Bitwise AND Operation

x = 5 # Binary: 0101

y = 3 # Binary: 0011

AND Operation (x & y)

0101

& 0011

0001 ? 1 (decimal)



CHECK YOUR PROGRESS

Q1. Give the correct answer:

i. Which operator is used to find the remainder in Python?

A. /

B. //

C. %

D. **

ii. What will be the output of True and False?

A. True

B. False

C. None

D. Error

iii. What will be the output of True or False?

A. True

B. False

C. None

D. Error

iv. What is the result of the expression 10 + 5 * 2?

A. 30

B. 20

C. 25

D. 15

3.3.6 Membership Operators

Understanding membership and identity operators is essential. Membership operators check whether a value exists in a sequence such as a list, string, or tuple. We will discuss lists and tuples in detail later in this course.

There are two membership operators in Python. They are:

- in — Returns True if the value exists in the sequence.
- not in — Returns True if the value does not exist in the sequence.

Let's take some examples.

Sample code 5:

```
# Using Membership Operators with Lists
my_list = [1, 2, 3, 4, 5]
print(3 in my_list)            # True
print(10 not in my_list)     # True
```

In this example, we have a list containing numbers from 1 to 5. We check if 3 is in the list, which returns True. Similarly, we check if 10 is not in the list, which also returns True.

Here's another example using a string:

Sample code 6:

```
# Using Membership Operators with Strings
text = "Hello, Teacher!"
print("Teacher" in text)   # True
print("Student" not in text) # True
```

In this example, we have a string 'Hello, Teacher!'. We check if 'Teacher' is in the string, and it returns True. Similarly, we check if 'Student' is not in the string, and it returns True.

Sample code 7:

```
#Using Membership Operators with Tuples
my_tuple = ('apple', 'banana', 'mango')
```

```
print('banana' in my_tuple)     # True
print('grape' not in my_tuple)  # True
```

In this example, we have a tuple with three elements: 'apple', 'banana', and 'mango'. We check if 'banana' is in the tuple, which returns True. We also check if 'grape' is not in the tuple, which returns True.

Sample code 8:

```
# Using Membership Operators with Sets
my_set = {10, 20, 30, 40}
print(20 in my_set)  # True
print(50 not in my_set)  # True
```

In this example, we have a set containing {10, 20, 30, 40}.
We check if 20 is in the set, which returns True.
We also check if 50 is not in the set, which returns True.

Sample code 9:

```
#Membership Operators with Dictionaries
my_dictionary = {'name': 'Rishi', 'age': 25, 'city': 'Guwahati'}
print('name' in my_dict)  # True
print('Rishi' in my_dict)  # False
print(25 not in my_dict)  # True
print('Guwahati' not in my_dictionary)    #True
```

When using membership operators with dictionaries, Python checks only the keys, not the values.
Here, 'name' is a key in the dictionary, so 'name' in my_dictionary' returns True. However, 'Rishi' is a value, not a key, so 'Rishi' in my_dict' returns False. Similarly, 25 not in my_dict returns True because 25 is a value, not a key. Lastly, we check if 'Guwahati' is not in the dictionary. 'Guwahati' not in my_dictionary' also returns True for the same reason-'Guwahati' is a value, not a key. When using membership operators with dictionaries, Python

checks only the keys by default. If we want to check for values, we need to explicitly check in my_dictionary.values().

3.3.7 Identity Operators

Identity Operators compare memory locations of two objects. Identity operators check if two variables point to the same object in memory. Python provides two identity operators-is and is not-to check if two variables reference the same memory location. Unlike equality comparison (==), which checks if values are the same, identity operators determine if two variables point to the same object in memory. Let us write some simple Python codes for identity operators and see the output.

Sample code 10:

```
# Identity operator
#Examples of is Operator
x = [1, 2, 3]
y = x  # y points to the same object as x
z = [1, 2, 3]  # z is a new object with the same values
print(x is y)  # Output: True
print(x is z)  # Output: False
```

In this example, x and y reference the same list in memory, so x is y returns True. However, z is a separate list with the same values as x, but it is stored in a different memory location, so x is z returns False. These operators help check whether two variables reference the same object.

Sample code 11:

```
# Example of is not Operator
a = 5
b = 5
c = 10
print(a is not b)       # Output: False (Same object)
print(a is not c)       # Output: True (Different object)
```

The is not operator returns True if two variables refer to different objects. In this case, a and b have the same value and reference the same memory location for small integers, so a is not b is False. However, a is not c is True, because a and c refer to different objects in memory. Let us take an example to understand the use of the identity operator with an if-else statement.

We define two variables:

Sample code 12:

```
a = 50
b = 50
if a is b:
   print("a and b have same identity")
else:
   print("a and b do not have same identity")
```

The output will be: a and b have same identity

Now, let's consider another example using lists. We create two lists, each containing four elements:

Sample code 13:

```
list1 = [1, 2, 3, 4]
list2 = [1, 2, 3, 4]
if list1 is list2:
   print("list1 and list2 are same")
else:
   print("list1 and list2 are not same")
```

The output will be: list1 and list2 are not same

At first glance, this may seem surprising because both lists contain the same elements. However, the output is different because the "is" operator does not compare values. Instead, it compares the identity numbers of the

objects. To understand this better, let's check the identity numbers of these two lists using the id() function:

```
id(list1)
id(list2)
```

Since both lists are created separately, they are stored at different memory locations, which means they have different identity numbers. Identity operators are useful when checking if two variables refer to the same object, rather than just comparing their values.

3.4 OPERATOR PRECEDENCE AND ASSOCIATIVITY

In Python, expression evaluation means computing the result of expressions, which are combinations of values, variables, operators, and function calls. When evaluating expressions in Python, operator precedence and associativity determine the order in which operations are performed.

Operator precedence defines the priority of operators when multiple operators appear in an expression. Operator precedence determines which operations are performed first in an expression. Operators with higher precedence are evaluated first. For example, multiplication has higher precedence than addition.

Table 4: Operators Precedence table

Highest	()	Parentheses (grouping)
	**	Exponentiation
	+a, -a, ~a	Unary plus, Unary minus, Bitwise complement
	*, / , // , %	Multiplication, Division, Floor division, Modulus
	+ , -	Addition, Subtraction
	<< , >>	Bitwise left shift, Bitwise right shift
	&	Bitwise AND
	^	Bitwise XOR
	|	Bitwise OR
	> , >=, <, <=, ==, !=	Relational (Comparison) operators Membership, and Identity
	=, %=, /=, //=, -=, +=, *=, **=	Assignment operators
	is, is not	Identity operators
	in, not in	Membership operators
	not	Logical not
	and	Logical and
Lowest	or	Logical or

Associativity helps resolve conflicts when multiple operators have the same precedence. When two operators of the same precedence appear in an expression, associativity determines the direction in which the expression is evaluated.

Table 5: Associativity table

Operators	Description	Associativity	Example	Evaluation
**	Exponentiation	Right to Left	2**3**2	2 ** (3 ** 2) = 2 ** 9 = 512
*, /, //, %, +, -	Arithmetic	Left to Right	10-5+2	(10 - 5) + 2 = 5 + 2 = 7
~	Bitwise NOT	Right to Left	~ 1001	0110
<<, >>, &, |, ^	Shifts, Bitwise AND, Bitwise XOR, Bitwise OR	Left to Right	a = 10 = 0000 1010 (Binary) a >> 1	a >> 1 = 0000 0101 = 5(Decimal)
==, !=, >, >=, <, <=, is, is not, in, not in	Relational, membership, identity	Left to Right	3 > 2 == 2	(3>2) and (2==2) → True and True → True
not	Boolean NOT	Right to Left	Not False	True
and	Boolean AND	Left to Right	True and False and True	(True and False) and True → False and True → False
or	Boolean OR	Left to Right	False or True or False	(False or True) or False → True or False → True

Let us take some examples to understand how Python evaluates expressions.

Example 1: Precedence

Consider the following expression:

print(10 + 2 * 3)

Since multiplication (*) has a higher precedence than addition (+), Python evaluates 2 * 3 first, which results in 6. Then, 10 + 6 gives the final output:16

Example 2: Associativity

Now, let us examine how operators of the same precedence level are evaluated.

print(10 - 5 - 2)

Subtraction follows left-to-right associativity, so Python evaluates (10 - 5), which is 5, and then 5 - 2, resulting in:3

Example 3: Exponentiation Associativity

Next, let's look at exponentiation.

print(2 ** 3 ** 2)

Exponentiation (**) follows right-to-left associativity, meaning Python evaluates 3 ** 2 first, which is 9, and then calculates 2 ** 9, resulting in: 512

Example 4: Logical AND and OR Precedence

Boolean operations also follow precedence rules.

print(True or False and False)

Here, AND (and) has higher precedence than OR (or), so Python evaluates False and False first, which results in False. Then, True or False gives the final result: True

Example 5: Using Parentheses for Clarity

Using parentheses can make expressions more readable and eliminate ambiguity. For example,

print((10 + 2) * 3)

Here, the parentheses ensure 10 + 2 is evaluated first, giving 12. Then, 12 * 3 results in: 36

Example 6: Write a Python expression to check if a number x is between 1 and 100 (inclusive).

```
x = int(input("Enter a number: "))
print(x >= 1 and x <= 100)
```

If x is between 1 and 100, it prints True, otherwise False. Here in the output screen, the result will be true for the number 55.

```
>>> x = int(input("Enter a number: "))
Enter a number: 55
>>> print(x >= 1 and x <= 100)
True
```

Example 7: Write a Python expression using logical operators that checks if a number is either less than 10 or greater than 100.

```
num = int(input("Enter a number: "))
result = num < 10 or num > 100
print("Is the number < 10 or > 100?:", result)
```

This expression will return True if the number is outside the range 10 to 100, and False if it is between them (inclusive).

Example 8: Write a Python program that takes two numbers and prints their sum, difference, product, quotient, and modulus.

```
num1 = float(input("Enter the first number: "))
num2 = float(input("Enter the second number: "))

# Perform operations
```

```
sum_result = num1 + num2
difference = num1 - num2
product = num1 * num2

# Handle division carefully
if num2 != 0:
    quotient = num1 / num2
    modulus = num1 % num2
else:
    quotient = "Undefined (division by zero)"
    modulus = "Undefined (division by zero)"

# Display results
print("\nResults:")
print(f"Sum = {sum_result}")
print(f"Difference = {difference}")
print(f"Product = {product}")
print(f"Quotient = {quotient}")
print(f"Modulus = {modulus}")
```

While we execute the program using command prompt, the output will be like this:

```
C:\pythonscript>python calc.py
Enter the first number: 4
Enter the second number: 2

Results:
Sum = 6.0
Difference = 2.0
Product = 8.0
Quotient = 2.0
Modulus = 0.0
```

While writing the code for Example 8, we used Python control statements, which will be discussed later in this course.



CHECK YOUR PROGRESS

Q2. Give the correct answer:

i. What will be the output of the statement 'p' in 'apple'?

A. True

B. False

C. None

D. Error

ii. What does the statement 'z' not in 'banana' return?

A. True

B. False

C. None

D. Error

iii. What does statement 3 not in (1, 2, 3, 4) return?

A. True

B. False

C. None

D. Error

iv. What will be the output of the expression 'Hello' in ['Hello', 'World']?

A. True

B. False

C. None

D. Error

Q3. What will be the output of the Python code:

```
3 not in (1, 2, 3, 4)
```

Q4. Write the output of the following Python code:

```
x = 5
x += 3
print(x)
```



3.5 LET US SUM UP

Python operators are essential for performing computations, comparisons, and logical evaluations. Key points include:

- Arithmetic operators handle mathematical calculations.
- Relational operators compare values, returning Boolean results.
- Logical operators combine conditions for decision-making.
- Assignment operators efficiently assign and modify values.
- Bitwise operators manipulate binary data.
- Identity operators check object identity, not just value equality.
- Membership operators verify presence in sequences.
- Operator precedence and associativity govern expression evaluation order, with parentheses providing clarity.

Understanding these operators enables efficient and precise Python programming.



3.6 ANSWERS TO CHECK YOUR PROGRESS

Ans.to.Q1.

1.i. C) % (The modulus operator)

1.ii. B) False

1.iii. A) True

1.iv. B) 20 (Operator precedence: 5 * 2 = 10, then 10 + 10 = 20)

Ans.to.Q2.

2.i. A) True

2.ii. A) True

2.iii. B) False

2.iv. A) True

Ans.to.Q3. False

Ans.to.Q4. 8



3.7 FURTHER READING

- Kanetkar, Y., & Kanetkar, A. (2024). Let us Python (6th ed.). BPB Publications.
- Rao, N. (2024). Core Python programming: PyCharm and Jupyter. Dreamtech Press.
- Python Official Documentation
  https://docs.python.org/3/tutorial/



3.8 MODEL QUESTIONS

1. What are arithmetic operators in Python? List them with examples.
2. Differentiate between == and is operators in Python with examples.
3. What will be the output of the following Python code?

```
a = 10
b = 20
print(a > 5 and b < 25)
```

4. What is the result of 10 + 5 * 3? Write the order of operations.
5. Write a Python expression to check whether a number is between 50 and 100 (inclusive).
6. What is the difference between / and // operators in Python? Give an example.
7. What will be the result of 3 ** 2 ** 2 and why?
8. State the purpose of the in operator. Give an example of a logical operator in Python.
9. What is operator precedence? Provide an example showing how parentheses can change the outcome of an expression.
10. Write a Python program to check if a given character exists in a string using membership operators.
11. Explain any three types of operators in Python with one example for each.
12. Define associativity of operators. What is the associativity of the exponentiation operator ** in Python?

*****

UNIT 4: CONTROL STRUCTURES

UNIT STRUCTURE

4.1 Learning Objectives
4.2 Introduction
4.3 Conditional Statements
    4.3.1 if-else Statements
    4.3.2 Nested if-else Statements
    4.3.3 elif Statements
4.4 Looping Statements
    4.4.1 for Loop
    4.4.2 while Loop
4.5 Control Flow Statements
4.5. 1 break Statement
    4.5.2 continue Statement
    4.5.3 pass Statement
4.6 Summing Up
4.7 Answers to Check Your Progress
4.8 References
4.9 Further Readings
4.10 Model Questions

4.1 LEARNING OBJECTIVES

By the end of this unit, you should be able to:

- Understand and use conditional statements like if, else, and elif to make decisions in your code.
- Implement nested conditional statements to handle complex decision-making scenarios.
- Use looping statements like for and while to repeat tasks.
- Apply control flow statements like break, continue, and pass to manage the flow of loops.
- Write Python programs that effectively use control structures to solve problems.

4.2 INTRODUCTION

Control structures are the backbone of programming, allowing you to control the flow of execution in your code. In Python, control structures include conditional statements, looping statements, and control flow statements. These structures enable you to make decisions, repeat tasks, and manage the flow of your program. In this unit, we will explore these concepts in detail, with plenty of examples to help you understand how they work and how to use them effectively.

4.3 CONDITIONAL STATEMENTS

Conditional statements allow your program to make decisions based on certain conditions. The most common conditional statements in Python are if, else, and elif.

4.3.1 if-else Statements

In Python, an if-else statement is used to make a decision between two possible outcomes. The 'if' part checks a condition, and if that condition is 'True', the code inside the 'if' block runs. If the condition is 'False', the code inside the 'else' block runs instead. This ensures that one of the two blocks will always execute, making it useful when you want your program to choose between two clear options.

Syntax:

```
if condition:
    # Code to execute if condition is True
else:
    # Code to execute if condition is False
```

Example 1: Checking Voting Eligibility Using if-else Statement

```
age = 18
if age >= 18:
    print("You are eligible to vote.")
else:
    print("You are not eligible to vote.")
```

Output: You are eligible to vote.

Explanation:

- The program checks if age is greater than or equal to 18.
- If the condition is True, it prints "You are eligible to vote."
- If the condition is False, it prints "You are not eligible to vote."

Example 2: Checking Weather Condition Using if-else Statement

```
temperature = 25
if temperature > 30:
    print("It's a hot day.")
else:
    print("It's not a hot day.")
```

Output: It's not a hot day.

Explanation:

- The program checks if temperature is greater than 30.
- Since 25 is not greater than 30, it prints "It's not a hot day."

4.3.2 Nested if-else Statements

In Python, a nested if-else statement means having an 'if-else' block inside another 'if' or 'else' block. This allows you to check multiple conditions in a step-by-step manner. First, the outer 'if' condition is tested. If it is 'True', the inner 'if-else' block is checked; otherwise, the outer 'else' block runs. Nested if-else statements are useful when decisions depend on more than one condition, and the outcome of one test leads to another decision.

Syntax:

```
if condition1:
    if condition2:
        # Code to execute if both condition1 and condition2 are True
    else:
        # Code to execute if condition1 is True but condition2 is False
else:
    # Code to execute if condition1 is False
```

Example 1: Checking Number Properties Using Nested if-else Statement

```
num = 10
if num > 0:
    if num % 2 == 0:
        print("The number is positive and even.")
    else:
        print("The number is positive and odd.")
else:
    print("The number is not positive.")
```

Output: The number is positive and even.

Explanation:

- The program first checks if num is greater than 0.
- If True, it checks if num is even or odd.
- If num is not positive, it prints "The number is not positive."

Example 2: Determining Grades Using Nested if-else Statement

```
score = 85
if score >= 90:
    print("Grade: A")
else:
    if score >= 80:
        print("Grade: B")
    else:
        if score >= 70:
            print("Grade: C")
        else:
            print("Grade: D")
```

Output: Grade: B

Explanation:

The program checks the value of score and assigns a grade based on the range it falls into.

4.3.3 elif Statements

In Python, an elif statement (short for else if) is used when you want to check multiple conditions one after another. It comes after an 'if' statement

and before an 'else' statement. If the initial 'if' condition is 'False', Python moves on to test the 'elif' condition. If that is 'True', the code under it runs. You can have multiple 'elif' statements to handle different possibilities, but only the first condition that evaluates to 'True' will execute. If none of the 'if' or 'elif' conditions are 'True', the 'else' block (if present) runs.

Syntax:

```
if condition1:
   # Code to execute if condition1 is True
elif condition2:
   # Code to execute if condition2 is True
else:
   # Code to execute if all conditions are False
```

Example: Determining Grades Using if-elif-else Statement

```
score = 85
if score >= 90:
   print("Grade: A")
elif score >= 80:
   print("Grade: B")
elif score >= 70:
   print("Grade: C")
else:
   print("Grade: D")
```

Output: Grade: B

Explanation:

- The program checks the value of score and assigns a grade based on the range it falls into.
- The elif statement makes the code more readable compared to nested if-else statements.

4.4 LOOPING STATEMENTS

In Python, loops are used to repeatedly execute a block of code as long as a certain condition is true or for a fixed number of times. Loops help automate repetitive tasks, making programs more efficient and concise. The two main types of loops in Python are the `for` loop, which iterates over a sequence like a list or range, and the `while` loop, which continues running as long as a specified condition remains true. Loops can also include `break` and `continue` statements to control the flow of iteration.

4.4.1 for Loop

In Python, a for loop is used to repeat a block of code a specific number of times or for each item in a sequence of values. It uses the for keyword, a variable to represent the current item, and a colon followed by an indented block of code to execute. The range() function is often used with for loops to control how many times the loop runs, making it useful for tasks that need repetition a set number of times.

Syntax:

```
for variable in sequence:
    # Code to execute for each item in the sequence
```

Example 1:

```
for i in range(5):
    print("Iteration:", i)
```

Output:

```
Iteration: 0
Iteration: 1
Iteration: 2
Iteration: 3
```

Iteration: 4

Explanation:

- The for loop iterates over a sequence of numbers generated by range(5).
- The loop runs 5 times, with i taking values from 0 to 4.

4.4.2 while Loops

In Python, a while loop is used to repeatedly execute a block of code as long as a specified condition is `True`. The loop first checks the condition, and if it is true, the code inside the loop runs. After each iteration, the condition is checked again, and the loop continues until the condition becomes `False`. While loops are useful when the number of repetitions is not known in advance and depends on a condition being met.

Syntax:

```
while condition:
    # Code to execute while the condition is True
```

Example 1:

```
count = 0
while count < 5:
    print("Count:", count)
    count += 1
```

Output:

```
Count: 0
Count: 1
Count: 2
Count: 3
Count: 4
```

Explanation:

? The while loop continues to execute as long as count is less than 5.

? After each iteration, count is incremented by 1.

Example 2:

```
num = 5
while num > 0:
    print("Number:", num)
    num -= 1
```

Output:

```
Number: 5
Number: 4
Number: 3
Number: 2
Number: 1
```

Explanation:

- The while loop continues to execute as long as num is greater than 0.
- After each iteration, num is decremented by 1.

4.5 CONTROL FLOW STATEMENTS

Control flow statements like break, continue, and pass allow you to alter the flow of loops.

4.5.1 break Statement

In Python, the break statement is used to immediately exit a loop, even if the loop's condition is still `True` or there are remaining items to iterate over. It is commonly used inside `for` or `while` loops to stop the loop when a certain condition is met, preventing further iterations. This helps in controlling the flow of the program and avoiding unnecessary processing once the desired result is achieved.

Example:

```
for i in range(10):
    if i == 5:
        break
    print("Iteration:", i)
```

Output:

```
Iteration: 0
Iteration: 1
Iteration: 2
Iteration: 3
Iteration: 4
```

Explanation:

- The loop stops executing when i equals 5 due to the break statement.

4.5.2 continue Statement

In Python, the continue statement is used inside a loop to skip the rest of the code in the current iteration and move directly to the next iteration. It does not terminate the loop entirely but allows the loop to continue running with the next value or condition check. This is useful when you want to ignore certain cases while still completing the overall loop.

Example:

```
for i in range(5):
    if i == 2:
        continue
    print("Iteration:", i)
```

Output:

Iteration: 0

Iteration: 1

Iteration: 3

Iteration: 4

Explanation:

- The loop skips the iteration where i equals 2 due to the continue statement.

4.5.3 pass Statement

The pass statement is a placeholder that does nothing. It is used when a statement is required syntactically but no action is needed.

Example:

```
for i in range(5):
    if i == 2:
        pass  # Do nothing
    print("Iteration:", i)
```

Output:

Iteration: 0

Iteration: 1

Iteration: 2

Iteration: 3

Iteration: 4

Explanation:

- The pass statement does nothing, so the loop continues to execute normally.



CHECK YOUR PROGRESS

Choose the correct option:

1. Which Python statement is used to execute a block of code only when a specific condition is true?
   a) elif
   b) pass
   c) if
   d) else
2. What happens in an if-else statement when the condition is false?
   a) Nothing happens
   b) The program stops
   c) The else block executes
   d) The if block executes
3. Why are nested if-else statements used in Python?
   a) To exit the program immediately
   b) To repeat code multiple times
   c) To skip certain iterations
   d) To check multiple conditions step by step
4. Which statement is used to skip the current iteration of a loop without terminating the loop?
   a) break
   b) else
   c) continue
   d) pass
5. What is the purpose of the elif statement in Python?
   a) To do nothing in a loop
   b) To replace the if statement entirely
   c) To check additional conditions after an if
   d) To terminate a loop



4.6 SUMMING UP

- Python uses if, else, and elif to make decisions based on conditions, allowing programs to choose different actions depending on the situation.
- An if-else statement executes one block of code if a condition is True and another if it is False, ensuring one of the two paths always runs.
- Nested if-else statements allow multiple conditions to be checked step by step by placing one if-else block inside another.
- The elif statement checks additional conditions after an if; only the first True condition executes, improving readability compared to nested if-else.
- Loops, like for and while, are used to repeat code multiple times, automating repetitive tasks efficiently.
- Control flow statements such as break, continue, and pass allow fine control over loop execution by exiting, skipping, or doing nothing in an iteration.



4.7 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q.No.1: c)

Ans. to Q.No.2: c)

Ans. to Q.No.3: d)

Ans. to Q.No.4: c)

Ans. to Q.No.5: c)

4.8 REFERENCES

- Python Documentation: https://docs.python.org/3/tutorial/controlflow.html#defining-functions



4.9 FURTHER READINGS

- Kanetkar, Y., & Kanetkar, A. (2023). Let us Python. BPB Publications.
- https://www.w3schools.com/python/



4.10 MODEL QUESTIONS

1. Write a Python program to check if a number is positive, negative, or zero.
2. Explain the difference between break and continue with examples.
3. Write a for loop to print all even numbers between 1 and 20.
4. Write a while loop to calculate the sum of numbers from 1 to 10.
5. What will be the output of the following code?

```
for i in range(5):
    if i == 3:
        continue
    print(i)
```

****

UNIT 5: FUNCTIONS

UNIT STRUCTURE

5.1 Learning Objectives

5.2 Introduction

5.3 Defining and Calling Functions

- 5.3.1 Defining Functions
- 5.3.2 Calling Functions

5.4 Parameters and Arguments

- 5.4.1 Positional Arguments
- 5.4.2 Keyword Arguments
- 5.4.3 Default Arguments
- 5.4.4 Variable-Length Arguments

5.5 Return Values

- 5.5.1 Returning Single Values
- 5.5.2 Returning Multiple Values

5.6 Scope of Variables

- 5.6.1 Local Variables
- 5.6.2 Global Variables

5.7 Lambda Functions

5.8 Summing Up

5.9 Answers to Check Your Progress

5.10 Further Readings

5.11 Model Questions

5.1 LEARNING OBJECTIVES

By the end of this unit, you should be able to:

- Understand the concept of functions and their role in Python programming.
- Define and call functions in Python.
- Work with parameters and arguments to make functions more flexible.
- Return values from functions to pass data back to the caller.

- Understand the scope of variables within functions.
- Use lambda functions for small, anonymous functions.

5.2 INTRODUCTION

Functions are one of the most important concepts in programming. They allow you to encapsulate a block of code that performs a specific task, making your code more modular, reusable, and easier to understand. In Python, functions are defined using the def keyword and can accept parameters and return values. This unit will introduce you to the basics of functions in Python, including how to define and call functions, work with parameters and return values, and understand the scope of variables.

5.3 DEFINING AND CALLING FUNCTIONS

In Python, functions are blocks of reusable code designed to perform a specific task. You can define a function using the def keyword followed by the function name, parentheses, and a colon. Inside the function, you write the code that should execute when the function is called. To call a function, you simply use its name followed by parentheses. Functions help make programs more organized, reduce code repetition, and improve readability. They can also accept input values, called parameters, and return results, allowing you to use them in different parts of your program efficiently.

5.3.1 Defining Functions

In Python, you define a function using the def keyword, followed by the function name and parentheses ().

Syntax:

```
def function_name():
    # Code to execute
```

Example: A Function That Greets a User

```
def greet():
    print("Hello, World!")
```

Explanation:

The greet function is defined to print "Hello, World!" when called.

5.3.2 Calling Functions

Once a function is defined, you can call it by using its name followed by parentheses ().

Example:

```
def greet():
    print("Hello, World!")

# Calling the function
greet()
```

Output:

```
Hello, World!
```

Explanation:

- The greet function is called, and it prints "Hello, World!".

5.4 PARAMETERS AND ARGUMENTS

Functions can accept parameters, which are variables that you pass to the function when you call it. These parameters allow you to make your functions more flexible and reusable.

5.4.1 Positional Arguments

Positional arguments are the most common type of arguments. They are passed to the function in the order they are defined.

Example:

```
def add(a, b):
    print("Sum:", a + b)

# Calling the function with positional arguments
add(3, 5)
```

Output:

```
Sum: 8
```

Explanation:

- The add function takes two parameters, a and b.
- When the function is called with add(3, 5), a is assigned the value 3 and b is assigned the value 5.

5.4.2 Keyword Arguments

Keyword arguments allow you to pass arguments to a function by specifying the parameter names. This makes the function call more readable and allows you to pass arguments in any order.

Example:

```
def greet(name, message):
    print(f"{message}, {name}!")

# Calling the function with keyword arguments
greet(name="Rajib", message="Hello")
```

Output:

```
Hello, Rajib!
```

Explanation:

- The greet function is called with keyword arguments, making it clear which value corresponds to which parameter.

5.4.3 Default Arguments

Default arguments allow you to specify default values for parameters. If the caller does not provide a value for a parameter, the default value is used.

Example:

```
def greet(name, message="Hello"):
    print(f"{message}, {name}!")

# Calling the function without providing the message argument
greet("Rajib")
```

Output:

```
Hello, Rajib!
```

Explanation:

- The greet function has a default value for the message parameter.
- Since the caller did not provide a value for message, the default value "Hello" is used.

5.4.4 Variable-Length Arguments

Sometimes, you may want to pass a variable number of arguments to a function. Python allows you to do this using *args for positional arguments and **kwargs for keyword arguments.

Example:

```
def print_args(*args):
```

```
    for arg in args:
        print(arg)

# Calling the function with multiple arguments
print_args(1, 2, 3, 4)
```

Output:

```
1
2
3
4
```

Explanation:

- The print_args function accepts a variable number of positional arguments.
- When the function is called, all arguments are packed into a tuple and printed.

5.5 RETURN VALUES

Functions can return values to the caller using the return statement. This allows you to pass data back to the caller after the function has executed.

5.5.1 Returning Single Values

Example:

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

# Calling the function and storing the result
result = add(3, 5)
print("Sum:", result)
```

Output:

Sum: 8

Explanation:

- The add function returns the sum of a and b.
- The returned value is stored in the result variable and printed.

5.5.2 Returning Multiple Values

Python allows you to return multiple values from a function by separating them with commas.

Example:

```
def calculate(a, b):
    return a + b, a - b, a * b

# Calling the function and unpacking the result
sum_result, diff_result, prod_result = calculate(10, 5)
print("Sum:", sum_result)
print("Difference:", diff_result)
print("Product:", prod_result)
```

Output:

Sum: 15
Difference: 5
Product: 50

Explanation:

- The calculate function returns three values: the sum, difference, and product of a and b.
- The returned values are unpacked into separate variables and printed.

5.6 SCOPE OF VARIABLES

The scope of a variable determines where it can be accessed within your code. In Python, variables can have either local or global scope.

5.6.1 Local Variables

Local variables are defined within a function and can only be accessed inside that function.

Example:

```
def my_function():
    local_var = 10
    print("Local variable:", local_var)

my_function()
# print(local_var)  # This would raise an error
```

Output:

```
Local variable: 10
```

Explanation:

- local_var is a local variable defined within my_function.
- It cannot be accessed outside the function.

5.6.2 Global Variables

Global variables are defined outside of any function and can be accessed throughout the entire program.

Example:

```
global_var = 20

def my_function():
```

```
    print("Global variable:", global_var)

my_function()
```

Output:

```
Global variable: 20
```

Explanation:

- global_var is a global variable that can be accessed inside my_function.

5.7 LAMBDA FUNCTIONS

Lambda functions are small, anonymous functions defined using the lambda keyword. They are often used for short, simple operations.

Syntax:

```
lambda arguments: expression
```

Example:

```
# Defining a lambda function
square = lambda x: x ** 2

# Calling the lambda function
print("Square of 5:", square(5))
```

Output:

```
Square of 5: 25
```

Explanation:

- The lambda function square takes one argument x and returns its square.
- It is called with the argument 5, and the result is printed.



CHECK YOUR PROGRESS

1. Which keyword is used to define a function in Python?
   a) func
   b) define
   c) def
   d) lambda
2. What type of arguments allow you to pass values to a function by specifying parameter names?
   a) Positional arguments
   b) Keyword arguments
   c) Default arguments
   d) Variable-length arguments
3. Which statement is used in Python to send a value back from a function to the caller?
   a) send
   b) return
   c) yield
   d) print
4. Where can a local variable in Python be accessed?
   a) Anywhere in the program
   b) Only inside the function where it is defined
   c) Only in other functions
   d) Outside the function but not inside it
5. What is the purpose of a lambda function in Python?
   a) To define large functions
   b) To create small anonymous functions for simple operations
   c) To declare global variables
   d) To create classes



5.8 SUMMING UP

- Functions in Python are reusable blocks of code defined using the def keyword, which help organize programs, reduce repetition, and improve readability.
- Functions are called using their name followed by parentheses, which executes the code inside them.
- Parameters and arguments allow functions to accept input values, making them flexible; these include positional arguments, keyword arguments, default arguments, and variable-length arguments (*args and **kwargs).
- Functions can return values to the caller using the return statement, supporting both single and multiple return values.
- Variable scope determines accessibility: local variables exist only within a function, while global variables can be accessed throughout the program.
- Lambda functions are small, anonymous functions defined using the lambda keyword, often used for simple, concise operations.



5.9 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q.No.1: c)

Ans. to Q.No.2: b)

Ans. to Q.No.3: b)

Ans. to Q.No.4: b)

Ans. to Q.No.5: b)

5.10 REFERENCES

- Python Documentation: https://docs.python.org/3/tutorial/controlflow.html#defining-functions



5.11 FURTHER READINGS

- Kanetkar, Y., & Kanetkar, A. (2023). Let us Python. BPB Publications.https://www.w3schools.com/python/



5.12 MODEL QUESTIONS

1. Define a function in Python. Explain its advantages with a suitable example.
2. Write the syntax for defining a function in Python. Explain each part of the syntax.
3. With an example, explain how to call a function in Python after defining it.
4. What are parameters and arguments in Python functions? Explain with suitable examples of positional, keyword, and default arguments.
5. What are variable-length arguments in Python? Differentiate between *args and **kwargs with examples.
6. Explain the concept of return values in Python functions. Write an example that returns multiple values and explain the output.
7. Differentiate between local variables and global variables in Python with examples.
8. Explain lambda functions in Python. Write a lambda function to calculate the square and cube of a number.
9. Write a Python function named student_info that accepts a student's name, age, and any number of subjects (using variable-length arguments) and prints the details.
10. Summarize the key features of Python functions discussed in the unit.

****

UNIT 6: LISTS IN PYTHON

UNIT STRUCTURE

6.1 Learning Objectives

6.2 Introduction

6.3 Creating Lists

6.4 Manipulating Lists

6.5 Slicing and Indexing

6.6 List Methods

6.7 Nested Lists

6.8 Summing Up

6.9 Answers to Check Your Progress

6.10 References

6.11 Further Readings

6.12 Model Questions

6.1 LEARNING OBJECTIVES

By the end of this unit, you will be able to:

- Understand the concept of lists and their role in Python programming.
- Create and access lists in Python.
- Manipulate lists by adding, removing, and modifying elements.
- Use slicing and indexing to access specific elements or sublists.
- Apply common list methods to perform various operations on lists.
- Understand and use list comprehensions for concise list creation.
- Work with nested lists to handle more complex data structures.

6.2 INTRODUCTION

Lists are one of the most versatile and commonly used data structures in Python. A list is an ordered collection of items that can be of any type, including numbers, strings, and even other lists. Lists are mutable, meaning you can change their content without creating a new list. This unit will introduce you to the basics of lists in Python, including how to create, ma-

nipulate, slice, and index lists. We will also explore some advanced topics like list methods and nested lists.

6.3 CREATING LISTS

In Python, lists are created by placing elements inside square brackets [], separated by commas.

Example 1:

```
# Creating a list of integers
numbers = [1, 2, 3, 4, 5]

# Creating a list of strings
fruits = ["apple", "banana", "cherry"]

# Creating a mixed list
mixed_list = [1, "apple", 3.14, True]

print(numbers)
print(fruits)
print(mixed_list)
```

Output:

```
[1, 2, 3, 4, 5]
['apple', 'banana', 'cherry']
[1, 'apple', 3.14, True]
```

Explanation:

- Lists can contain elements of any type, including integers, strings, floats, and booleans.

Accessing List Elements

You can access individual elements of a list using their index. Python uses zero-based indexing, meaning the first element is at index 0.

Example 2:

```
fruits = ["apple", "banana", "cherry"]

# Accessing the first element
print(fruits[0])

# Accessing the second element
print(fruits[1])

# Accessing the third element
print(fruits[2])
```

Output:

```
apple
banana
cherry
```

Explanation:

- The first element is accessed using fruits[0], the second with fruits[1], and so on.

6.4 MANIPULATING LISTS

(a) Adding Elements

You can add elements to a list using methods like append(), insert(), and extend().

Example:

```
fruits = ["apple", "banana"]

# Adding an element to the end of the list
```

```
fruits.append("cherry")
print(fruits)

# Inserting an element at a specific position
fruits.insert(1, "orange")
print(fruits)

# Adding multiple elements to the list
fruits.extend(["grape", "kiwi"])
print(fruits)
```

Output:

```
['apple', 'banana', 'cherry']
['apple', 'orange', 'banana', 'cherry']
['apple', 'orange', 'banana', 'cherry', 'grape', 'kiwi']
```

Explanation:

- append() adds an element to the end of the list.
- insert() adds an element at a specific position.
- extend() adds multiple elements to the list.

(b) Removing Elements

You can remove elements from a list using methods like remove(), pop(), and del.

Example:

```
fruits = ["apple", "banana", "cherry", "orange"]

# Removing a specific element
fruits.remove("banana")
print(fruits)
```

```
# Removing an element by index
fruits.pop(1)
print(fruits)

# Deleting an element by index
del fruits[0]
print(fruits)
```

Output:

```
['apple', 'cherry', 'orange']
['apple', 'orange']
['orange']
```

Explanation:

- remove() removes the first occurrence of the specified element.
- pop() removes and returns the element at the specified index.
- del deletes the element at the specified index.

(c) Modifying Elements

You can modify elements in a list by assigning a new value to a specific index.

Example:

```
fruits = ["apple", "banana", "cherry"]

# Modifying the second element
fruits[1] = "orange"
print(fruits)
```

Output:

```
['apple', 'orange', 'cherry']
```

Explanation:

- The second element is modified by assigning a new value to fruits[1].

6.5 SLICING AND INDEXING

(a) Indexing

Indexing allows you to access individual elements of a list using their position.

Example:

```
fruits = ["apple", "banana", "cherry", "orange"]

# Accessing the first element
print(fruits[0])

# Accessing the last element
print(fruits[-1])
```

Output:

```
apple
orange
```

Explanation:

- Positive indices start from the beginning of the list, while negative indices start from the end.

(b) Slicing

Slicing allows you to access a sublist by specifying a range of indices.

Example:

```
fruits = ["apple", "banana", "cherry", "orange", "kiwi"]
# Slicing from index 1 to 3
```

```
print(fruits[1:3])

# Slicing from the beginning to index 2
print(fruits[:2])

# Slicing from index 2 to the end
print(fruits[2:])

# Slicing with a step
print(fruits[::2])
```

Output:

```
['banana', 'cherry']
['apple', 'banana']
['cherry', 'orange', 'kiwi']
['apple', 'cherry', 'kiwi']
```

Explanation:

- fruits[1:3] returns elements from index 1 to 2.
- fruits[:2] returns elements from the beginning to index 1.
- fruits[2:] returns elements from index 2 to the end.
- fruits[::2] returns every second element.

6.6 LIST METHODS

(a) Common List Methods

Python provides several built-in methods to manipulate lists.

Example:

```
fruits = ["apple", "banana", "cherry"]

# Finding the index of an element
```

```
print(fruits.index("banana"))

# Counting occurrences of an element
print(fruits.count("apple"))

# Sorting the list
fruits.sort()
print(fruits)

# Reversing the list
fruits.reverse()
print(fruits)
```

Output:

```
1
1
['apple', 'banana', 'cherry']
['cherry', 'banana', 'apple']
```

Explanation:

- index() returns the index of the first occurrence of the specified element.
- count() returns the number of occurrences of the specified element.
- sort() sorts the list in ascending order.
- reverse() reverses the order of the list.

(b) List Comprehensions

List comprehensions provide a concise way to create lists.

Example:

```
# Creating a list of squares
squares = [x ** 2 for x in range(10)]
print(squares)
```

```
# Creating a list of even numbers
evens = [x for x in range(20) if x % 2 == 0]
print(evens)
```

Output:

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Explanation:

? The first list comprehension creates a list of squares of numbers from 0 to 9.

? The second list comprehension creates a list of even numbers from 0 to 19.

6.7 NESTED LISTS

Nested lists are lists that contain other lists. They are useful for representing more complex data structures like matrices.

Example:

```
# Creating a nested list
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
# Accessing elements in a nested list
print(matrix[0][1])  # Accessing the second element of the first list
print(matrix[1][2])  # Accessing the third element of the second list
```

Output:

```
2
6
```

Explanation:

- matrix[0][1] accesses the second element of the first list.
- matrix[1][2] accesses the third element of the second list.



CHECK YOUR PROGRESS

Choose the correct option:

Q1. Which of the following is the correct way to create a list in Python?

a) list = (1, 2, 3)

b) list = {1, 2, 3}

c) list = [1, 2, 3]

d) list = <1, 2, 3>

Q2. If fruits = ["apple", "banana", "cherry"], what will fruits[1] return?

a) apple

b) banana

c) cherry

d) error

Q3. Which method is used to add multiple elements to an existing list?

a) append()

b) insert()

c) extend()

d) add()

Q4. What will be the output of the following code?

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

print(numbers[-2])

a) 20

b) 30

c) 40

d) 50

Q5. Which of the following statements about lists is false?

a) Lists are mutable.

b) Lists can contain elements of different data types.

c) Lists support both positive and negative indexing.

d) Lists cannot contain another list as an element.



6.8 SUMMING UP

- Lists in Python are created using square brackets [] and can store different data types like integers, strings, floats, and booleans.
- Elements are accessed using indices starting from 0, and lists are mutable, so elements can be modified directly.
- Elements can be added using append(), insert(), and extend(), and removed using remove(), pop(), or del.
- Lists support positive and negative indexing as well as slicing (list[start:end:step]) to extract sublists or patterns.
- Built-in methods like index(), count(), sort(), and reverse() help manipulate lists, and list comprehensions provide a concise way to create lists.
- Nested lists allow storing lists within lists and can represent structures like matrices, where elements are accessed using double indices.



6.9 ANSWERS TO CHECK YOUR PROGRESS

Ans.to.Q1: c)

Ans.to.Q2: b)

Ans.to.Q3: c)

Ans.to.Q4: c)

Ans.to.Q5: d)

6.10 REFERENCES

- Python Documentation: https://docs.python.org/3/tutorial/introduction.html#lists



6.11 FURTHER READINGS

- Kanetkar, Y., & Kanetkar, A. (2023). Let us Python. BPB Publications.
- https://www.w3schools.com/python/



6.12 MODEL QUESTIONS

1. Define a list in Python. Write examples of creating a list of integers, strings, and a mixed list.
2. Explain how list elements are accessed in Python using indexing. Illustrate with an example of both positive and negative indices.
3. Write a short note on modifying elements in a list. Give an example to replace an element at a particular index.
4. Write a Python program to demonstrate the use of append(), insert(), and extend() methods with suitable outputs.
5. Differentiate between remove(), pop(), and del with examples.
6. Explain list slicing with examples for the following cases:
   a. slicing with a start and end index
   b. slicing from the beginning to a given index
   c. slicing with a step
7. Describe any four commonly used list methods (index(), count(), sort(), reverse()) with examples.
8. What are list comprehensions? Explain with examples to generate (i) a list of squares of numbers from 1 to 10 and (ii) a list of even numbers from 1 to 20.
9. What are nested lists in Python? Write an example of creating a 3x3 matrix using a nested list and explain how to access its elements.
10. Summarize the key features of Python lists with suitable examples.

****

UNIT 7: TUPLES

UNIT STRUCTURE

7.1 Learning Objectives
7.2 Introduction
7.3 Creating Tuples
    7.3.1 Accessing Tuple Elements
7.4 Manipulating Tuples
    7.4.1 Immutability of Tuples
    7.4.2 Concatenating Tuples
    7.4.3 Repeating Tuples
7.5 Difference Between Lists and Tuples
7.6 Tuple Methods
7.7 Summing Up
7.8 Answers to Check Your Progress
7.9 References
7.10 Further Readings
7.11 Model Questions

7.1 LEARNING OBJECTIVES

By the end of this unit, you will be able to:

- Understand the concept of tuples and their role in Python programming.
- Create and access tuples in Python.
- Manipulate tuples by concatenating and repeating them.
- Understand the immutability of tuples and its implications.
- Differentiate between lists and tuples in terms of mutability, performance, and use cases.
- Use common tuple methods to perform operations on tuples.

7.2 INTRODUCTION

Tuples are another fundamental data structure in Python, similar to lists but with one key difference: tuples are immutable. This means that once a

tuple is created, its elements cannot be changed, added, or removed. Tuples are often used for fixed collections of items, such as coordinates, database records, or configurations. In this unit, we will explore how to create and manipulate tuples, understand their immutability, and compare them with lists.

7.3 CREATING TUPLES

(a) Creating Tuples

Tuples are created by placing elements inside parentheses (), separated by commas. If a tuple contains only one element, you need to include a trailing comma.

Example:

```
# Creating a tuple of integers
numbers = (1, 2, 3, 4, 5)

# Creating a tuple of strings
fruits = ("apple", "banana", "cherry")

# Creating a tuple with a single element
single_element_tuple = (42,)

# Creating a mixed tuple
mixed_tuple = (1, "apple", 3.14, True)

print(numbers)
print(fruits)
print(single_element_tuple)
print(mixed_tuple)
```

Output:

(1, 2, 3, 4, 5)
('apple', 'banana', 'cherry')
(42,)
(1, 'apple', 3.14, True)

Explanation:

- Tuples can contain elements of any type, including integers, strings, floats, and booleans.
- A single-element tuple requires a trailing comma to distinguish it from a regular value in parentheses.

(b) Accessing Tuple Elements

You can access individual elements of a tuple using their index, just like with lists. Python uses zero-based indexing.

Example:

```
fruits = ("apple", "banana", "cherry")

# Accessing the first element
print(fruits[0])

# Accessing the second element
print(fruits[1])

# Accessing the third element
print(fruits[2])
```

Output:

```
apple
banana
cherry
```

Explanation:

- The first element is accessed using fruits[0], the second with fruits[1], and so on.

7.4 MANIPULATING TUPLES

(a) Immutability of Tuples

Tuples are immutable, meaning their elements cannot be changed after creation. However, you can create new tuples by concatenating or repeating existing tuples.

Example:

```
fruits = ("apple", "banana", "cherry")

# Attempting to modify a tuple will raise an error
# fruits[1] = "orange"  # This will raise a TypeError

# Creating a new tuple by concatenation
new_fruits = fruits + ("orange", "kiwi")
print(new_fruits)
```

Output:

```
('apple', 'banana', 'cherry', 'orange', 'kiwi')
```

Explanation:

- Tuples cannot be modified directly, but you can create new tuples by combining existing ones.

(b) Concatenating Tuples

You can concatenate tuples using the + operator.

Example:

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)

# Concatenating two tuples
combined_tuple = tuple1 + tuple2
print(combined_tuple)
```

Output:

```
(1, 2, 3, 4, 5, 6)
```

Explanation:

- The + operator combines the elements of tuple1 and tuple2 into a new tuple.

(c) Repeating Tuples

You can repeat the elements of a tuple using the * operator.

Example:

```
tuple1 = (1, 2, 3)

# Repeating the tuple
repeated_tuple = tuple1 * 3
print(repeated_tuple)
```

Output:

```
(1, 2, 3, 1, 2, 3, 1, 2, 3)
```

Explanation:

The * operator repeats the elements of tuple1 three times.

7.5 DIFFERENCE BETWEEN LISTS AND TUPLES

(a) Mutability

Lists: Lists are mutable, meaning you can change their elements after creation.

Tuples: Tuples are immutable, meaning their elements cannot be changed after creation.

Example:

```
# Lists are mutable
my_list = [1, 2, 3]
my_list[1] = 20
print(my_list)

# Tuples are immutable
my_tuple = (1, 2, 3)
# my_tuple[1] = 20  # This will raise a TypeError
```

Output:

```
[1, 20, 3]
```

Explanation:

- Lists can be modified, but tuples cannot.

(b) Performance

- Lists: Lists are generally slower than tuples because they are mutable and require more memory.
- Tuples: Tuples are faster and more memory-efficient than lists because they are immutable.

(c) Use Cases

- Lists: Use lists when you need a collection of items that may change over time, such as a to-do list or a list of user inputs.
- Tuples: Use tuples when you need a fixed collection of items that should not change, such as coordinates or configuration settings.

7.6 TUPLE METHODS

Common Tuple Methods

Tuples have fewer methods than lists due to their immutability, but they still support some useful operations.

Example:

```
fruits = ("apple", "banana", "cherry", "banana")

# Counting occurrences of an element
print(fruits.count("banana"))

# Finding the index of an element
print(fruits.index("cherry"))
```

Output:

```
2
2
```

Explanation:

- count() returns the number of occurrences of the specified element.
- index() returns the index of the first occurrence of the specified element.



CHECK YOUR PROGRESS

Choose the correct option:

Q1. How do you create a tuple with a single element 42?

a) (42)

b) (42,)

c) [42]

d) {42}

Q2. Which of the following statements about tuples is true?

a) Tuples are mutable.

b) Tuples support more methods than lists.

c) Tuples are immutable.

d) Tuples cannot contain elements of different data types.

Q3. What will be the output of the following code?

```
tuple1 = (1, 2, 3)
tuple2 = tuple1 * 2
print(tuple2)
```

a) (1, 2, 3, 2, 3, 1)

b) (1, 2, 3, 1, 2, 3)

c) (1, 2, 3, 1, 3, 2)

d) (2, 4, 6)

Q4. Which operator is used to combine two tuples into a new tuple?

a) *

b) +

c) &

d) |

Q5. What will fruits.count("banana") return if fruits = ("apple", "banana", "cherry", "banana")?

a) 1

b) 2

c) 3

d) 0



7.7 SUMMING UP

- Tuples are created using parentheses () and can contain elements of any data type; a single-element tuple requires a trailing comma.
- Elements in a tuple are accessed using zero-based indexing, similar to lists.
- Tuples are immutable, meaning their elements cannot be changed after creation; however, new tuples can be created using concatenation or repetition.

- The + operator concatenates tuples, while the * operator repeats the elements of a tuple multiple times.
- Unlike lists, tuples are faster and more memory-efficient due to their immutability; lists are mutable and suitable for collections that may change.
- Tuples support limited methods such as count() to count occurrences and index() to find the first occurrence of an element.



7.8 ANSWERS TO CHECK YOUR PROGRESS

Ans.to.Q1: b)

Ans.to.Q2: c)

Ans.to.Q3: b)

Ans.to.Q4: b)

Ans.to.Q5: b)

7.9 REFERENCES

- Python Documentation: https://docs.python.org/3/tutorial/datastructures.html#tuples-and-sequences



7.10 FURTHER READINGS

- Kanetkar, Y., & Kanetkar, A. (2023). Let us Python. BPB Publications.
- https://www.w3schools.com/python/



7.11 MODEL QUESTIONS

1. Define a tuple in Python. Write examples of creating a tuple of integers, strings, a mixed tuple, and a single-element tuple.
2. Explain how to access elements in a tuple using indexing. Illustrate with an example using both positive and negative indices.
3. What is meant by immutability of tuples? Give examples to show that tuple elements cannot be modified directly.

4. Explain how tuples can be manipulated using concatenation and repetition, with suitable examples.
5. Differentiate between lists and tuples in terms of mutability, performance, and use cases.
6. Describe the common tuple methods count() and index() with examples.

****

UNIT 8: DICTIONARIES

UNIT STRUCTURE

8.1 Learning Objectives
8.2 Introduction
8.3 Dictionaries in Python
    8.3.1 Creating Dictionaries
8.4 Working with Dictionaries
    8.4.1 Accessing Dictionary Elements
    8.4.2 Adding or Updating Elements
    8.4.3 Removing Elements
8.5 Looping in Dictionaries
8.6 Dictionary Methods
8.7 Let Us Sum Up
8.8 Answers to Check Your Progress
8.9 Further Reading
8.10 Model Questions

8.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- learn what dictionaries are and their role in Python.
- learn to create dictionaries in Python in different ways.
- access dictionary keys and values efficiently
- perform common operations on dictionaries, including adding, updating, and removing elements.
- iterate through a dictionary
- use Python programs to solve problems involving dictionaries, nested dictionaries, dictionary methods.

8.2 INTRODUCTION

In the previous unit, we learned how to create and manipulate tuples in Python, examined the key differences between lists and tuples, and explored their practical applications in Python programs.

Now, we will turn our attention to another essential data structure in Python: dictionaries. Unlike sequences such as lists and tuples, which rely on index-based access, dictionaries store data in key-value pairs, allowing for efficient and intuitive data retrieval. This makes dictionaries particularly useful when working with structured data where each element has a unique identifier.

8.3 DICTIONARIES IN PYTHON

In Python, a dictionary is an unordered collection of key-value pairs, where each unique key maps to a corresponding value. Dictionaries are sometimes referred to as maps or associative arrays in other programming languages due to their ability to efficiently link keys with their associated values.

8.3.1 Creating Dictionaries

We can create a dictionary in several ways, either by directly assigning key-value pairs or using the dict() constructor.

¢ Creating a Dictionary using Curly Braces { }

A dictionary in Python is defined by enclosing key-value pairs within curly braces {}. Each key-value pair is separated by a colon, and pairs are separated by commas.

Syntax:

```
dictionary_name = {
   "key1": "value1",
   "key2": "value2",
   "key3": "value3"
}
```

To display the dictionary, we have to type print(dictionary_name) and press enter key. Some examples of dictionaries are:

```
d1= {'k1': 'Jitu', 'k2': 'Ritu', 'k3': 'Riki'}
print(d1)
```

Here, 'k1', 'k2', 'k3' are keys, whereas 'Rita', 'Raju', 'Rahul' are values.

- Different keys may have same values.

This is shown in the following example:

d3 = {"1":"A", "2": "A", "3": "F"} # will store all

- Keys in a dictionary must be unique and immutable. We can use numbers, strings, tuples as keys. If tuple is used as a key, it should not contain any mutable element like list. If the keys are same, but values are different, latest key value pair stored.

Here's an example of Python code that demonstrates how keys in a dictionary must be unique. If the same key is used multiple times with different values, only the last key-value pair will be stored:

```
# Creating a dictionary with duplicate keys
my_dict = {
    "name": "Manoj",
    "age": 25,
    "name": "Pallav"   # Duplicate key with a new value
}

# Displaying the dictionary
print(my_dict)
```

Output:

{'name': 'Pallav', 'age': 25}

In this example, the key "name" is duplicated. Initially, its value is "Manoj", but the second occurrence of "name" with the value "Pallav" overwrites the first one. Thus, the final value for the key "name" is "Pallav".

- If the key value pairs are repeated in a dictionary, only one pair will be stored, with the last occurrence of the key. Here's a simple example.

```
# Creating a dictionary with repeated key-value pairs
my_dict = {
    "name": "Rajib", "age": 25,
    "age": 30, # Repeated keys with a different value
    "city": "Guwahati"
}
print(my_dict)# Print the dictionary
```

Output:

```
{'name': 'Rajib', 'age': 30, 'city': 'Guwahati'}
```

In this example, the key "age" is repeated, but only the last occurrence ("age": 30) is stored in the dictionary, while the earlier value ("age": 25) is overwritten

¢ Creating a Dictionary using the dict() constructor

We can create a dictionary in several ways, either by directly assigning key-value pairs or using the dict() constructor.

```
person = dict(name="Smriti", age=30, city="New Delhi")
print(person)
```

Output:

```
{'name': 'Smriti', 'age': 30, 'city': New Delhi'}
```

¢ Creating an Empty Dictionary

In Python, an empty dictionary is a dictionary that contains no key-value pairs. It can be created in two ways:

" Using curly braces {}: d1 = {}

" Using the dict() constructor: d2 = dict()

Both methods are valid, but {} is more commonly used due to its simplicity and readability.

Example:

```
empty_dict = {}
```

```
print(type(empty_dict))      # Output: <class 'dict'>
print(empty_dict)        # Output: {}
```

8.4 WORKING WITH DICTIONARIES

Dictionaries in Python are mutable. This means we can change their contents without creating a new dictionary. We can add, remove, or modify key-value pairs after the dictionary has been created. Here are some characteristics of mutable dictionaries:

- We can add new key-value pairs.

```
my_dict = {'a': 1, 'b': 2}
my_dict['c'] = 3 # Adding a new key-value pair
```

- We can change the value associated with a specific key.

```
my_dict['a'] = 10 # Modifying the value of key 'a'
```

- We can remove key-value pairs.

```
del my_dict['b'] # Removing the key 'b'
```

Since dictionaries can be altered in these ways, so they are classified as mutable types in Python. Let us explore the manipulation of dictionaries in greater detail.

8.4.1 Accessing Dictionary Elements

In Python, dictionaries are not indexed by position. They are accessed by their keys, not by numerical indices like sets, lists or tuples. Thus, we can access elements/values in a dictionary by referring to the associated key. For example:

```
person = {"name": "Pritam", "age": 25, "city": "Mumbai"}
print(person["name"])
```

Output: Pritam

If we attempt to access a non-existent key, Python raises a KeyError. For example, if we write the command with a key "profession: which is not available in the "person" dictionary like print(person["profession"]) it will generate the KeyError.

To avoid this, we can use the get() method, which returns None or a default value if the key is not found.

print(person.get("profession", "Not Available"))

Output:

Not Available

Here are examples of Python code where a dictionary holds multiple values against a key using lists or tuples:

Example: Using Lists as Values

```
# Dictionary with lists as values
student_courses = {
    'Gautam': ['Math', 'Physics', 'Chemistry'],
    'Juri': ['History', 'Literature'],
    'Rahul': ['Biology', 'Math']
}
# Accessing the courses for Gautam
print(student_courses['Gautam'])
# Output: ['Math', 'Physics', 'Chemistry']
print(student_courses['Juri'])
# Output: ['History', 'Literature']
```

Example: Using Tuples as Values

```
# Dictionary with tuples as values
employee_info = {
    'David': ('Manager', 45000),
    'Rashmi': ('Developer', 60000),
    'Rajib': ('Analyst', 50000)
}
# Accessing tuple values for an employee
print(employee_info['Rashmi'])  # Output: ('Developer', 60000)
# Accessing a specific value within the tuple
print(employee_info['Rashmi'][1])  # Output: 60000
```

Example: Using a Combination of Lists and Tuples

```
# Dictionary with a combination of lists and tuples as values
company = {
   'HR': ['Smriti', 'Nayan', 'Pankaj'],
   'IT': [('Devid', 'Analyst'), ('Sara', 'Developer')]
}
# Accessing HR employees
print(company['HR']) # Output:['Smriti','Nayan','Pankaj']

# Accessing IT team members and their roles
print(company['IT'][0])  # Output: ('Devid', 'Analyst')
```

In these examples, the dictionary keys hold multiple values, either as lists or tuples, allowing multiple pieces of data to be associated with a single key.

Output screen:

8.4.2 Adding or Updating Elements

We can add a new key-value pair or update an existing key-value pair by assigning a value to a key. For example:

```
#Adding new key-value pair in the person dictionary
person["profession"] ="Manager"
#Updating existing value
person["age"] = 26
print(person)
```

Output:

```
{'name': 'Pritam', 'age': 26, 'city': 'Mumbai', 'profession': 'Manager'}
```

8.4.3 Removing Elements

We can remove elements from a dictionary using methods like pop(), del, and popitem().

- pop(): Removes a specific key and returns its value.
- del: Deletes a specific key-value pair.
- popitem(): Removes and returns the last inserted key-value pair

Removing an item with pop()

```
removed_value = person.pop("city")
print(removed_value)    # Output: Mumbai
```

It will remove the city "Mumbai" from the dictionary person. We can verify it by printing again with print(person) command.

Removing an item with del

```
del person["age"]
print(person)
```

Removing an item with popitem()

Here is an example of how to use the popitem() method in Python with a dictionary. In this example, the popitem() method removes the last inserted key-value pair ('Rajib': 90) and returns it as a tuple. The dictionary is then updated without the removed item.

```
# Create a dictionary
student_scores = {
    'Pritam': 85,
    'Ritam': 92,
    'Barun': 78,
    'Rajib': 90
}
print(student_scores)
# Use popitem() to remove the last inserted item
last_item = student_scores.popitem()
# Output the result
print("Popped item:", last_item)
print("Updated dictionary:", student_scores)
```

Output:

```
{'Pritam': 85, 'Ritam': 92, 'Barun': 78, 'Rajib': 90}
Popped item: ('Rajib', 90)
Updated dictionary: {'Pritam': 85, 'Ritam': 92, 'Barun': 78}
```

Let us take another example to first create and then apply add/delete/modify operation on a dictionary.

```
subjects = {"s1": "Java", "s2": "C++", "s3": "Python", "s4": "Web Technology"}
print(subjects)
subjects["s5"]="DBMS"    #add new key-vale pair
print(subjects)
subjects["s2"] ="Operating System" #modify value for a key s2
print(subjects)
del(subjects["s1"]) #delete a key-value pair "s1": "Java"
print(subjects)
del(subjects) #delete dictionary objects
```

Checking if a Key Exists: We can check whether a specific key exists in a dictionary using the in keyword. For example:

```
if "age" in person:
    print("Age is present in the dictionary.")
```



CHECK YOUR PROGRESS

Q1. State true or false:

i. The very first key-value pair in a dictionary d can be accessed using the expression d[0].

ii. Dictionaries are immutable.

iii. Dictionary elements can be accessed using position-based index.

iv. It is possible to hold multiple values against a key in a dictionary.

v. Unlike sets, dictionaries preserve insertion order.

8.5 LOOPING IN DICTIONARIES

Like strings, lists, tuples and sets, dictionaries can also be iterated over using for loop. For iteration, there are three ways:

Looping through keys

Looping though values

Looping through key-value pairs

We can loop through a dictionary's keys, values, or key-value pairs. If we use a dictionary as an iterator (e.g. in a for statement), it traverses the keys of the dictionary. For example:

```
d = {'a': 1, 'b': 2, 'c':3}
for key in d:
    print(key, d[key])
# c 3
# b 2
# a 1
# Looping through keys
person = {"name": "Pritam", "age": 25, "city": "Mumbai"}
for key in person:
   print(key)

# Looping through values
for value in person.values():
   print(value)

# Looping through key-value pairs
for key, value in person.items():
   print(f"{key}: {value}")
```

Let's provide a complete example of how to loop through a dictionary in Python to access both keys and values.

```
# Define a dictionary
student_scores = {
   "Gautam": 85,
   "Rajib": 90,
   "Megha": 78,
   "Rashmi": 92
}
```

```
# Loop through the dictionary
for k, v in student_scores.items():
    print(f"{k} scored {v} marks.")
```

Output:

```
Gautam scored 85 marks.
Rajib scored 90 marks.
Megha scored 78 marks.
Rashmi scored 92 marks.
```

In the Python code,

print(f"{student} scored {score} marks.")

the f before the string denotes an f-string (formatted string literal), which is a way to embed expressions inside string literals for formatting. f-strings make it easy to insert variable values or expressions directly into a string. The part inside the curly braces {k} and {v} refers to the variable student and score. Python evaluates these expressions and inserts their values into the string.

- We can merge two dictionaries by unpacking them with **. If we use *, only the keys will be unpacked. As the merging takes place, duplicates get overwritten from left to right.

  For instance:

```
fruits= {"Mango":20, "Orange": 30, "apple":50}
flowers= {"Lotus":10, "Rose":20, "Sun flower":20}
mix={**fruits, **flowers}
print(mix)
```

Output:

```
{'Mango': 20, 'Orange': 30, 'apple': 50, 'Lotus': 10, 'Rose': 20, 'Sun flower':
20}
```

  The dictionaries are merged in the order listed in the expression.

- Dictionaries can be nested. Here are some examples of nested dictionaries in Python:

Example of simple Nested Dictionary

```
# Creating a simple nested dictionary
student = {
    'name': 'Bibek',
    'age': 22,
    'subjects': {'Math': 85, 'Physics': 92, 'Chemistry': 88 }
}
# Accessing nested dictionary elements
print(student['name'])              # Output: Bibek
print(student['subjects']['Math'])  # Output: 85
print(student)
#output: {'name': 'Bibek', 'age': 22, 'subjects': {'Math': 85, 'Physics': 92, 'Chemistry': 88}}
```

In this example, inside the dictionary student, there is one more dictionary subjects.

Example of Nested Dictionary with Multiple Levels

```
# Creating a nested dictionary with multiple levels
company = {
    'department1': {
        'manager': 'Kunal',
        'employees': ['Rashmi', 'Rajib', 'Varun'],
        'budget': 100000
    },
    'department2': {
        'manager': 'Bhabna',
        'employees': ['Devid', 'Eva'],
        'budget': 80000
    }
}
# Accessing nested elements
print(company['department1']['manager'])        # Output: Kunal
print(company['department2']['employees'][1])   # Output: Eva
```

```
print(company['department2']['employees'][0])    # Output: Devid
```

This Python code defines a nested dictionary representing a company with multiple departments, each containing specific details like the manager, employees, and budget. The outer dictionary company has two keys: 'department1' and 'department2', representing different departments. For accessing nested elements:

- company['department1']['manager'] : Retrieves the manager of department1, which is "Kunal".
- company['department2']['employees'][1]: Accesses the second employee (index 1) in department2's employee list, which is "Eva".
- company['department2']['employees'][0]: Accesses the first employee (index 0) in department2's employee list, which is "Devid".

8.6 DICTIONARY METHODS

Python provides a variety of built-in dictionary methods that enable efficient manipulation of key-value pairs. These methods simplify common operations such as accessing, adding, updating, and removing elements, making dictionaries a highly flexible and powerful data structure.
Here is a list of commonly used dictionary methods in Python, along with examples:

dict.clear(): Removes all items from the dictionary.

```
d = {'a': 1, 'b': 2}
d.clear()
print(d)  # Output: {}
```

dict.copy(): Returns a shallow copy of the dictionary.

```
d = {'a': 1, 'b': 2}
copy_d = d.copy()
print(copy_d)  # Output: {'a': 1, 'b': 2}
```

dict.get(key[, default]): Returns the value for key if key is in the dictionary,

else returns default (or None).

```
d = {'a': 1}
print(d.get('a'))       # Output: 1
print(d.get('b', 0))    # Output: 0
```

dict.items(): Returns a view object of key-value pairs.

```
d = {'a': 1, 'b': 2}
print(d.items())#output: dict_items([('a', 1), ('b', 2)])
```

dict.keys(): Returns a view object of dictionary keys.

```
d = {'a': 1, 'b': 2}
print(d.keys())  # Output: dict_keys(['a', 'b'])
```

dict.values(): Returns a view object of dictionary values.

```
d = {'a': 1, 'b': 2}
print(d.values())  # Output: dict_values([1, 2])
```

dict.pop(key[, default]): Removes and returns the value for the given key. Returns default if key is not found.

```
d = {'a': 1, 'b': 2}
print(d.pop('a'))  # Output: 1
print(d)           # Output: {'b': 2}
```

dict.popitem(): Removes and returns the last inserted key-value pair (LIFO order in Python 3.7+).

```
d = {'a': 1, 'b': 2}
print(d.popitem())  # Output: ('b', 2)
```

dict.update([other]): Updates the dictionary with key-value pairs from another dictionary or iterable.

```
d = {'a': 1}
d.update({'b': 2})
print(d)  # Output: {'a': 1, 'b': 2}
```

dict.setdefault(key[, default]): Returns the value of key. If key is not present, inserts it with default value.

```
d = {'a': 1}
d.setdefault('b', 2)
print(d)  # Output: {'a': 1, 'b': 2}
```



CHECK YOUR PROGRESS

Q2. Answer the following questions:

i. How are elements in a dictionary accessed?

ii. What method can be used to remove a specific key-value pair in a dictionary?

iii. What is the output of an empty dictionary?

iv. Can a dictionary key be of any data type?

v. What does the del keyword do in a dictionary?

vi. How can you check if a key exists in a dictionary?

vii. What happens if a key appears multiple times in a dictionary?

viii. What does the popitem() method do?

Q3. Choose the correct option:

i. Which of the following is true about Python dictionaries?

a) Keys must be unique

b) Values must be unique

c) Both keys and values must be unique

d) Neither keys nor values must be unique

ii. What will the following code return?

```
d = {"key1": "value1", "key2": "value2", "key1": "value3"}
print(d)
```

a) {'key1': 'value1', 'key2': 'value2'}

b) {'key1': 'value3', 'key2': 'value2'}

c) {'key1': 'value3', 'key2': 'value3'}

d) {'key1': 'value1', 'key2': 'value3'}

iii. Which method is used to safely access dictionary elements when a key may not exist?

a) get()

b) pop()

c) del

d) index()

iv. How can you add a new key-value pair to an existing dictionary in Python?

a) Use the append() method

b) Use the insert() method

c) Assign a value to a new key

d) Use the pop() method

v. What is the output of the following code?

```
d = {"a": 1, "b": 2}
d["c"] = 3
print(d)
```

a) {"a": 1, "b": 2}

b) {"a": 1, "b": 2, "c": 3}

c) {"a": 1, "b": 3, "c": 3}

d) {"a": 3, "b": 2, "c": 1}

vi. What does the popitem() method do in a dictionary?

a) Removes the first inserted item

b) Removes the last inserted item

c) Removes a specific key-value pair

d) Removes all key-value pairs

vii. Which statement is true about the get() method in dictionaries?

a) It raises an error if the key doesn't exist.

b) It returns a default value if the key doesn't exist.

c) It removes the key-value pair.

d) It adds a new key-value pair if the key doesn't exist.

viii. How do you remove an element with a specific key from a dictionary?

a) Using del

b) Using remove()

c) Using index()

d) Using sort()

ix. Which method returns a list of all values in a dictionary?

a) keys()

b) items()

c) values()

d) get()

x. Which of the following is NOT a valid way to create a dictionary in Python?

a) d = {}

b) d = dict()

c) d = {"a": 1, "b": 2}

d) d = list()



8.7 LET US SUM UP

This unit focuses on the basics of Python dictionaries, including their structure and essential operations for effective data handling. Here's a summary of what we learned from this unit:

- Dictionaries in Python is a collection of key-value pairs, where each key maps to a specific value.
- Dictionaries can be created/defined with key-value pairs enclosed in {}.
  Example: dictionary_name = {"key1": "value1", "key2": "value2"}
- Dictionaries can also be created/defined using the dict() constructor:
  Example: person = dict(name="Smriti", age=30)
- We can access elements/values in a dictionary by referring to the associated key.
- Dictionary elements are not position indexed, but key indexed.
- Keys must be unique, whereas values can be of any data type.
- Unlike tuples or lists, dictionaries use keys rather than index positions for data storage.
- Dictionaries in Python are mutable. This means we can modify them after they are created by adding new key-value pairs, updating existing values, removing key-value pairs, changing the order or structure

of the dictionary.

- If the same key is used multiple times, the last value assigned is retained.
- Working with Dictionaries: We can add, remove, or modify key-value pairs after creation due to its mutable characteristics.
- Accessing Elements: We can access elements using keys.
- Values in Dictionaries can be lists, tuples, or other dictionaries.
- Adding or Updating Elements: New pairs added or existing pairs updated via assignment:

Example: person["profession"] = "Manager"

- Removing Elements: We can remove elements in a dictionary using methods: pop(), del, and popitem().
- Looping in Dictionaries: Iteration can be done through keys, values, or key-value pairs using for loops.
- Dictionary Operations: We cannot combine dictionaries with + or compare them with < or >.
- Dictionaries can be merged using unpacking.
- Nested Dictionaries: Dictionaries can contain other dictionaries.
- The unit also discussed dictionary methods that help in working with dictionaries efficiently.



8.8 ANSWERS TO CHECK YOUR PROGRESS

Ans.to Q.No.1:

i. False.

ii. False.

iii. False.

iv. True.

v. True.

Ans.to Q. No. 2:

i. Dictionary elements are accessed by their keys.

ii. The pop() method.

iii. {} (an empty dictionary).

iv. No, keys must be of an immutable data type.

v. It deletes a key-value pair.

vi. Using the in keyword.

vii. Only the last occurrence is stored, and the previous ones are overwritten.

viii. It removes and returns the last inserted key-value pair.

Ans. to. Q.No.3:

i. a) Keys must be unique

ii. b) {'key1': 'value3', 'key2': 'value2'}

iii. a) get()

iv. c) Assign a value to a new key

v. b) {"a": 1, "b": 2, "c": 3}

vi. b) Removes the last inserted item

vii. b) It returns a default value if the key doesn't exist.

viii. a) Using del

ix. c) values()

x. d) d = list()



8.9 FURTHER READING

1. Rao, N. (2024). Core Python programming: PyCharm and Jupyter. Dreamtech Press.
2. Kanetkar, Y., & Kanetkar, A. (2020). Let Us Python: Python Is Future, Embrace It Fast (Vol. 2). BPB Publications.
3. Lee, K. D. (2011). Python programming fundamentals. London: Springer.



8.10 MODEL QUESTIONS

1. What is a dictionary in Python, and how does it differ from other data structures like lists?
2. Write a Python program to create a dictionary with the following key-value pairs: "name": "Anuj", "age": 25, "city": "Mumbaii". Add a new

key-value pair for "profession" and update the age to 26.

3. What are the differences between using indexing and key-based access?
4. Explain with an example how you can check if a key exists in a dictionary.
5. Write a Python program to iterate over a dictionary's keys and values.
6. There are two dictionaries called students and players containing names as keys ang ages as values. Write the Python code to merge the two dictionaries into a third dictionary.
7. Discuss the characteristics of mutable types in Python, specifically in relation to dictionaries.
8. Provide examples demonstrating how dictionaries can be modified after creation, including adding new key-value pairs and updating existing ones.
9. Explain how to check for the existence of a key in a dictionary. What is the significance of using the in keyword?
10. Define nested dictionaries and explain how they can be created and accessed in Python. Provide an example of a nested dictionary.
11. Describe the process of iterating through dictionaries in Python.
12. Describe the importance of unique keys in a Python dictionary. What happens when duplicate keys are present? Provide code examples to demonstrate how Python handles duplicate keys in dictionaries.

****

UNIT 9: SETS

UNIT STRUCTURE

9.1 Learning Objectives

9.2 Introduction

9.3 Creating Sets

9.4 Manipulating Sets

9.5 Set Methods

9.6 Frozen Sets

9.7 Summing Up

9.8 Answers to Check Your Progress

9.9 References

9.10 Further Readings

9.11 Model Questions

9.1 LEARNING OBJECTIVES

By the end of this unit, you will be able to:

- Understand the concept of sets and their role in Python programming.
- Create and access sets in Python.
- Manipulate sets by adding, removing, and performing set operations.
- Use common set methods to perform various operations on sets.
- Understand the concept of frozen sets and their use cases.

9.2 INTRODUCTION

Sets are a powerful and versatile data structure in Python used to store unique elements. Unlike lists or tuples, sets do not allow duplicate values, and they are unordered, meaning the elements do not have a specific position. Sets are particularly useful for performing mathematical operations like unions, intersections, and differences. In this unit, we will explore how to create and manipulate sets, understand their properties, and use common set methods.

9.3 CREATING SETS

Sets are created by placing elements inside curly braces {} or by using the set() constructor. Sets can contain elements of any type, but they must be immutable (e.g., numbers, strings, tuples).

Example 1:

```
# Creating a set of integers
numbers = {1, 2, 3, 4, 5}

# Creating a set of strings
fruits = {"apple", "banana", "cherry"}

# Creating a set using the set() constructor
mixed_set = set([1, "apple", 3.14, True])

print(numbers)
print(fruits)
print(mixed_set)
```

Output:

```
{1, 2, 3, 4, 5}
{'banana', 'apple', 'cherry'}
{1, 3.14, 'apple'}
```

Explanation:

- Sets are unordered, so the order of elements may vary when printed.
- Sets do not allow duplicate values; duplicates are automatically removed.

Example 2:

```
# Creating a set with duplicate elements
```

```
duplicates = {1, 2, 2, 3, 3, 3}
print(duplicates)
```

Output:

```
{1, 2, 3}
```

Explanation:

- The set automatically removes duplicate values, so only unique elements are stored.

Accessing Set Elements

Since sets are unordered, you cannot access elements using indices like you do with lists or tuples. However, you can check if an element is present in a set using the in keyword.

Example 3:

```
fruits = {"apple", "banana", "cherry"}

# Checking if an element is in the set
print("banana" in fruits)
print("orange" in fruits)
```

Output:

```
True
False
```

Explanation:

- The in keyword checks if an element is present in the set and returns True or False.

Example 4:

```
# Iterating over a set
```

```
for fruit in fruits:
    print(fruit)
```

Output:

```
banana
apple
cherry
```

Explanation:

- You can iterate over a set using a for loop, but the order of elements is not guaranteed.

9.4 MANIPULATING SETS

(a) Adding Elements

You can add elements to a set using the add() method or multiple elements using the update() method.

Example 1:

```
fruits = {"apple", "banana"}

# Adding a single element
fruits.add("cherry")
print(fruits)

# Adding multiple elements
fruits.update(["orange", "kiwi"])
print(fruits)
```

Output:

```
{'banana', 'apple', 'cherry'}
```

{'banana', 'apple', 'cherry', 'orange', 'kiwi'}

Explanation:

- add() adds a single element to the set.
- update() adds multiple elements to the set.

Example 2:

```
# Adding a tuple to a set
fruits.add(("grape", "melon"))
print(fruits)
```

Output:

{'banana', 'apple', 'cherry', 'orange', 'kiwi', ('grape', 'melon')}

Explanation:

- You can add immutable elements like tuples to a set.

(b) Removing Elements

You can remove elements from a set using methods like remove(), discard(), and pop().

Example 3:

```
fruits = {"apple", "banana", "cherry", "orange"}

# Removing a specific element
fruits.remove("banana")
print(fruits)

# Removing an element (no error if not found)
fruits.discard("kiwi")
print(fruits)
```

```
# Removing and returning an arbitrary element
removed_element = fruits.pop()
print("Removed element:", removed_element)
print(fruits)
```

Output:

```
{'apple', 'cherry', 'orange'}
{'apple', 'cherry', 'orange'}
Removed element: apple
{'cherry', 'orange'}
```

Explanation:

- remove() removes the specified element and raises an error if the element is not found.
- discard() removes the specified element but does not raise an error if the element is not found.
- pop() removes and returns an arbitrary element from the set.

Example 4:

```
# Clearing all elements from the set
fruits.clear()
print("Cleared set:", fruits)
```

Output:

```
Cleared set: set()
```

Explanation:

- clear() removes all elements from the set.

(c) Set Operations

Sets support mathematical operations like union, intersection, difference, and symmetric difference.

Example 5:

```
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}

# Union of sets
print("Union:", set1 | set2)

# Intersection of sets
print("Intersection:", set1 & set2)

# Difference of sets
print("Difference:", set1 - set2)

# Symmetric difference of sets
print("Symmetric Difference:", set1 ^ set2)
```

Output:

```
Union: {1, 2, 3, 4, 5, 6}
Intersection: {3, 4}
Difference: {1, 2}
Symmetric Difference: {1, 2, 5, 6}
```

Explanation:

- The union (|) combines elements from both sets.
- The intersection (&) includes only elements present in both sets.
- The difference (-) includes elements present in the first set but not in the second.
- The symmetric difference (^) includes elements present in either set but not in both.

Example 6:

```
# Checking if one set is a subset of another
print("Is set1 a subset of set2?", set1 <= set2)

# Checking if one set is a superset of another
print("Is set1 a superset of set2?", set1 >= set2)
```

Output:

```
Is set1 a subset of set2? False
Is set1 a superset of set2? False
```

Explanation:

- <= checks if one set is a subset of another.
- >= checks if one set is a superset of another.



CHECK YOUR PROGRESS

Choose the correct option:

1. What is the primary characteristic of a set in Python?
   a) It allows duplicate elements.
   b) It is an ordered collection of elements.
   c) It stores only unique elements and is unordered.
   d) It can only store numeric values.
2. Which of the following methods is used to add a single element to a set?
   a) update()
   b) append()
   c) add()
   d) insert()

3. What will be the output of the following code?

   my_set = {1, 2, 3}

   my_set.add(4)

   print(my_set)

   a) {1, 2, 3}

   b) {1, 2, 3, 4}

   c) {4, 1, 2, 3}

   d) {1, 2, 3, 4, 4}

4. Which of the following operations returns the elements present in either of the two sets but not in both?

   a) Union (|)

   b) Intersection (&)

   c) Difference (-)

   d) Symmetric Difference (^)

5. What is the purpose of the discard() method in sets?

   a) It removes and returns an arbitrary element from the set.

   b) It removes a specific element and raises an error if the element is not found.

   c) It removes a specific element without raising an error if the element is not found.

   d) It adds a new element to the set.

9.5 SET METHODS

Common Set Methods

Python provides several built-in methods to manipulate sets.

Example 1:

```
fruits = {"apple", "banana", "cherry"}

# Finding the length of the set
print("Length:", len(fruits))
```

```
# Copying a set
fruits_copy = fruits.copy()
print("Copy of fruits:", fruits_copy)
```

Output:

```
Length: 3
Copy of fruits: {'banana', 'apple', 'cherry'}
```

Explanation:

- len() returns the number of elements in the set.
- copy() creates a shallow copy of the set.

Example 2:

```
# Checking if two sets are disjoint
set1 = {1, 2, 3}
set2 = {4, 5, 6}
print("Are set1 and set2 disjoint?", set1.isdisjoint(set2))
```

Output:

```
Are set1 and set2 disjoint? True
```

9.6 FROZEN SETS

Frozen sets are immutable versions of sets. Once created, their elements cannot be changed. Frozen sets are useful when you need an immutable collection of unique elements.

Example 1:

```
# Creating a frozen set
frozen_fruits = frozenset(["apple", "banana", "cherry"])
```

```
# Attempting to modify a frozen set will raise an error
# frozen_fruits.add("orange")  # This will raise an AttributeError

print(frozen_fruits)
```

Output:

```
frozenset({'banana', 'apple', 'cherry'})
```

Explanation:

- Frozen sets are immutable, so you cannot add or remove elements after creation.

Example 2:

```
# Performing set operations on frozen sets
frozen_set1 = frozenset([1, 2, 3])
frozen_set2 = frozenset([3, 4, 5])

# Union of frozen sets
print("Union:", frozen_set1 | frozen_set2)
```

Output:

```
Union: frozenset({1, 2, 3, 4, 5})
```

Explanation:

- Frozen sets support set operations like union, intersection, and difference.



9.7 SUMMING UP

- Sets are created using {} or set(), store only immutable elements, and remove duplicates automatically.
- They are unordered, cannot be indexed, but allow membership checks with in and iteration using loops.

- Elements can be added with add() for one item, update() for many, and immutable types like tuples can be included.
- Elements can be removed with remove(), discard(), pop(), or all cleared with clear().
- Common operations include union, intersection, difference, symmetric difference, subset, and superset checks.
- Frozen sets are immutable collections that cannot be changed but support standard set operations.



9.8 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q1: c)

Ans. to Q2: c)

Ans. to Q3: b)

Ans. to Q4: d)

Ans. to Q5: c)

9.9 REFERENCES

- Python Documentation: https://docs.python.org/3/tutorial/datastructures.html



9.10 FURTHER READINGS

- Kanetkar, Y., & Kanetkar, A. (2023). Let us Python. BPB Publications.
- https://www.w3schools.com/python/



9.11 MODEL QUESTIONS

1. Explain how sets are created in Python with suitable examples.
2. Describe the properties of sets that distinguish them from lists and tuples.
3. What happens when duplicate elements are added to a set? Illustrate with an example.

4. Explain how membership testing in sets is performed using the in keyword with examples.
5. Write a short note on adding elements to a set using add() and update() methods. Provide examples.
6. Differentiate between remove(), discard(), and pop() methods of sets with examples.
7. Discuss set operations such as union, intersection, difference, and symmetric difference with suitable code examples.
8. Explain how subset and superset relationships are checked in sets. Provide examples.
9. Write a note on common set methods like len(), copy(), and isdisjoint() with examples.
10. What are frozen sets in Python? Explain their characteristics and operations with examples.

****

UNIT 10: FILE HANDLING

UNIT STRUCTURE

10.1 Learning Objectives

10.2 Introduction

10.3 File Handling

10.4 Opening and Closing Text Files

10.5 Reading and Writing Text Files

10.6 Handling Errors with Files in Python

10.7 Let Us Sum Up

10.8 Answers to Check Your Progress

10.9 Further Reading

10.10 Model Questions

10.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- explain file handling
- describe the different types of files in Python
- open a file using different modes
- read and write files in Python
- use "with" statement in Python
- use seek() and tell() methods in Python programs
- handle errors related to file handling

10.2 INTRODUCTION

In this unit, we will cover the fundamentals of file handling. In the last unit, we had covered the concept of sets in Python. In this unit, we will cover how to read and write text files in Python. We will also cover how to use the with statement along with functions like readline(). Methods like seek() and tell() are also covered in detail in this unit. Additionally, the unit covers how to handle errors related to file handling in Python. In the next unit, we will explore the concept of object oriented programming in Python.

10.3 FILE HANDLING

File handling refers to the process of performing operations on a file. File handling in Python involves various operations such as creation of file, writing in file, reading of file and closing the file. These operations are essential for working with data stored in a file.

There are two different types of files in Python. They are:

- Text files: Text files store the data in the form of characters.
- Binary files: Binary files store the data in the form of bytes.

In this unit, we will look into how to perform various operations on text files. Python provides built-in functions to access the files. Let us look at few of these functions.

10.4 OPENING AND CLOSING TEXT FILES

Opening a text file:

The open() function is used to open a file. The open() function returns a file object or file handler which points to the beginning of the file. This function is most commonly used with two positional arguments and one keyword argument. The syntax for this function is given below.

Syntax:

f = open("file_name", "mode", encoding=None)

The first argument represents the name of the file on which the data is stored. The second argument represents the purpose of opening the file. The table 10.1 below specifies some of the different file open modes and their meanings.

Table 10.1: File Open Modes

File open mode	Description
w	Opens the file to write data into file. Creates a new file or data in previous file is deleted.
r	Reads the data in the file.
a	Appends data to the file
w+	Opens the file to write and read data of a file. The previous data in the file will be deleted.

r+	Opens the file for both reading and writing. The previous data in the file will not be deleted.
a+	Opens file to append and read data from a file.
x	Opens file in exclusive creation mode. Returns an error if file already exists.

The encoding parameter is used to specify how the text in the file should be decoded or encoded. This should only be used in text mode. If the encoding is not specified, then Python uses the default encoding of the system.

Closing a text file:

The close() function can be used to close a file that has been opened. If the file is not closed, then the memory utilized by the file is not freed, so it is mandatory to close a file. The syntax of close() is given below.

Syntax:

f.close()

In the syntax, the file is represented by the file object "f". This means that the "f" is deleted from memory, and if we want to access the file now, then we will have to open the file again using open() function.

Let us now look at how we can write and read data from a file.

10.5 READING AND WRITING TEXT FILES

The write() function is used to store a character or a group of characters into a file by using the file object. Let us look at a program to write few words into a text file.

Example 1: Program to write into a text file

```
f = open("myfile.txt",'w')
str = input("Enter text: ")
f.write(str)
f.close()
```

Output:

```
C:\Users\Shruti\Desktop\Python_Programs>python file_write.py
Enter text: Hello, we are learining Python
```

In the above program 10.1, the first line opens a file name "myfile.txt" in write mode which can now be accessed by the file object "f". In the second

line, input is taken from the user and is stored in the variable "str". The write() function is then used to write the user input into the file. The close() function is then used to close the file.

The read() function is used to read data from a file in Python. It is a built-in method available for file objects. Let us now look at a program to read words from the text file "myfile.txt".

Program 10.2: Program to read from a text file

```
f = open("myfile.txt",'r')
str = f.read()
print(str)
f.close()
```

Output:

```
C:\Users\Shruti\Desktop\Python_Programs>python file_read.py
Hello, we are learining Python
```

If we want to store a group of strings or sentences then we have to use the write() function inside a loop. For example, the program 10.3 takes input from the user as long as the user does not type @ symbol.

Program 10.3: Program to store group of strings in a file

```
f = open("file1.txt",'w')
print("Enter text (@ at end): ")
while str!='@':
    str = input()
    if(str != '@'):
        f.write(str+"\n")
f.close()
```

Output:

```
C:\Users\Shruti\Desktop\Python_Programs>python file_write_sent.py
Enter text (@ at end):
Hello
Welcome to KKHSOU
This course is on Python Programming
@
```

To read the sentences from the file "file1.txt", we can use the read() function as shown in program 10.4.

Program 10.4: Program to read from a text file

```
f = open("file1.txt",'r')
print("The contents of file1.txt are: ")
str = f.read()
print(str)
f.close()
```

Output:

```
C:\Users\Shruti\Desktop\Python_Programs>python file_read_sent.py
The contents of file1.txt are:
Hello
Welcome to KKHSOU
This course is on Python Programming
```

We can use also use another method readline() to read single line from the file at a time. The function returns the line as a string and moves the pointer to the next line. If there are no more lines in the file, then it returns an empty string. This function is memory efficient since it only loads one line at a time. Let us look at a program to understand how readline() function works.

Program 10.5: Program to demonstrate readline() function

```
f = open("file1.txt",'r')
line = f.readline()
while line:
    print(line.strip())
    line = f.readline()
```

The program opens a file named "file1.txt" in read mode ('r') and then reads the first line from the file using readline(). A while loop is used to process the file line by line until there are no more lines to read. Inside the loop, strip() function is used to remove any newline characters, and the cleaned line is then printed. The next line is then read using readline() to continue the loop. The loop ends when all lines have been read and line becomes empty.

Output:

```
C:\Users\Shruti\Desktop\Python_Programs>python readline.py
Hello
Welcome to KKHSOU
This course is on Python Programming
```

Let us now see in program 10.6 how we can use the "a" (append) mode to add new lines to an existing file.

Program 10.6: Program to demonstrate append mode of file

```
f = open("file1.txt",'a')
print("Enter text (@ at end): ")
while str!='@':
    str = input()
    if(str != '@'):
        f.write(str+"\n")
f.close()
```

Output:

```
C:\Users\Shruti\Desktop\Python_Programs>python file_append.py
Enter text (@ at end):
These are additional lines
which are added at the end of the file.
@
```

It can be seen form the output that we have added two additional lines to the file "file1.txt". When we run the program 10.5 where we read all the lines of the file "file1.txt", we can see from the next part of the output that the newly added two lines have been added to the file after the first three lines and this can be seen in the output.

```
C:\Users\Shruti\Desktop\Python_Programs>python readline.py
Hello
Welcome to KKHSOU
This course is on Python Programming
These are additional lines
which are added at the end of the file.
```

The with statement:

The "with" statement can be used while opening a file. The advantage of

with statement is that it will take care of closing of file which is opened by it. So, we do not need to close the file explicitly. In cases of exceptions also, the 'with' statement will close the file before the exception is handled. The syntax for using 'with' statement is given below.

Syntax:

```
with open("filename", "openmode") as file_object:
```

Let us look at an example to understand how the 'with' statement works.

Program 10.7: Program to demonstrate the use of with statement to write in a file

```
with open("newfile.txt","w") as f:
    f.write("This file demonstrate the use of with statement\n")
    f.write("in Python Programming.")
```

Output:

```
C:\Users\Shruti\Desktop\Python_Programs>python with_statement.py
```

In program 10.7, we opened a file called "newfile.txt" and stored two lines of text. This file is then opened using 'with' statement in program 10.8 and the contents of the file "newfile.txt" are displayed on the screen.

Program 10.8: Program to demonstrate the use of with statement to read from a file

```
with open("newfile.txt","r") as f:
    for line in f:
        print(line)
```

Output:

```
C:\Users\Shruti\Desktop\Python_Programs>python with_statement_read.py
This file demonstrate the use of with statement

in Python Programming.
```

The seek() and tell():

The tell() method can be used to get the position of the file pointer. It returns the current position of the file pointer from the beginning of the file. It

can be used as shown below:

n = f.tell()

where, 'f' is the file object and 'n' is an integer that represents the byte position.

The seek() method is used to change the file object's position. The syntax for using seek() is:

f.seek(offset,whence)

Here, 'f' is the file object. The position is calculated by adding the "offset" to a reference point and the reference point is selected by the whence argument. When the value of whence argument is 0 it measures from the beginning of the file. When the value of whence argument is 1 it uses the current position and a when the whence argument is 2 then it measures from the end of the file. The default option for whence is 0 which means it uses the beginning of the file as reference point.

Let us look at an example to understand how these methods work.

Program 10.9: Program to demonstrate the use of seek() and tell() methods

```
with open("files.txt","w+") as f:
    f.write("This file demonstrate the use of seek()\n")
    f.write("and tell() methods in Python Programming.")
    n=f.tell()
    print("The pointer is at position: ",n)
    f.seek(10)
    print("After seeking the pointer is now at position: ",f.tell())
    print("The character at this position is: ",f.read(1))
    f.seek(n-12)
    print("After seeking from the end the pointer is now at position: ",f.tell())
    print("The characters at this position are: ",f.read(12))
```

The program 10.9 opens a file named "files.txt" in text mode with read/write access. The mode "w+" clears the file if it already exists and allows both writing and reading. Two lines of text are written to the file using the f.write() method. After these two lines are written, the file pointer is at the end of the file. Since, f.tell() returns the current file pointer position so the first time

f.tell() is called, it shows how far we are or how many characters have been written to the file. The line f.seek(10) moves the file pointer to character position 10 from the beginning and f.read(1) reads 1 character from that position i.e. "d" and prints it. In the next line, f.seek(n-12) moves the pointer to 12 characters before the end of the file and f.read(12) reads the last 12 characters from the file i.e. "Programming." And prints it.

Output:

```
C:\Users\Shruti\Desktop\Python_Programs>python seek_tell.py
The pointer is at position:  82
After seeking the pointer is now at position:  10
The character at this position is:  d
After seeking from the end the pointer is now at position:  70
The characters at this position are:  Programming.
```

10.6 HANDLING ERRORS WITH FILES IN PYTHON

Sometimes we encounter errors when working with files. Errors such as file does not exist or doesn't have read or write permissions are some of the common errors in file handling in Python. To make the program more reliable, it is important to handle these errors gracefully using exception handling.

Some typical errors that may occur during file operations are listed below:

- FileNotFoundError: This error occurs when we try to read a file that doesn't exist.
- PermissionError: This error occurs when the program does not have the permission to access the file.
- IsADirectoryError: This error occurs if a directory is used in place of a file.
- UnicodeDecodeError / UnicodeEncodeError: This error occurs when there's a problem with the encoding or decoding of the file content.

Let us now look at a program to handle these errors. The program uses try-except block to catch and handle errors. We will learn more about exception handling in Unit 12.

Program 10 10: Program to demonstrate error handling in files

```
try:
    with open("new_file.txt", "r", encoding="utf-8") as f:
        data = f.read()
        print(data)
except FileNotFoundError:
    print("The file was not found.")
except PermissionError:
    print("You do not have permission to access this file.")
except UnicodeDecodeError:
    print("There was an encoding problem while reading the file.")
except Exception as e:
    print("An unexpected error occurred:", e)
```

Output:

```
C:\Users\Shruti\Desktop\Python_Programs>python error_handling.py
The file was not found.
```

The program 10.10 tries to open a file named "new_file.txt" in read mode with UTF-8 encoding. If the file exists and is readable, it reads the entire content and prints it. If the file doesn't exist, a FileNotFoundError is caught, and a message is printed. If there are permission issues, a PermissionError is handled. If there's a problem with text encoding, it catches a UnicodeDecodeError. Any other unexpected errors are caught by the general Exception block and displayed.



CHECK YOUR PROGRESS

Q1. What are the different types of files in Python?

Q2. Which file mode represents exclusive creation mode for open function?

Q3. What is the purpose of the with statement?

Q4. What is the function of tell() method?



10.6 LET US SUM UP

- File handling refers to the process of performing operations on a file.
- Text files store the data in the form of characters.
- Binary files store the data in the form of bytes.
- The open() function is used to open a file and it returns a file object or file handler which points to the beginning of the file.
- The close() function can be used to close a file that has been opened.
- The write() function is used to store a character or a group of characters into a file by using the file object.
- The readline() function is used to read single line from the file at a time.
- The "with" statement can be used to take care of closing of file which is opened by it.
- The tell() method can be used to get the position of the file pointer.
- The seek() method is used to change the file object's position.



10.7 ANSWERS TO CHECK YOUR PROGRESS

Ans to Q1: There are two different types of files in Python: text files and binary files.

Ans to Q2: The "x" file mode represents exclusive creation mode for open() function.

Ans to Q3: When the "with" statement is used while it takes care of closing of file which is opened by it. So, we do not need to close the file explicitly.

Ans to Q4: The tell() method returns the current position of the file pointer from the beginning of the file.



10.8 FURTHER READING

1) Kanetkar, Y., & Kanetkar, A. (2020). Let Us Python: Python Is Future, Embrace It Fast (Vol. 2). BPB Publications.



10.9 MODEL QUESTIONS

Q1. Define file handling. What are the different types of files in Python?

Q2. Describe the different file open modes in open() function.

Q3. Explain the opening and closing of file using example.

Q4. Explain how the with statement is used using an example.

Q5. Explain the working of seek() and tell() method using examples.

****

UNIT 11: OBJECT-ORIENTED PROGRAMMING

UNIT STRUCTURE

11.1 Learning Objectives

11.2 Introduction

11.3 Object-Oriented Programming (OOP)

11.4 Classes and Objects

11.5 Inheritance

11.6 Polymorphism

11.7 Let Us Sum Up

11.8 Answers to Check Your Progress

11.9 Further Reading

11.10 Model Questions

11.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- explain the concept of OOP
- define classes and objects
- explain the concept of inheritance
- describe the different types of inheritances
- describe the concept and different types of polymorphism

11.2 INTRODUCTION

In this unit, you will be introduced to the concept of object oriented programming (OOP) in Python. In the last unit, we concept of file handling in Python was covered. The unit covers the features of OOP along with detailed explanation of classes and objects. The important concept of inheritance is introduced in this unit. The different types of inheritance like single, multiple, multilevel and hierarchical inheritances are also explained in this unit. The concept of polymorphism is also explained in this unit. The different types of polymorphism like duck typing, operator overloading and method overriding are also explained in this unit. In the next unit, you will explore

the concept of exception handling.

11.3 OBJECT ORIENTED PROGRAMMING

Object-Oriented Programming (OOP) is a popular programming paradigm that is based on the concept of objects. An object is a collection of data (attributes) and functions (methods) that operate on the data. These objects are instances of classes, which act as blueprints for creating similar objects.

Features of OOP:

OOP promotes concepts such as encapsulation, inheritance, polymorphism, and abstraction, making it easier to design modular, reusable, and maintainable code.

Abstraction: Abstraction hides complex implementation details and shows only the necessary features. This simplifies the complex systems by exposing only the relevant parts.

Encapsulation: Encapsulation means hiding of internal details while showing only the necessary parts of the object. This protects the data from being changed accidentally.

Inheritance: Inheritance allows one class to use the properties and methods of another class. This promotes code reuse.

Polymorphism: Polymorphism provides the flexibility to use the same method or operator to perform different operations.

11.4 CLASSES AND OBJECTS

A class is a blueprint or template for creating objects. It defines a set of attributes and methods that each object will have. An object is an instance of a class. It is a real-world entity that contains data and methods that operate on the data.

The syntax for defining a class is:

```
class Classname(object):
    """ docstring describing the class """
    attributes
    def__init__(self):
    def method1():
```

```
    def method2():
```

The syntax for creating an object is:

```
objectname = Classname()
```

To define a class, we us the keyword "class" followed by the classname. The word object is optional and is placed in the parentheses to indicate that the class inherits from Pythons base class. This is followed by the docstring which is used to describe the purpose of the class and is written between triple single or triple double quotes. The class contains attributes which store data and methods that are used to access the data. The __init__(self) method is a special initializer method which is used to assign values to the attributes.

Let us look at a program to understand how classes and objects function.

Example 1: Program to demonstrate classes and objects

```
class Book:
   def __init__(self):
      self.name = "Programming in Python"
      self.pages = 200
      self.year = 2025
   def show_data(self):
      print("Book Name: ",self.name)
      print("Pages: ",self.pages)
      print("Year of publicaation: ",self.year)
b=Book()
b.show_data()
```

In program 11.1 a Book class has been defined. The name of the class "Book" starts with a capital letter because by convention, class names in Python begin with a capital letter. Three variable declarations, name, pages and year have been made inside the init() method, which is a special method to initialize variables. The keyword "self" is passed as the first parameter to the __init__() method. "self" is a default variable that refers to the current instance of the class and holds its memory address. When an object of the class is created, the object's reference is automatically passed to self within class methods. This allows access to the instance's attributes and methods from within the class. The dot operator is used to access the instance vari-

ables, for example, self.name, self.pages and self.year. The show_data() method takes "self" as a parameter and uses it to access and display the values of the instance variables. An object "b" of Book class has been created. To display the values of the variables in object b, the method show_data() has been called using dot operator. The output of the program is shown below.

Output:

```
C:\Users\Shruti\Desktop\Python_Scripts>python book.py
Book Name:  Programming in Python
Pages:  200
Year of publicaation:  2025
```

Constructor:

The constructor is a special method used to initialize objects and is automatically called when a new object of the class is created. It is defined using the special method __init__(). The "self" variable is always the first parameter of the constructor and holds the memory address of the current instance.

Constructor can have one or more parameters. In Program 11.1, the constructor has only one parameter "self". However, let us look at an example where constructors have multiple parameters. In Program 11.2, we modify the constructor to accept multiple arguments.

Example 2: Program to demonstrate constructor

```
class Book:
    def __init__(self,n="Programming in Python",p=200,y=2025):
        self.name = n
        self.pages = p
        self.year = y
    def show_data(self):
        print("Book Name: ",self.name)
        print("Pages: ",self.pages)
        print("Year of publicaation: ",self.year)

b1=Book()
b1.show_data()
```

```
print()
# passing multiple arguments
b2=Book("Operating System",350,2020)
b2.show_data()
```

The program 11.2 demonstrates the use of constructor both with and without multiple arguments. The first object "b1" is created without passing any values. So, the default values will be assigned to the instance variables name, pages and year. The second object "b2" is created by passing three actual arguments: 'Operating System', 350 and 2020. These values are assigned to the parameters n, p, and y and then stored in the instance variables name, pages and year as can be seen in the output below.

Output:

```
C:\Users\Shruti\Desktop\Python_Scripts>python book_constructor.py
Book Name:  Programming in Python
Pages:  200
Year of publicaation:  2025

Book Name:  Operating System
Pages:  350
Year of publicaation:  2020
```



CHECK YOUR PROGRESS

Q1. Define class.

Q2. What is an object?

Q3. How is a constructor defined?

11.5 INHERITANCE

Inheritance is a fundamental concept of OOP that allows a class to inherit attributes and methods from another class. The class that inherits the methods and properties is called the child or subclass or derived class while the class it inherits from is called the parent or superclass or base class. Inher-

itance promotes code reuse and helps to create a hierarchical relationship between classes as the child class inherits all the methods and properties of the parent class. There are five types of inheritance: single inheritance, multiple inheritance, multilevel inheritance, hierarchical inheritance and hybrid inheritance.

We will look at each of these inheritances in detail in this section.

- **Single inheritance:**

In single inheritance, a child class inherits from only one parent class. This means the child class acquires all the attributes and methods of that single parent class. Let us look at an example to understand this.

Example 3: Program to demonstrate single inheritance

```
class BankAccount:
    def __init__(self,name):
        self.name=name
    def display_info(self):
        print("This account belongs to",self.name)

class SavingsAccount(BankAccount):
    def show_interest(self):
        print("Rate of Interest is 8.5%")

s1=SavingsAccount("Sruti")
s1.display_info()
s1.show_interest()
```

In Program 11.3, we have a parent class named BankAccount and a child class named SavingsAccount. The SavingsAccount class inherits from BankAccount class and reuses its constructor and method, while also defining its own method in the program. In this example, an object "s1" of class SavingsAccount is created. The value "Sruti" is passed as the name when the object is created. Because SavingsAccount inherits from BankAccount, it uses the __init__() method of BankAccount. The line

s1.display_info() calls the display_info() method inherited from the BankAccount class which prints the name of the account holder. The line s1.show_interest() calls the show_interest() method of the SavingsAccount class which prints the rate of interest for savings account.

Output:

```
C:\Users\Shruti\Desktop\Python_Scripts>python single_inh.py
This account belongs to Sruti
Rate of Interest is 8.5%
```

- **Multilevel Inheritance:**

Multilevel inheritance is a type of inheritance in which a class inherits from a child class, which itself is derived from another parent class. This forms a chain of inheritance across multiple levels, hence the name "multilevel" inheritance. In this type of inheritance, a child class inherits from a parent, and another child class inherits from that child. This allows the lowest-level class to access all attributes and methods from its entire inheritance chain i.e. the last class in the chain can use all the features of its ancestors. Python uses method resolution order (MRO) to determine which method to execute when there are multiple levels. Let us look at an example to understand multilevel inheritance.

Program 4: Program to demonstrate multilevel inheritance

```
class BankAccount:
    def __init__(self,name):
        self.name=name
    def display_info(self):
        print("This account belongs to",self.name)


class SavingsAccount(BankAccount):
    def __init__(self,name,rate):
        super().__init__(name)
        self.rate=rate
    def show_interest(self):
        print("Rate of Interest is ",self.rate)
```

```
class ChildSavingsAccount(SavingsAccount):
    def __init__(self,name,rate,gname):
        super().__init__(name,rate)
        self.gname=gname
    def show_guardian_name(self):
        print("Guardian Name is: ",self.gname)


c1=ChildSavingsAccount("Sia",9,"R. Deka")
c1.display_info()
c1.show_guardian_name()
```

This program demonstrates multilevel inheritance in Python using three classes: BankAccount, SavingsAccount, ChildSavingsAccount. The base class BankAccount has a constructor __init__() that initializes the name of the account holder and a method display_info() to display it. The SavingsAccount class inherits from BankAccount. In the SavingsAccount class constructor, super().__init__(name) is used to call the parent class constructor to initialize the name. The SavingsAccount class also adds a new attribute rate and defines a method show_interest() to display the rate of interest. The ChildSavingsAccount class inherits from the SavingsAccount class, which means it also indirectly inherits from BankAccount, forming a multilevel inheritance chain. The ChildSavingsAccount constructor calls super().__init__(name, rate) to initialize attributes from both the BankAccount and SavingsAccount classes. It adds a new attribute gname and defines a method show_guardian_name() to display the name of the guardian of the child. An object "c1" of the ChildSavingsAccount class is created with the values "Sia", 9 and "R. Deka" as name, rate and gname respectively.

Output:

```
C:\Users\Shruti\Desktop\Python_Scripts>python multilevel_inh.py
This account belongs to Sia
Guardian Name is:  R. Deka
```

- **Multiple inheritance:**

In multiple inheritance, a child class inherits from two or more parent classes. This allows the child class to access attributes and methods from all its

parent classes. The child class can combine features of all parent classes and also add more of its own features. Python uses Method Resolution Order (MRO) to decide the order in which methods are inherited. Let us look at an example to understand multiple inheritance.

Example 5: Program to demonstrate multiple inheritance

```
class BankAccount:
    def __init__(self,name):
        self.name=name
    def display_info(self):
        print("This account belongs to",self.name)

class LoanServices:
    def __init__(self, loan_type):
        self.loan_type = loan_type
    def display_loan(self):
        print("Loan Type: ",self.loan_type)

class BankManager(BankAccount, LoanServices):
    def __init__(self, name, loan_type, branch_name):
        BankAccount.__init__(self, name)
        LoanServices.__init__(self, loan_type)
        self.branch_name = branch_name
    def display_manager_info(self):
        print("Branch: ",self.branch_name)

manager = BankManager("A. Hazarika","Home Loan","Beltola Branch")

manager.display_info()
manager.display_loan()
manager.display_manager_info()
```

The program 11.5 demonstrates multiple inheritance in Python. It defines two parent classes: BankAccount and LoanServices. The BankAccount class has a constructor that sets the name of the account holder and a method

display_info() that displays it. The LoanServices class similarly has a constructor that sets the loan type and a method display_loan() that prints it. The BankManager class inherits from both BankAccount and LoanServices. In the BankManager constructor, it calls the constructors of both parent classes using to initialize their attributes. It also defines an additional attribute branch_name specific to the BankManager class. The display_manager_info() method in BankManager displays the name of the bramch where loan was issued. An object "manager" of class BankManager is created with customer name "A. Hazarika" , loan type "Home Loan", and branch name "Beltola Branch". The manager.display_info () and manager.display_loan () methods display the inherited information from the parent classes. Finally, manager.display_manager_info() shows the name of the branch that issued loan.

Output:

```
C:\Users\Shruti\Desktop\Python_Scripts>python multiple_inh.py
This account belongs to A. Hazarika
Loan Type:  Home Loan
Branch:  Beltola Branch
```

- **Hierarchical inheritance:**

When multiple child classes inherit from the same parent class, then the inheritance is said to be hierarchical inheritance. In this type of inheritance, there is only one parent class but multiple child classes that inherit from the same parent. All the child classes share the features of the common parent, but each child can also have its own additional features. Let us look at an example to understand hierarchical inheritance.

In this program the base class BankAccount has a constructor __init__() that initializes the name attribute. It also defines a method display_info() that prints the name of the account holder. The SavingsAccount class inherits from BankAccount, so it can access all attributes and methods of BankAccount. The BankAccount class also defines an additional attribute "interest" to store the rate of interest. The method show_interest() prints the rate of interest. Similarly, the CurrentAccount class also inherits from Vehicle. CurrentAccount class also defines its own attribute limit and a

method show_limit() to display the overdraft limit capacity for that account.

Example 6: Program to demonstrate hierarchical inheritance

```
class BankAccount:
    def __init__(self,name):
        self.name=name
    def display_info(self):
        print("This account belongs to",self.name)

class SavingsAccount(BankAccount):
    def __init__(self,name,rate):
        super().__init__(name)
        self.rate=rate
    def show_interest(self):
        print("Rate of Interest is ",self.rate)

class CurrentAccount(BankAccount):
    def __init__(self,name,limit):
        super().__init__(name)
        self.limit=limit
    def show_limit(self):
        print("Overdraft limit is ",self.limit)

s=SavingsAccount("P. Das",9)
s.display_info()
s.show_interest()
print()

c=CurrentAccount("S. Gogoi",50000)
c.display_info()
c.show_limit()
```

An object "s" of the SavingsAccount class is created with name "P. Dasl" and rat eof interest 9. An object "c" of the CurrentAccount class is created with name "S. Gogoi" and limit 50,000. The method s.display_info() and

s.show_interest() are called to display the name and rate of interest for savings account. Similarly, the methods c.display_info() and c.show_limit() are called to show the name and overdraft limit in current account.

Output:

```
C:\Users\Shruti\Desktop\Python_Scripts>python hier_inh.py
This account belongs to P. Das
Rate of Interest is  9

This account belongs to S. Gogoi
Overdraft limit is  50000
```

Hybrid inheritance: Hybrid inheritance is a combination of two or more types of inheritance in a single program. In hybrid inheritance, the combination may be in any form from any of the inheritances described so far: single, multilevel, multiple and hierarchical.

11.6 POLYMORPHISM

Polymorphism is a fundamental concept in object-oriented programming (OOP) that means "many forms". Polymorphism enables the same function name or operator to behave differently based on the context or the objects it is operating on. There are different types of polymorphism in Python: Duck Typing, Operator Overloading and Method Overriding.

Let us look at each of these forms of polymorphism.

- **Duck Typing:**

In this type of polymorphism, Python doesn't check the type of object but checks for the presence of a particular method or behaviour. This is based on the saying: "If it looks like a duck and quacks like a duck, it is a duck".

Let us look at the program to understand duck typing.

Example 7: Program to demonstrate duck typing polymorphism

```
class Duck:
    def sound(self):
        print("Quack Quack")
class Human:
    def sound(self):
        print("Hi Hello")
```

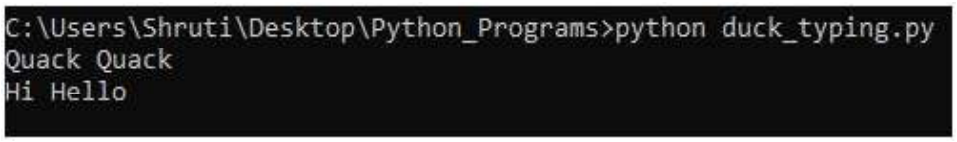
```
def call_sound(obj):
    obj.sound()

d=Duck()
h=Human()

#depending on type of object
call_sound(d)
call_sound(h)
```

The program 11.7 demonstrates polymorphism, where different classes Duck and Human have the same method name sound(). The call_sound() function calls the sound() method on any object passed to it. It is not required to mention the type of the object or to check whether the sound() method belongs to that object or not. Depending on whether a Duck or Human object is passed, it prints either "Quack Quack" or "Hi Hello" respectively.

Output:



- **Operator Overloading:**

In operator overloading, a single operator is given the ability to perform different tasks depending on the type of operands. This means that although the operator may look the same (such as +, -, or *), its behavior can change based on the context in which it is used. Instead of carrying out only its default or predefined operation, the operator is redefined to handle user-defined operations as well, making it more versatile and flexible in object-oriented programming. Let us look at a program to understand this.

Example 8: Program to demonstrate operator overloading

```
class Slm:
    def __init__(self,pages):
        self.pages = pages
    #adding two objects
```

```
    def __add__(self,other):
        return self.pages + other.pages
s1=Slm(35)
s2=Slm(50)
print("Total pages: ",s1+s2)
```

The program 11.8 demonstrates operator overloading using the "+" operator. A class Slm is defined with an initializer __init__ that sets the number of pages. The special method __add__ is overridden to allow the "+" operator to add "pages" attribute of two Slm objects. Two instances s1 and s2 are created with 35 and 50 pages respectively. When s1 + s2 is executed, Python internally calls s1.__add__(s2), which returns the sum of pages. Finally, the total number of pages (85) is printed.

Output:

```
C:\Users\Shruti\Desktop\Python_Programs>python oper_overloading.py
Total pages:  85
```

- **Method Overriding:**

Method overriding occurs when a sub class provides a specific implementation of a method that is already defined in its superclass. This is generally done to add some new functionality to the same method in the sub class. Run time polymorphism is achieved using method overriding in Python. Let us look at an example to understand method overriding.

Example 9: Program to demonstrate method overriding

```
class Employee:
    def salary(self):
        print("Employee salary is 30,000")

class Manager(Employee):
    def salary(self):
        print("Manager salary is 50,000")

m = Manager()
m.salary()
```

The program 11.9 demonstrates method overriding in Python. The Employee class has a method salary() that prints a base salary. The Manager class inherits from Employee but overrides the salary() method to provide a different output. When an object "m" of Manager is created and m.salary() is called, the overridden method in Manager is executed, printing "Manager salary is 50,000". This shows how a subclass can redefine a method from its parent class.

Output:

```
C:\Users\Shruti\Desktop\Python_Programs>python method_overriding.py
Manager salary is 50,000
```



CHECK YOUR PROGRESS

Q4. What is inheritance?

Q5. What is duck typing polymorphism?



11.7 LET US SUM UP

- Object-Oriented Programming (OOP) is a popular programming paradigm that is based on the concept of objects.
- Encapsulation means hiding of internal details while showing only the necessary parts of the object.
- A class is a blueprint or template for creating objects.
- An object is an instance of a class.
- The constructor is a special method used to initialize objects and is automatically called when a new object of the class is created. It is defined using the special method __init__().
- Inheritance is a fundamental concept of OOP that allows a class to inherit attributes and methods from another class.
- There are five types of inheritance: single inheritance, multiple inheritance, multilevel inheritance, hierarchical inheritance and hybrid inheritance.

- In single inheritance, a child class inherits from only one parent class.
- Multilevel inheritance is a type of inheritance in which a class inherits from a child class, which itself is derived from another parent class.
- In multiple inheritance, a child class inherits from two or more parent classes.
- When multiple child classes inherit from the same parent class, then the inheritance is said to be hierarchical inheritance.
- Polymorphism enables the same function name or operator to behave differently based on the context or the objects it is operating on.
- There are different types of polymorphism in Python: Duck Typing, Operator Overloading and Method Overriding.
- In operator overloading, a single operator is given the ability to perform different tasks depending on the type of operands.
- Method overriding occurs when a sub class provides a specific implementation of a method that is already defined in its superclass.



11.8 ANSWERS TO CHECK YOUR PROGRESS

Ans to Q1: A class is a blueprint or template for creating objects. It defines a set of attributes and methods that each object will have.

Ans to Q2: An object is an instance of a class. It is a real-world entity that contains data and methods that operate on the data.

Ans to Q3: Constructor is defined using the special method __init__().

Ans to Q4: Inheritance is a fundamental concept of OOP that allows a class to inherit attributes and methods from another class. The class that inherits the methods and properties is called the child or subclass or derived class while the class it inherits from is called the parent or superclass or base class.

Ans to Q5: In duck typing polymorphism, Python doesn't check the type of object but checks for the presence of a particular method or behaviour. This is based on the saying: "If it looks like a duck and quacks like a duck, it is a duck".



11.9 FURTHER READING

1) Kanetkar, Y., & Kanetkar, A. (2020). Let Us Python: Python Is Future, Embrace It Fast (Vol. 2). BPB Publications.



11.10 MODEL QUESTIONS

Q1. Define classes and objects. What is the purpose of a constructor?

Q2. Describe the features of OOP.

Q3. Explain the concept of classes and objects using example.

Q4. Define inheritance. What are the different types of inheritance?

Q5. Explain single inheritance using an example.

Q6. Explain multiple inheritance using an example.

Q7. Explain multilevel inheritance using an example.

Q8. Explain hierarchical inheritance using an example.

Q9. What is polymorphism? What are the different types of polymorphism?

Q10. Explain duck typing polymorphism using an example.

Q11. Explain operator overloading using example.

Q12. Describe how method overriding works using example.

Q13. Write a program to explain the concept of hybrid inheritance.

****

UNIT 12: EXCEPTION HANDLING

UNIT STRUCTURE

12.1 Learning Objectives
12.2 Introduction
12.3 Exception Handling
12.4 Types of Exceptions in Python
12.5 Handling Exceptions in Python
12.6 User Defined Exceptions
12.7 Let Us Sum Up
12.8 Answers to Check Your Progress
12.9 Further Reading
12.10 Model Questions

12.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- explain the concept of exception handling
- describe the different types of exceptions
- explain the purpose of use try, except, raise and finally in exception handling
- use try, except and finally blocks in programs for exception handling
- create user-defined exceptions in programs

12.2 INTRODUCTION

In this unit, you will be introduced to the concept of exception handling in Python. In the last unit, we had covered the concept of object oriented programming. Topics like classes and objects, inheritance and polymorphism were covered in detail in the last unit.

This unit covers the types of exceptions in Python along with the class hierarchy that is associated with it. We will learn the purpose of and how to use try, except, raise and finally blocks in exception handling in Python. The unit covers a lot of examples that help to understand the concept of

exception handling. In addition to this, we will also learn how to create our own exceptions using user-defined exceptions. In the next unit, you will explore the concept of regular expressions.

12.3 EXCEPTION HANDLING

An exception is an error that occurs during the execution of a program, interrupting the expected flow of instructions. Unlike syntax errors, which are detected during the compilation or parsing stage, exceptions are runtime errors, meaning they occur while the program is running. These errors can be caused by various situations, such as dividing a number by zero or trying to open a file that doesn't exist.

Exceptions can, however be handled by the programmer by using certain mechanisms provided by the programming language, like try, except, finally, and raise in Python. Exceptions handling enables programmers to have even more precise error management. By handling exceptions properly, programmers can prevent the program from crashing and instead provide informative error messages so that alternative actions can be taken to ensure smooth execution.

Let us now look at the different types of exceptions.

12.4 TYPES OF EXCEPTIONS IN PYTHON

All exceptions are represented as classes in Python. In Python, the exceptions that are already defined and provided by the language itself are known as "built-in" exceptions. These exceptions are part of the Python standard library and are automatically available for use without the need for any additional imports or definitions. The base class for all built-in exceptions is "BaseException" class. This is because Python organizes the built-in exceptions in a structured hierarchy under the BaseException class. This class has several sub classes in the hierarchy like "SystemExit", "KeyboardInterrupt", but "Exception" sub class is one of the most important sub class. Various standard errors like ArithmeticError, EOFError and ImportError are defined under the "Exception" class. An error has to be compulsorily be handled, otherwise the program would not be executed.

Another sub class is "Warning". All the warnings are derived as sub class from the "Warning" class. A warning represents a caution and even if it is not handled, the program would execute. So, while writing programs we may neglect the warnings but not the errors.

Let us look at some of the important built-in exceptions that are available in Python. The exception class name and their description is listed in table 10.1.

Table 10.1: Few built-in exceptions

Exception Class Name	**Description**
ArithmeticError	Represents the base class for arithmetic errors like ZeroDivisionError, FloatingPoint Error
ZeroDivisionError	This exception is raised when the denominator is zero in a division or modulus operation.
FloatingPointError	This exception is raised when a floating point operation fails.
AssertionError	This is raised when an assert statement gives error.
SyntaxError	This exception is raised when the compiler finds a syntax error.
TypeError	This exception is raised when an operation is called for an inappropriate datatype.
EOFError	This exception is raised when input() function reaches the end of file condition (EOF) without reading any data.
RuntimeError	This exception is raised when an error occurs but does not belong to any specific category.
ImportError	This exception is raised when an import statement fails to find the module.
NameError	This exception is raised when an identifier is not found.
IndexError	This exception is raised when a sequence index or subscript is out of range.
IndentationError	This exception is raised when indentation is not specified properly.

KeyboardInterrupt	This exception is raised when the user hits the interrupt key (normally Control-C or Delete)
SystemExit	This exception is raised by the sys.exit() function. The Python interpreter exits when this exception is not handled.

In addition to the built-in exceptions, Python also has the feature of user-defined exceptions. If a specific exception we need is not provided by Python's built-in exceptions, we can define our own custom exceptions to handle such situations in a more meaningful and controlled way. The exceptions which are written by the programmer and are not available in Python library are called user-defined exceptions. For creating the user-defined exceptions, the class should be derived from "Exception" class and not "BaseException" class.



CHECK YOUR PROGRESS

Q1. Define exception.

Q2. What are the different types of exceptions?

12.5 HANDLING EXCEPTIONS IN PYTHON

Exception handling is done by programmers to make their code more robust and to prevent unexpected crashes due to runtime errors. In Python, exception handling is achieved by using the try, except, raise and finally blocks. These blocks help to manage and respond to errors that may occur during the execution of a program, allowing the program to continue running smoothly instead of crashing. The try block contains the code that might raise an exception. Python monitors this block for any errors. If an error occurs, the control moves to the except block, where the error can be handled gracefully by displaying a message. The finally block is optional and contains code that is always executed regardless of whether an exception occurred or not. It is typically used for tasks like closing files. We will learn about raise statements in the next section.

The syntax for exception handling is given below.

Syntax:

```
try:
    statements
except ExceptionClass1:
    handler1
except ExceptionClass2:
    handler2
……
else:
    statements
finally:
    statements
```

The statements where there is possibility of exceptions are written inside the try block. The advantage of using try block is that when some exceptions occurs, the program will not be terminated, but rather the Python Virtual Machine (PVM) understands that there is an exception and the control jumps to the "except" block. The except block contains display messages regarding the type of exception encountered. The block can also be used to instruct the user on what to avoid so as to not repeat the error. The statements written inside the except block are called "handlers" since they handle the situation when the exception occurs. The try block can be followed by several except blocks since we can use multiple except blocks to handle multiple exceptions in the same program. But we cannot have an except block without a try block. We can also use a single except block and write many exceptions as a tuple inside parenthesis. This can be done by using the syntax below:

```
except(ExceptionClass1, ExceptionClass2,……. )
```

The else and the finally block are not compulsory. The else block is only executed when there is no exception. The finally block is always executed and so it must include statements that need to always run whether an exception occurs or not. This block is useful for clean-up actions like closing the files and terminating any other process which are running. These tasks together constitute exception handling. Let us now look at few examples of

exception handling.

Example 1: Program to demonstrate ZeroDivisionError exception

```
# Program to demonstrate ZeroDivisionError exception
try:
    num = int(input("Enter a number to divide 100: "))
    result = 100 / num
    print("Result:", result)
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
finally:
    print("The program ends here.")
```

In program 12.1, in the first case, the user has given the input as 500, so the division is calculated and the result is printed. The message inside the finally block is also displayed since finally block is always executed. In the second case, the user enters 0, so Python raises a ZeroDivisionError because dividing by zero is mathematically undefined. The except block catches this error and prints an error message instead of crashing the program. The finally block will execute as well because it is guaranteed to run, regardless of whether an exception occurs or not. This ensures that the statement inside it is always displayed.

Output:

```
C:\Users\Shruti\Desktop\Python_Programs>python zero_division.py
Enter a number to divide 100: 500
Result: 0.2
The program ends here.

C:\Users\Shruti\Desktop\Python_Programs>python zero_division.py
Enter a number to divide 100: 0
Error: Cannot divide by zero.
The program ends here.
```

Example 2: Program to demonstrate FileNotFound Error

```
# Program to demonstrate FileNotFound Error
try:
    file = open("file.txt", "r")
```

```
    data = file.read()
    print(data)
except FileNotFoundError:
    print("Error: The file does not exist.")
finally:
    print("File operation attempted.")
```

The program 12.2 tries to open a file named "file.txt" in read mode ("r"). If the file exists, the program reads its contents using file.read() function and then prints the content of the file. However, if the file "file.txt" is not found, Python then raises a FileNotFoundError. This error is caught by the except block where the program then prints an error message. This prevents the program from crashing and allows graceful handling of the missing file. The code in the finally block is then executed to print a message that file operation was attempted.

Output:

```
C:\Users\Shruti\Desktop\Python_Programs>python file_not_found.py
Error: The file does not exist.
File operation attempted.
```

Example 3: Program to demonstrate Type Error exception

```
# Program to demonstrate Type Error exception
try:
    a = 5
    b = 10
    text1 = "Hello Learner"
    text2 = "Welcome to KKHSOU"
    res1 = a + b
    print("Result 1:", res1)
    res2 = text1 + text2
    print("Result 2:", res2)
    res3 = a + text1
    print("Result 3:", res3)
except TypeError:
    print("Error: Cannot add a number and a string together.")
```

```
finally:
    print("Type checking operation completed.")
```

In this program, two integers "a" and "b" along with two strings "text1" and "text2" are defined. The program first adds the two integers and prints the result, then concatenates the two strings and prints that result as well since both these operations are valid. However, when it tries to add an integer "a" to a string "text1", it causes a TypeError because Python does not allow adding a number and a string directly. This error is caught by the except block, which prints an appropriate error message. The finally block is then executed to print a message indicating the operation is complete.

Output:

```
C:\Users\Shruti\Desktop\Python_Programs>python type_error.py
Result 1: 15
Result 2: Hello LearnerWelcome to KKHSOU
Error: Cannot add a number and a string together.
Type checking operation completed.
```

Example 4: Program to demonstrate NameError and IndexError together in one except block

```
# Program to demonstrate NameError and IndexError together
try:
    my_list = [10, 20, 30]
    print(my_list[5])
    print(value)
except (NameError, IndexError):
    print("Error: Either an undefined name was used or a list index is out of range.")
finally:
    print("Exception handling completed.")
```

In program 12.4, we have defined a list named "my_list" with three values. However, in the program we try to print the value of "my_list[5]" which is an index that doesn't exist in the list (only indices 0-2 exist), so it raises an IndexError. Additionally, the variable "value" is not defined, and so this would

also raise a NameError, but it's not reached in this run because the previous line already raises an error. The except block handles both exceptions using a tuple (NameError, IndexError). The finally block always executes, indicating the end of the program.

Output:

```
C:\Users\Shruti\Desktop\Python_Programs>python Name_Index_error.py
Error: Either an undefined name was used or a list index is out of range.
Exception handling completed.
```

12.6 USER DEFINED EXCEPTIONS

User-defined exceptions, also known as custom exceptions, are exceptions that are specifically created by the programmer to handle unique or specialized situations that are not covered by Python's built-in exceptions. These are particularly useful when we want to raise errors that are meaningful and relevant to specific programs or applications. We can make our programs more robust, readable, and easier to debug, by defining our own exception classes especially when dealing with complex or domain-specific error conditions.

Since all exceptions are classes, so the user-defined exception should also be a class. This class should be a sub class to the in-built "Exception" class. The syntax for building own exceptions is given below.

Syntax:

```
class MyException(Exception):
    def __init__(self,arg):
        self.msg = arg
```

In the syntax, the name of our exception class is "MyException" which is a sub class for "Exception" class. This class has a constructor where a variable "msg" is defined. When the programmer suspects the possibility of exception in the program, the "raise" statement is used to raise the user-defined exception. The syntax for using raise statement is given below.

```
raise MyException("message")
```

Here, the raise statement is raising MyException class object that contains the given "message". This code can be written inside a try block and the exception can be caught in the except block. The syntax for this is given

below.

```
try:
        code
except MyException as me:
        print(me)
```

Here, the object "me" contains the message that was given in the raise statement. Let us look at an example to understand how to create and use user-defined exceptions.

Program 12.5: Program to demonstrate user-defined exceptions

```
# Program to demonstarte user-defined exception class
class InvalidAgeError(Exception):
    def __init__(self, message="Age must be above 18. "):
        self.message = message
        super().__init__(self.message)

def adminssion(age):
    if age < 18 or age > 60:
        raise InvalidAgeError(f"Invalid age: {age}. Age must be above 18.")
    else:
        print("Admission successful!")

try:
    user_age = int(input("Enter your age: "))
    adminssion(user_age)
except InvalidAgeError as e:
    print("User-defined Exception Caught:", e)
finally:
    print("Program successfully ends.")
```

In program 12.5, "InvalidAgeError" is a user-defined exception class that inherits from Python's built-in Exception class. In the admission() function, we raise InvalidAgeError if the age is below the allowed age of 18 years. The try block attempts to take the user's input and call admission(). If the age is invalid, the custom exception is caught and handled in the except

InvalidAgeError block. The finally block always runs to indicate the end of the process. From the output, we can see that when the user enters the age as "45", this age is above 18 so it falls within the valid range, so no exception is raised, and the admission is successful. However, in the second case, the user inputs the age as "15", which is less than the threshold value of 18. As a result, the custom exception is triggered and handled appropriately, displaying the error message.

Output:

```
C:\Users\Shruti\Desktop\Python_Programs>python user_defined_exception.py
Enter your age: 45
Admission successful!
Program successfully ends.

C:\Users\Shruti\Desktop\Python_Programs>python user_defined_exception.py
Enter your age: 15
User-defined Exception Caught: Invalid age: 15. Age must be above 18.
Program successfully ends.
```



12.7 LET US SUM UP

- An exception is an error that occurs during the execution of a program, interrupting the expected flow of instructions.
- All exceptions are represented as classes in Python.
- In Python, the exceptions that are already defined and provided by the language itself are known as "built-in" exceptions.
- The base class for all built-in exceptions is "BaseException" class.
- "Exception" sub class is one of the most important sub class in exception handling.
- In Python, exception handling is achieved by using the try, except, raise and finally blocks.
- The try block contains the code that might raise an exception. Python monitors this block for any errors.
- If an error occurs, the control moves to the except block, where the error can be handled gracefully by displaying a message.

- The finally block is optional and contains code that is always executed regardless of whether an exception occurred or not.
- User-defined exceptions, also known as custom exceptions, are exceptions that are specifically created by the programmer to handle unique or specialized situations that are not covered by Python's built-in exceptions.
- User-defined exception should be a sub class to the in-built "Exception" class.
- The "raise" statement is used to raise the user-defined exception.



12.8 ANSWERS TO CHECK YOUR PROGRESS

Ans to Q1: An exception is an error that occurs during the execution of a program, interrupting the expected flow of instructions.

Ans to Q2: There are two types of exceptions: built-in exceptions and user-defined exceptions.

Ans to Q3: The finally block is optional and contains code that is always executed regardless of whether an exception occurred or not.

Ans to Q4: User-defined exceptions are exceptions that are specifically created by the programmer to handle unique or specialized situations that are not covered by Python's built-in exceptions.



12.9 FURTHER READING

1) Kanetkar, Y., & Kanetkar, A. (2020). Let Us Python: Python Is Future, Embrace It Fast (Vol. 2). BPB Publications.



12.10 MODEL QUESTIONS

Q1. What is exception handling in Python?

Q2. Describe the different types of exceptions in Python.

Q3. Define a warning. What is the importance of "Exception" class in exception handling?

Q4. Define user-defined exceptions. How are they defined in Python?

Q5. Explain the ZeroDivisionError exception using an example.

Q6. Explain how try, except and finally are used in exception handling using an example.

Q7. Explain any five exceptions along with their purpose.

Q8. Create a user-defined exception class that raises an exception if a student scores less than the minimum required marks (28) in a theory exam. Write a program to input the marks and handle the exception appropriately.

Q9. Create a user-defined exception class that raises an exception if a student's attendance is less than 75%. Write a program to input the number of total and attended classes and display an appropriate message using exception handling.

Q10. Explain the ImportError exception using an example.

****

UNIT 13: REGULAR EXPRESSIONS

UNIT STRUCTURE

13.1 Introduction

13.2 Learning Objectives

13.3 Regular Expressions in Python

    13.3.1 Defining Regular Expressions in Python

13.4 Working with Regular Expressions in Python

13.5 Common Elements used in Constructing Regex Patterns

13.6 Practical Applications

13.7 Let Us Sum Up

13.8 Answers to Check Your Progress

13.9 Further Reading

13.10 Model Questions

13.1 INTRODUCTION

In the previous unit, we explored exception handling in Python, learning how to detect, manage, and respond to different types of runtime errors. In this unit, we will discuss regular expressions in Python, an essential tool for handling and manipulating textual data.

We will explore the use of regular expressions in Python through the built-in re module. The focus will be on understanding how to define and apply regular expression patterns to perform various string operations. These include searching, matching, extracting, replacing, and validating textual data. Regular expressions are powerful tools widely used in real-world programming scenarios such as data processing, form validation, log analysis, and web development.

13.2 LEARNING OBJECTIVES

After going through this units, you will be able to:

- learn the syntax and construction of regular expressions in Python.
- learn how to import and use Python's re module.
- apply common re functions such as match(), search(), findall(), sub(),

and split().

- use regular expressions for real-world tasks like validation, text searching, replacing, and data extraction.
- write and interpret various regex patterns using literals, metacharacters, quantifiers, character classes, and anchors.

13.3 REGULAR EXPRESSIONS IN PYTHON

A regular expression is a string that contains special symbols and characters to find and extract the information required by us from a given data. Regular expressions are often used for validating input. Whether we need to search for specific patterns, validate input, or replace parts of a string, regular expressions can be extremely helpful.

For instance, when we fill out an online form, fields such as name, age, email address, and mobile number are checked against predefined acceptable patterns. If the entered values match these patterns, the input is considered valid. For example, a mobile number is not valid if it is not 10 digits long or if it has any letters or special characters.

Regular expressions are available not only in Python but also in many languages like Java, Perl, AWK, etc. Python's regular expressions are essential tools for working with patterns in text data. Regular expressions, often abbreviated as regex, are a powerful tool for working with strings.

13.3.1 Defining Regular Expressions in Python

In Python, regular expressions are defined using the re module. This module allows us to search, match, and manipulate strings using regular expressions in an efficient and concise manner. To use the re module in Python, we first need to import it:

import re

The re module contains many useful functions and methods that help us create and use regular expressions. It offers functions like search(), findall(), and sub() to locate, extract, or replace text that matches the defined patterns, making it easy to manipulate strings based on the desired criteria.

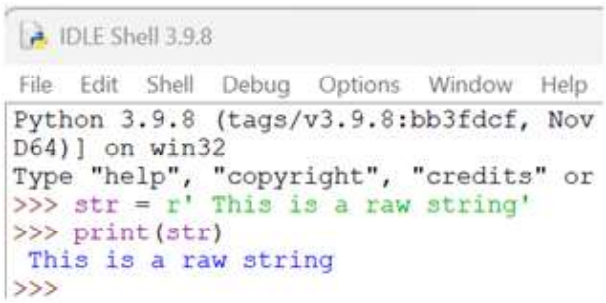
Generally, we write regular expressions as raw strings. This is done by

prefixing 'r' before the string.

str = r' This is a raw string'

print(str)

Here is the code in Python IDLE



Python IDLE (Integrated Development and Learning Environment) is the default editor and interactive shell that comes bundled with Python. It provides an easy way to write, edit, and execute Python code, especially useful for beginners.

13.4 WORKING WITH REGULAR EXPRESSIONS IN PYTHON

Most modern programming languages support regular expressions, typically through libraries or built-in features.

1. Matching strings using re.match()

The re.match() function checks for a match only at the beginning of the string.

Syntax: re.match(pattern, string)

Returns: A match object if the pattern matches at the start of the string, otherwise None.

We can also execute a Python program, from command prompt. For this, we can follow these steps:

Create the script:

- Open a text editor like Notepad (or any code editor)
- Write your Python code
- Save the file with a .py extension (e.g., reg1.py)

Run the script:

- Open Command Prompt (Windows) or Terminal (macOS/Linux)
- Navigate to the directory containing your file
- Type python filename.py (e.g.,python reg1.py)
- Press Enter to execute the program

Example 1: filename reg1.py

```
import re
pattern = r'hello'
test = 'hello world'
result = re.match(pattern, text)
if result:
    print("Match found:", result.group())
else:
    print("No match")
```

Output:

Match found : hello

```
C:\pythonscript>python reg1.py
Match found: hello
```

Example 2: match.py

```
import re            # imports the re module
pattern = r"\d+"     # Regex to match one or more digits
text = "781022kkhsou456"

# Using match() to check if the pattern matches
match_result = re.match(pattern, text)
if match_result:
        print(f"Match found: {match_result.group()}")
else:
        print("No match at the start of the string")
```

Output:

Match found: 781022

```
C:\pythonscript>python match.py
Match found: 781022
```

A detailed explanation of the code (Example 2) is as follows:

import re # This imports the re module

pattern = r"\d+" # Regex to match one or more digits

- pattern is a string that defines a regular expression (regex).
- The r before the string denotes a raw string in Python, which treats backslashes (\) as literal characters rather than escape characters.
- \d+ is the actual regular expression where \d represents a digit (0-9). + is a quantifier that means "one or more of the preceding element." In this case, one or more digits. So, \d+ matches sequences of digits like 781022.

text = "781022kkhsou456"

This is the input string in which we are trying to search for patterns. The string contains a combination of digits and letters: "781022kkhsou456".

match_result = re.match(pattern, text)

The re.match() function tries to find a match only at the beginning of the string. In this case, "781022" is a sequence of digits, so it matches.

- pattern: The regular expression (\d+ in this case) to be searched. \d+ searches for one or more digits in the string.
- text: The string ("781022kkhsou456") in which the match will be attempted.

If the pattern matches the beginning of the string, the result is a match object; if it doesn't match, the result will be None.

```
if match_result:
    print(f"Match found: {match_result.group()}")
else:
    print("No match at the start of the string")
```

if match_result:

If a match is found, the block inside the if statement is executed.

match_result.group() returns the actual matched substring. In this case, it will be "781022" because the pattern \d+ matches the digits at the beginning of the string. If no match is found, the else block is executed, printing "No match at the start of the string."

Output:
Match found: 781022

This occurs because the string "781022kkhsou456" starts with digits, and the \d+ pattern successfully matches "781022." If the string started with non-digits, no match would be found.

2. Searching patterns with re.search()

The re.search() function looks for the first match anywhere in the string.
Syntax: re.search(pattern, string)
Returns: A match object for the first occurrence of the pattern, or None if no match is found.

Example 3:

```
pattern = r'world'
string = 'hello world'
result = re.search(pattern, string)
if result:
    print("Search result found:", result.group())
else:
    print("No match found")
```

Output:
Search result found: world

Example 4: ex1.py

```
import re
```

```
pattern = r'\d+'
match = re.search(pattern, 'There are 120 students')
if match:
    print("Found number:", match.group())
```

Output:

```
C:\pythonscript>python ex1.py
Found number: 120
```

Example 5: Python program to create a regular expression to search for strings starting with m and having 5 characters using the search() method.

```
import re
str= 'school mango monkey donkey money book pencil'
result=re.search(r'm\w\w\w\w', str)
if result:
    print(result.group())
```

```
C:\pythonscript>python prog2.py
mango
```

Only 'mango' is displayed as the search() method searches for the strings according to the regular expression and returns only the first string.

3. Finding all matches using re.findall()

The re.findall() method returns all non-overlapping matches of the pattern in the string as a list.

Syntax: re.findall(pattern, string)

Returns: A list of all matches found.

Example 6: Filename reg4.py

```
import re
pattern = r'\d+' # Matches one or more digits
string = 'I have 10 pencils, 5 pens, 3 erasers'

result = re.findall(pattern, string)
print("Numbers found:", result)
```

Output:

Numbers found: ['10', '5', '2']

```
C:\pythonscript>python reg4.py
Numbers found: ['10', '5', '3']
```

4. Replacing substrings using re.sub()

The re.sub() method allows us to replace or substitute occurrences of the pattern with a specified replacement string.

Syntax: re.sub(pattern, replacement, string)

Returns: A new string with all the matches replaced.

Example 7: filename reg2.py

```
Import re
pattern = r'apple'
replacement = 'mango'
string = 'I like apple shake.'

result = re.sub(pattern, replacement, string)
print("Modified string:", result)
```

If we execute it with python reg2.py, it will give the output with a modified string replacing the string 'apple' with 'mango'.

Output:

Modified string: I like mango shake.

```
C:\pythonscript>python reg2.py
Modified string: I like mango shake.
```

5. Splitting string using re.split()

The re.split() function splits a string by the occurrences of a pattern, similar to Python's built-in split() function but more flexible due to pattern matching.

Syntax: re.split(pattern, string)

Returns: A list of substrings.

Example 8:

```
Import re
pattern = r'\s+'    # Split by one or more spaces
string = 'Split  this   sentence  properly.'
result = re.split(pattern, string)
print(result)
```

Output:

['Split', 'this', 'sentence', 'properly.']

```
C:\pythonscript>python reg3.py
['Split', 'this', 'sentence', 'properly.']
```



CHECK YOUR PROGRESS

13.5 COMMON ELEMENTS USED IN CONSTRUCTING REGEX PATTERNS

Let us first explore some of the most common elements used in constructing regular expression patterns.

1) Literal Characters:

In Python's regular expressions, literal characters are the characters that match themselves directly in a string. These characters are interpreted literally without any special meaning. Some examples of literal characters in regular expressions are:

- **Alphabetic characters:** The regular expression r'abc' matches the exact sequence of 'abc' in a string. For example,

```
import re
result = re.findall(r'abc', 'I like abc and abcdefg')
print(result)
# Output: ['abc', 'abc']
```

- **Numeric characters:** The regular expression r'123' matches the exact sequence '123' in a string. For example,

```
import re
result=re.findall(r"781022", "The pin number is 7810223456")
print(result)
# Output: ['781022']
```

- **Whitespace characters:** The regular expression r" " matches a literal space character. For example,

```
import re
result=re.findall(r" ", "Hello World")
print(result) # Output: [' ']
```

- **Punctuation characters:** The regular expression r"\." matches aA literal dot (.). The dot is escaped because dot (.) is a special character in regular expressions.

```
result = re.findall(r"\.", "Example.com")
print(result)  # Output: ['.']
```

Literal characters match themselves exactly, whereas special characters (like . or *) need to be escaped using a backslash (\) if we want to treat them literally.

2) Metacharacters:

Characters that have special meanings in regex. Some common metacharacters include:

Table 13.1: Metacharacters and their description

Metacharacters	Description	Example
.	Matches anything except a newline. It can be used where we wish to match any character.	a.b matches a+b, a-b
^	Matches beginning of a string.	^abc matches abc at start.
$	Anchors the match at the end of a string.	abc$ matches abc at the end.
*	Matches 0 or more occurrences of the preceding element.	ab*c matches ac, abc, abbc, abbbc, …
+	Matches 1 or more occurrences of the preceding element.	ab+c matches abc, abbc, abbbc, …
?	Matches 0 or 1 occurrence of the preceding element (makes it optional).	colou?r matches color or colour
[ ]	It is used to specify a character class, i.e., a set of characters that we wish to match.	For example, [xyz] matches any of "x", "y", or "z".
( )	Groups pattern for capturing and reusing	(abc)+ matches abcabc, abcabcabc,…
|	Alternation between regular expressions, acts like a logical OR.	a|b matches "a" or "b".

3) Special Sequence Characters in Regular Expressions:

In Python, sequence character match only one character in the string. Here are some of the commonly used sequence characters in regular expressions:

Table 13.2.: Special Sequence of Character in Regular Expressions

Character	Description
\d	Matches any digit. Equivalent to: [0-9].
\D	Matches any non-digit character. Equivalent to: [^0-9]
\w	Matches any alphanumeric character (letters, digits and underscore). Equivalent to: [a-zA-Z0-9_]. For example, word123_
\W	Matches any non-alphanumeric character. Equivalent to: [^a-zA-Z0-9_]
\s	Matches any whitespace character (spaces, tabs, newline, etc.). Equivalent to: [\t\n\r\f\v]
\S	Matches any non-whitespace character. Equivalent to: [^ \t\n\r\f\v]
\b	Represents a space around words.

Each of these sequence characters represent a single character matched in the string. For example, '\w' indicates any one alphanumeric character. If we write it as [w*], it represents 0 or more alphanumeric characters.

Let us write a regular expression to retrieve all words starting with 'b'.

r'b[\w]*'

Example 9: A Python program to create a regular expression to retrieve all words starting with a in a given string "'an apple a day keeps the doctor away."

```
import re
str='an apple a day keeps the doctor away.'
result=re.findall(r'a[\w]*', str)
for word in result:
    print(word)
```

```
C:\pythonscript>python program1.py
an
apple
a
ay
away
```

If we execute, it will give the output with all the words containing the letter a. But "ay" is meaningless. It is a part of the word away. We want only the words starting with 'a'. Since a word will have a blank space in the beginning or ending, we can use '\b' before and after the words in the regular expression. So, the actual regular expression will be:

result =re.findall(r'\ba[\w]*\b', str)

The fildall() returns a list, retrieving the elements from list.

The correct output will be:

```
C:\pythonscript>python program1.py
an
apple
a
away
```

Example 10: A Python program to create a regular expression to retrieve all words having 5 characters length of a string.

```
import re
text='Krishna Kanta Handiqui State Open University, Guwahati, Assam,India'
# findall() returns a list
result=re.findall(r'\b\w{5}\b', text)
print(result)
```

The output will a list of all five letter words of the string

```
C:\pythonscript>python program3.py
['Kanta', 'State', 'Assam', 'India']
```

Here, re.findall() function returns all non-overlapping matches of the pattern in the string, as a list.

Regular Expression is r'\b\w{5}\b'

where

\b - A word boundary. Ensures we're matching whole words, not substrings within words.

\w{5} - Matches exactly 5 word characters. \w includes letters (A-Z, a-z), digits (0-9), and the underscore (_).

\b - Another word boundary at the end.

So, the full pattern \b\w{5}\b matches any whole word that is exactly 5 characters long.

text is the input string in which the search is done.

4. Quantifiers:

In regular expressions, some characters represent more than one character to be matched in the string. Such characters are called quantifiers. Quantifiers specify how many instances of a character, group, or character class must be present for a match. Table 13.3 shows quantifiers available in Python:

Table 13.3: Quantifiers used in Regular Expressions

Character	Description
*	Matches 0 or more occurrences of the preceding element.
+	Matches 1 or more occurrences of the preceding element.
?	Matches 0 or 1 occurrence of the preceding element.
{n}	Matches exactly n occurrences of the preceding element.
{n,}	Matches at least n occurrences.
{m, n}	Matches between m and n occurrences.

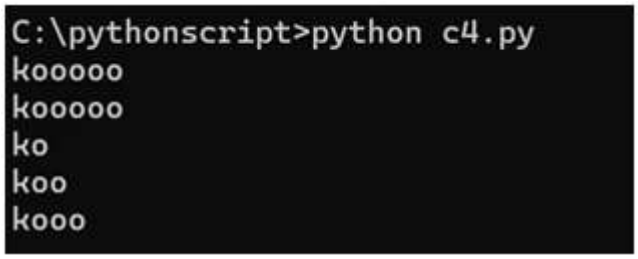
```
import re
pattern = r"ko*"
text = "kooooo"
match = re.search(pattern, text)
print(match.group())  # Output: kooooo

pattern = r"ko+"
match = re.search(pattern, text)
print(match.group())  # Output: kooooo

pattern = r"ko?"
match = re.search(pattern, text)
print(match.group())  # Output: ko
```

```
pattern = r"ko{2}"
match = re.search(pattern, text)
print(match.group())  # Output: koo

pattern = r"ko{3}"
match = re.search(pattern, text)
print(match.group())  # Output: kooo
```



5. Character Classes ([ ]):

Matches any one character inside the brackets.

```
import re
pattern = r"kkh[pqrst]ou"
text = "kkhsou"
match = re.search(pattern, text)
print(match.group())  # Output: kkhsou
```



6. Negated Character Classes ([^]):

Matches any character not in the brackets.

```
import re
pattern = r"h[^aeiou]llo"
text = "h5llo"
match = re.search(pattern, text)
print(match.group())  # Output: h5llo
```

7. Anchors (^ and $) :

^ matches the start of a string, $ matches the end.

```
import re
```

```
pattern = r"^hello"
text = "hello world"
match = re.search(pattern, text)
print(match.group())  # Output: hello

pattern = r"learner$"
text = "hello learner"
match = re.search(pattern, text)
print(match.group())  # Output: learner
```



8. Alternation (|):

Matches either the pattern on the left or the pattern on the right.

```
import re
pattern = r"cat|dog"
text = "I have a cat."
match = re.search(pattern, text)
print(match.group())    # Output: cat
```



9. Grouping and Capturing (()):

Groups multiple patterns and captures them as a single unit.

```
import re
pattern = r"(Welcome to) (KKHSOU)"
text = "Welcome to KKHSOU"
match = re.search(pattern, text)
print(match.group())  # Output:
print(match.group(1))  # Output: Welcome to
print(match.group(2))  # Output: KKHSOU
```

10. Word Boundaries (\b) in Regular Expressions:

In regular expressions, \b represents a word boundary, which is a position between a word character (letters, digits, or underscores) and a non-word character (such as space, punctuation, or the beginning/end of a string). It does not match a character, but rather the position where a word starts or ends.

```
import re
pattern = r"\bOpen\b"
text = " State Open University "
match = re.search(pattern, text)
print(match.group())  # Output: Open
```

\bOpen\b ensures the word "Open" is matched only when it stands alone, not as part of another word. In the text " State Open University ", the word "Open" is surrounded by spaces, which are non-word characters.Therefore, the word boundaries are valid, and the match is successful.

```
pattern = r"\bUniversity\b"
match = re.search(pattern, text)
print(match.group())  # Output: University
```

The pattern \bUniversity\b matches the exact word "University". As "University" is preceded and followed by a space in the text, both are considered valid word boundaries. Hence, the pattern matches and outputs "University".

```
pattern = r"\bOpen University\b"
match = re.search(pattern, text)
print(match.group())  # Output: Open University
```

This pattern looks for the phrase "Open University" as a whole, starting and ending at word boundaries. In the string " State Open University ", the phrase "Open University" appears exactly like that, with spaces before and after.

Both "Open" and "University" are complete words and the pattern matches successfully.

```
C:\pythonscript>python c7.py
Open
University
Open University
```

11. Escape Sequences (\):

Escapes special characters and denotes predefined character classes.

\d: Matches any digit (0-9).

\D: Matches any non-digit.

\w: Matches any word character (alphanumeric + _).

\W: Matches any non-word character.

\s: Matches any whitespace character (space, tab, newline).

\S: Matches any non-whitespace character.

```
import re
pattern = r"\d+"
text = "There are 250 students."
match = re.search(pattern, text)
print(match.group())  # Output: 250

match = re.findall(pattern, text)
print(match) # Output: ['250']

text = "I have 1 laptop and 2 pen drives."
match = re.findall(pattern, text)
print(match) # Output: ['1','2']

pattern = r"\D+" # Matches one or more non-digit
                 #characters
text = "I have 1 laptop and 2 pen drives."
match = re.findall(pattern, text)
print(match) # Output: ['I have ', ' laptop and ', ' pen drives.']
```

```
pattern = r"\w+"
text = "Dear Learner!"
match = re.search(pattern, text)
print(match.group())  # Output: Dear

pattern = r"\W+" # Matches one or more non-word characters
text = "Dear Learner!"
match = re.findall(pattern, text)
print(match) # Output: [' ', '!']

pattern = r"\s+" # Matches one or more whitespace characters
text = "Hello Learner \t Welcome to Python!"
match = re.findall(pattern, text)
print(match)  # Output: [' ', ' \t ', ' ']

pattern = r"\S+"  # Matches one or more non-whitespace characters
text = "Hello Learner \t Welcome to Python!"
match = re.findall(pattern, text)
print(match)  # Output: ['Hello', 'Learner', 'Welcome', 'to', 'Python!']
```

```
C:\pythonscript>python c8.py
250
['250']
['1', '2']
['I have ', ' laptop and ', ' pen drives.']
Dear
[' ', '!']
[' ', ' \t ', ' ', ' ']
['Hello', 'Learner', 'Welcome', 'to', 'Python!']
```

12. Lookahead ((?=...)) and Lookbehind ((?<=...)):

Lookahead ensures that a certain pattern follows the current position, while lookbehind ensures that a pattern precedes it.

```
import re
```

```
pattern = r"hello(?=\sworld)"
text = "hello world"
match = re.search(pattern, text)
print(match.group())  # Output: hello

pattern = r"(?<=hello\s)world"
match = re.search(pattern, text)
print(match.group())  # Output: world
```

```
C:\pythonscript>python c9.py
hello
world
```

13. Non-Capturing Groups ((?:...)):

Groups without capturing the content for reference.

```
import re
pattern = r"(?:Let us) learn Python"
text = "Let us learn Python Programming."
match = re.search(pattern, text)
print(match.group())  # Output: Let us learn Python

match = re.findall(pattern, text)
print(match)  # Output: [Let us learn Python]
```

```
C:\pythonscript>python c10.py
Let us learn Python
['Let us learn Python']
```

These common regex elements, along with examples in Python, will help us to construct patterns for matching and manipulating text.

13.6 PRACTICAL APPLICATIONS

Whether we are validating emails, extracting information, or searching through large datasets, understanding and mastering regular expressions will greatly enhance our programming skills.

1. Validating Email Addresses:

We can use a regular expression to check if a string is a valid email address.

```
import re
pattern = r'^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$'
email = 'rohit.sarma@gmail.com'

if re.match(pattern, email):
    print("Valid email address")
else:
    print("Invalid email address")

email = 'virat@delhi'
if re.match(pattern, email):
    print("Valid email address")
else:
    print("Invalid email address")
```

Output:

```
C:\pythonscript>python email.py
Valid email address
Invalid email address
```

2. Extracting Dates from Text

We can extract dates in a specific format from text using regular expressions.

```
import re
pattern = r'\b\d{2}/\d{2}/\d{4}\b'
text = 'The interview is scheduled for 12/06/2024 and 16/06/2024.'
dates = re.findall(pattern, text)
print("Interview dates:", dates)
```

Output:

```
C:\pythonscript>python dates.py
Interview dates: ['12/06/2024', '16/06/2024']
```

1. Phone Number Validation

```
import re
pattern = r"\(\d{2}\) \d{5}\d{5}"
text = "My phone number is (91) 9435055441."
match = re.search(pattern, text)
if match:
    print("Phone number is:", match.group())

# Output: Phone number: (91) 94350 55441
```

```
C:\pythonscript>python phone.py
Phone number is: (91) 9435055441
```



CHECK YOUR PROGRESS

Q1. Choose the correct answer of the following MCQs

i. Which module in Python is used for regular expressions?

a) regex
b) string
c) re
d) pattern

ii. What does the \d in a regular expression pattern match?

a) A digit
b) A non-digit
c) A word character
d) A whitespace character

iii. Which function returns all non-overlapping matches of a pattern in a string?

a) re.match()
b) re.search()
c) re.findall()
d) re.replace()

iv. What is the output of re.match(r"\d+", "123abc")?

a) None

b) Match object

c) True

d) False

v. Which pattern matches a string that ends with 'abc'?

a) ^abc

b) abc$

c) abc^

d) $abc

vi. What does re.sub(r"apple", "mango", "apple pie") return?

a) apple pie

b) mango pie

c) apple

d) mango

vii. Which of the following matches a single whitespace character?

a) \w

b) \d

c) \s

d) \S

viii. Which regex matches any character except a newline?

a) .

b) .

c) \n

d) ^

ix. What will re.findall(r'\b\w{5}\b', "Hello world again after long time") return?

a) ['Hello', 'world']

b) ['again', 'after']

c) ['long', 'time']

d) ['Hello', 'world', 'again']

x. What does the ^ metacharacter signify in regular expressions?

a) End of string

b) Start of string

c) Any digit

d) Repetition



13.7 LET US SUM UP

Python's re module provides a powerful toolkit for working with regular expressions, enabling efficient pattern matching, validation, and text manipulation. Key functions offered by this module include re.match(), re.search(), re.findall(), re.split(), and re.sub(), each serving a specific purpose in string processing:

- **re.match():** Checks for a match only at the beginning of the string.
- **re.search():** Searches for the first occurrence of a pattern anywhere in the string.
- **re.findall():** Returns all non-overlapping matches of a pattern as a list.
- **re.split():** Splits the string at each occurrence of the pattern.
- **re.sub():** Substitutes occurrences of the pattern with a replacement string.

Regular expressions are composed using a combination of **literals, metacharacters** (., ^, $, *, +, ?, [], (), |), **special sequences** (\d, \w, \s), **anchors** (^ for start and $ for end), as well as **lookaheads, lookbehinds,** and **groups** to build complex patterns.

These patterns are extensively used in practical applications such as validating email addresses, extracting dates, and verifying phone number formats. Regular expressions provide a concise and flexible means for handling text data and are commonly used in areas such as **web development, data validation,** and **text processing.**

By mastering the basic syntax and functionalities of Python's re module, learners can build a strong foundation in regular expressions.



13.8 ANSWERS TO CHECK YOUR PROGRESS

Ans.Qu.No.1

i. c) re - The re module in Python is used for working with regular expressions.

ii. a) A digit - The metacharacter \d matches any single digit (0-9).

iii. c) re.findall() - This function returns all non-overlapping matches of a pattern in a string.

iv. b) Match object - re.match(r"\d+", "123abc") matches one or more digits at the start and returns a match object.

v. b) abc$ - The pattern abc$ matches strings that end with 'abc'.

vi. b) mango pie - re.sub() replaces all occurrences of 'apple' with 'mango' in the input string.

vii. c) \s - This metacharacter matches any single whitespace character like space, tab, or newline.

viii. b) . - The dot . matches any character except a newline by default.

ix. a) ['Hello', 'world'] - The pattern \b\w{5}\b finds all 5-letter whole words in the string.

x. b) Start of string - The caret ^ is used to match the beginning of a string in regular expressions.



13.9 FURTHER READING

1. Kanetkar, Y., & Kanetkar, A. (2020). Let Us Python: Python Is Future, Embrace It Fast (Vol. 2). BPB Publications.



13.10 MODEL QUESTIONS

1. What is a regular expression? How is it useful in text processing?
2. Explain the function of the following regex symbols: ^, $, *, +, and [].
3. Write a Python program using regular expressions to extract all phone numbers from a given text.
4. How does the re.sub() function work? Provide an example.

Short Answer Questions (2-4 marks)

5. Define a regular expression. Give an example.
6. What is the use of the re.match() function in Python?
7. Write a regex pattern to validate an email address.
8. Explain the difference between re.match() and re.search().

9. What is the role of \b in regex? Give an example.
10. List any four special sequences used in regular expressions and their meaning.
11. Write a short note on quantifiers used in Python regex.
12. What does the re.sub() function do? Write a sample code.

Long Answer Questions (6-10 marks)

13. Explain the use of various metacharacters in regular expressions with examples.
14. Write a Python program to extract all phone numbers from a given text using regex.
15. Discuss the role of grouping () and alternation | in Python regex with code.
16. Describe how to extract dates in the format dd/mm/yyyy using regex in Python.
17. What are character classes in regular expressions? Explain with examples.
18. Explain the difference between re.findall() and re.split() with examples.
19. Demonstrate how lookahead and lookbehind assertions work in regular expressions using Python code.

****

UNIT 14: MODULES AND PACKAGES

UNIT STRUCTURE

14.1 Learning Objectives

14.2 Introduction

14.3 Modules

14.4 Packages

14.5 Python Standard Library

14.6 Summing Up

14.7 Answers to Check Your Progress

14.8 References

14.9 Further Readings

14.10 Model Questions

14.1 LEARNING OBJECTIVES

By the end of this unit, you will be able to:

- Understand the concept of modules and how to create and import them.
- Use the import and from...import statements to access module functionality.
- Understand the concept of packages and how to create and use them.
- Explore the Python Standard Library and use commonly used modules.
- Write modular and reusable code using modules and packages.

14.2 INTRODUCTION

In Python, modules and packages are essential tools for organizing and reusing code. A module is a file containing Python code (e.g., functions, classes, or variables) that can be imported and used in other programs. A package is a collection of modules organized in a directory hierarchy. Packages allow you to structure your code into logical and reusable components. This unit will introduce you to modules and packages, explain how to create and use them, and explore some commonly used modules from the

Python Standard Library.

14.3 MODULES

A module is a file containing Python definitions and statements. The file name is the module name with the .py extension. Modules allow you to organize your code into reusable components.

Example 1:

```
# my_module.py
def greet(name):
    return f"Hello, {name}!"

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

Explanation:

- The above code defines a module named my_module with two functions: greet() and add().
- This module can be imported and used in other Python programs.

(a) Creating and Importing Modules

To use a module, you need to import it into your program using the import statement.

Example 2:

```
# main.py
import my_module

print(my_module.greet("Alice"))  # Output: Hello, Alice!
print(my_module.add(3, 5))       # Output: 8
```

Explanation:

- The import statement is used to import the my_module module.
- Once imported, you can access its functions using the

module_name.function_name syntax.

(b) Standard Library Modules

Python comes with a rich Standard Library that provides many useful modules. Some commonly used modules include:

- math: Mathematical functions.
- random: Random number generation.
- os: Operating system interactions.
- datetime: Date and time manipulation.

Example 3:

```
import math

print(math.sqrt(16))  # Output: 4.0
```

Explanation:

- The math module provides mathematical functions like sqrt() for calculating the square root.

(c) The import Statement

The import statement is used to import an entire module.

Example 4:

```
import math

print(math.pi)  # Output: 3.141592653589793
```

Explanation:

- The math module is imported, and the pi constant is accessed using math.pi.

(d) The from...import Statement

The from...import statement allows you to import specific functions or vari-

ables from a module.

Example:

```
from math import pi, sqrt

print(pi)       # Output: 3.141592653589793
print(sqrt(25))  # Output: 5.0
```

Explanation:

- Only the pi and sqrt functions are imported from the math module.
- You can directly use these functions without prefixing them with the module name.

(e) The dir() Function

The dir() function lists all the names defined in a module.

Example 5:

```
import math

print(dir(math))
```

Output:

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos',
'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'comb', 'copysign', 'cos',
'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor',
'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf',
'isnan', 'isqrt', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan',
'perm', 'pi', 'pow', 'prod', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh',
'tau', 'trunc']
```

Explanation:

- The dir() function lists all the functions, classes, and variables defined

in the math module.

14.4 PACKAGES

A package is a collection of modules organized in a directory hierarchy. Packages allow you to structure your code into logical and reusable components.

Example Directory Structure:

```
my_package/
   __init__.py
   module1.py
   module2.py
   subpackage/
      __init__.py
      module3.py
```

(a) Creating and Using Packages

To create a package, you need to include an __init__.py file in the directory. This file can be empty or contain initialization code.

Example 6:

```
# my_package/module1.py
def greet(name):
   return f"Hello, {name}!"

# my_package/module2.py
def add(a, b):
   return a + b
```

Explanation:

- The my_package directory contains two modules: module1.py and module2.py.
- The __init__.py file makes Python treat the directory as a package.

The __init__.py File

The __init__.py file is required to make Python treat the directory as a package. It can also be used to initialize package-level variables or import submodules.

Example 7:

```
# my_package/__init__.py
from .module1 import greet
from .module2 import add
```

Explanation:

- The __init__.py file imports the greet and add functions from module1 and module2, respectively.

(b) Importing from Packages

You can import modules or specific functions from a package.

Example 8:

```
# main.py
from my_package import greet, add

print(greet("Alice"))  # Output: Hello, Alice!
print(add(3, 5))       # Output: 8
```

Explanation:

- The greet and add functions are imported from the my_package package and used in the program.

14.5 PYTHON STANDARD LIBRARY

The Python Standard Library comes with many built-in modules that make programming easier by providing ready-to-use functions and classes. Below are some of the most commonly used modules:

1. math Module - Mathematical Functions

The math module provides access to mathematical functions like square roots, powers, trigonometry, logarithms, and constants such as ? and e.

```
import math

# Square root
print("Square root of 16:", math.sqrt(16))   # Output: 4.0

# Power
print("2 raised to power 3:", math.pow(2, 3))  # Output: 8.0

# Trigonometric function
print("cos(0):", math.cos(0))  # Output: 1.0

# Mathematical constant
print("Value of pi:", math.pi)  # Output: 3.141592653589793
```

Explanation:

- sqrt() finds square root.
- pow(a, b) computes a^b.
- Trigonometric functions like sin(), cos(), tan() are available.
- Constants like math.pi and math.e are useful in calculations.

2. random Module - Random Number Generation

The random module is used for generating random numbers, selecting random items, and shuffling sequences.

```
import random

# Generate a random integer between 1 and 10
print("Random integer (1-10):", random.randint(1, 10))
```

```
# Generate a random float between 0 and 1
print("Random float:", random.random())
# Choose a random element from a list
fruits = ["apple", "banana", "cherry", "mango"]
print("Random fruit:", random.choice(fruits))

# Shuffle a list randomly
random.shuffle(fruits)
print("Shuffled fruits:", fruits)
```

Explanation:

- randint(a, b) returns a random integer between a and b.
- random() gives a float between 0 and 1.
- choice() picks a random element from a sequence.
- shuffle() rearranges items in a list randomly.

3. os Module - Operating System Interactions

The os module allows interaction with the operating system, such as working with files, directories, and environment variables.

```
import os

# Get current working directory
print("Current directory:", os.getcwd())

# List files in current directory
print("Files in directory:", os.listdir())

# Create a new folder
os.mkdir("test_folder")
print("Folder 'test_folder' created.")

# Rename a folder
os.rename("test_folder", "renamed_folder")
print("Folder renamed to 'renamed_folder'.")
```

```
# Remove a folder
os.rmdir("renamed_folder")
print("Folder removed.")
```

Explanation:

- getcwd() gives the current directory path.
- listdir() shows all files and folders in a directory.
- mkdir() creates a folder, rename() renames it, and rmdir() removes it.

4. datetime Module - Date and Time Manipulation

The datetime module is used to work with dates and times. It allows you to get the current date/time, format them, and perform calculations.

```
from datetime import datetime, timedelta

# Get current date and time
now = datetime.now()
print("Current date and time:", now)

# Format date and time
print("Formatted date:", now.strftime("%Y-%m-%d"))
print("Formatted time:", now.strftime("%H:%M:%S"))

# Extract specific components
print("Year:", now.year)
print("Month:", now.month)
print("Day:", now.day)

# Date arithmetic using timedelta
yesterday = now - timedelta(days=1)
tomorrow = now + timedelta(days=1)
print("Yesterday:", yesterday)
print("Tomorrow:", tomorrow)
```

Explanation:

- datetime.now() gets the current system date and time.
- strftime() converts date/time to a formatted string (e.g., YYYY-MM-DD).
- Components like year, month, day, hour, etc., can be accessed individually.
- timedelta helps in adding or subtracting days, hours, or minutes.

5. sys Module - System-Specific Parameters and Functions

The sys module provides access to system-level functions and variables used to manipulate the Python runtime environment.

```
import sys

# Python version
print("Python version:", sys.version)

# Command-line arguments
print("Command-line arguments:", sys.argv)

# Exit program
# sys.exit()  # Uncomment to exit the script
```

```
import sys

# Python version
print("Python version:", sys.version)

# Command-line arguments
print("Command-line arguments:", sys.argv)

# Exit program
# sys.exit()  # Uncomment to exit the script
```

Explanation:

- sys.version returns the Python version being used.
- sys.argv stores command-line arguments passed to the program.
- sys.exit() is used to terminate the program.

6. statistics Module - Statistical Calculations

The statistics module provides functions to perform statistical operations like mean, median, mode, and standard deviation.

```
import statistics

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

print("Mean:", statistics.mean(data))       # Average
print("Median:", statistics.median(data))   # Middle value
print("Mode:", statistics.mode(data))       # Most common value
print("Standard Deviation:", statistics.stdev(data))  # Spread of data
```

Explanation:

- mean() calculates the average.
- median() finds the middle value.
- mode() gives the most frequent value.
- stdev() calculates standard deviation, showing how spread out data is.

7. json Module - Working with JSON Data

The json module is used for reading and writing JSON (JavaScript Object Notation) data, which is widely used in web APIs and data storage.

```
import json

# Python dictionary
student = {
    "name": "Alice",
    "age": 21,
    "course": "Computer Science"
```

```
}

# Convert dictionary to JSON string
json_string = json.dumps(student)
print("JSON String:", json_string)

# Convert JSON string back to Python dictionary
python_dict = json.loads(json_string)
print("Python Dictionary:", python_dict)

# Write JSON data to a file
with open("student.json", "w") as file:
    json.dump(student, file)

# Read JSON data from a file
with open("student.json", "r") as file:
    data = json.load(file)
    print("Data from file:", data)
```

Explanation:

- dumps() converts Python objects into JSON strings.
- loads() converts JSON strings back into Python objects.
- dump() writes JSON data to a file, while load() reads JSON from a file.

8. re Module - Regular Expressions

The re module is used for pattern matching, searching, and replacing text using regular expressions.

```
import re

text = "Python is amazing. Python is powerful."

# Find all occurrences of "Python"
matches = re.findall("Python", text)
```

```
print("Matches:", matches)   # ['Python', 'Python']

# Search for the first occurrence
search_result = re.search("amazing", text)
print("Found at position:", search_result.start())

# Replace text
new_text = re.sub("Python", "Java", text)
print("After replacement:", new_text)
```

Explanation:

- findall() returns all matches.
- search() finds the first occurrence.
- sub() replaces text based on pattern.

9. collections Module - Specialized Data Structures

The collections module provides additional data structures like Counter, defaultdict, namedtuple, deque.

```
from collections import Counter, defaultdict, deque, namedtuple

# Counter - counts occurrences
fruits = ["apple", "banana", "apple", "cherry", "banana", "apple"]
print("Counter:", Counter(fruits))

# defaultdict - provides default value for missing keys
scores = defaultdict(int)
scores["Alice"] += 10
print("Defaultdict:", scores)

# deque - double-ended queue
dq = deque([1, 2, 3])
dq.appendleft(0)
dq.append(4)
```

```
print("Deque:", dq)

# namedtuple - lightweight object with named fields
Student = namedtuple("Student", ["name", "age"])
s1 = Student("Bob", 20)
print("Namedtuple:", s1.name, s1.age)
```

Explanation:

- Counter counts frequencies.
- defaultdict gives a default value when key doesn't exist.
- deque allows fast adding/removing elements from both ends.
- namedtuple is like a tuple but with named fields.

10. time Module - Time-Related Functions

The time module deals with time representation, delays, and performance measurement.

```
import time

# Get current time in seconds since epoch
print("Current time:", time.time())

# Sleep (pause execution)
print("Waiting for 2 seconds...")
time.sleep(2)
print("Done waiting!")

# Measure execution time
start = time.time()
for i in range(1000000):
    pass
end = time.time()
print("Loop execution time:", end - start, "seconds")
```

Explanation:

- time() gives the current time in seconds since 1970 (epoch).
- sleep() pauses execution for given seconds.
- Useful for performance measurement with start & end times.

11. urllib Module - Working with URLs

The urllib module helps in fetching data from the web (basic web requests).

```
import urllib.request

# Fetch data from a URL
url = "https://www.example.com"
response = urllib.request.urlopen(url)

# Print status and content
print("Status Code:", response.status)
print("First 100 characters:", response.read(100).decode("utf-8"))
```

Explanation:

- urllib.request.urlopen() opens a URL and fetches response data.
- read() fetches content, and .decode("utf-8") converts it to text.

12. itertools Module - Iterators for Efficient Looping

The itertools module provides powerful tools for handling iterators, especially useful for combinatorics and efficient looping.

```
import itertools

# Infinite counter
counter = itertools.count(start=1, step=2)
print(next(counter))  # 1
print(next(counter))  # 3

# Cycle through a sequence
cycle_items = itertools.cycle(["A", "B", "C"])
```

```
print(next(cycle_items))  # A
print(next(cycle_items))  # B

# Repeat an element
for x in itertools.repeat("Hello", 3):
    print(x)

# Combinations
items = [1, 2, 3]
print("Combinations of 2:", list(itertools.combinations(items, 2)))

# Permutations
print("Permutations of 2:", list(itertools.permutations(items, 2)))
```

Explanation:

- count() generates infinite sequence.
- cycle() loops through items endlessly.
- repeat() repeats a value multiple times.
- combinations() & permutations() generate possible selections.

13. functools Module - Higher-Order Functions

The functools module provides functions that act on or return other functions (functional programming style).

```
import functools

# reduce() - applies a function cumulatively
numbers = [1, 2, 3, 4]
product = functools.reduce(lambda x, y: x * y, numbers)
print("Product of numbers:", product)

# lru_cache - caches function results
@functools.lru_cache(maxsize=3)
def fibonacci(n):
```

```
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

print("Fibonacci(10):", fibonacci(10))

# partial - pre-fills function arguments
def multiply(a, b):
    return a * b

double = functools.partial(multiply, 2)
print("Double of 5:", double(5))
```

Explanation:

- reduce() performs cumulative operations.
- lru_cache improves performance by caching function calls.
- partial() creates new functions with preset arguments.

14. logging Module - Logging Messages

The logging module is used for tracking events in programs, useful for debugging and monitoring.

```
import logging

# Configure logging
logging.basicConfig(level=logging.INFO)

logging.debug("This is a debug message")   # Not shown (level too low)
logging.info("This is an info message")    # Shown
logging.warning("This is a warning")       # Shown
logging.error("This is an error")          # Shown
logging.critical("This is critical")       # Shown
```

Explanation:

- Logging levels: DEBUG < INFO < WARNING < ERROR < CRITICAL.
- Helps in debugging and monitoring without using print().
- Output can be sent to files, consoles, or remote servers.

15. shutil Module - File Operations

The shutil module provides high-level operations on files and directories (copy, move, delete, archive).

```
import shutil
import os

# Create a file for demo
with open("demo.txt", "w") as f:
    f.write("Hello, World!")

# Copy file
shutil.copy("demo.txt", "copy_demo.txt")
print("File copied.")

# Move file
shutil.move("copy_demo.txt", "moved_demo.txt")
print("File moved.")

# Create archive (zip)
shutil.make_archive("backup", "zip", ".")
print("Archive created: backup.zip")

# Remove a file
os.remove("demo.txt")
os.remove("moved_demo.txt")
print("Files deleted.")
```

Explanation:

- copy() copies files.
- move() moves/renames files.
- make_archive() creates compressed archives.
- Useful for backup and deployment tasks.



CHECK YOUR PROGRESS

Choose the correct option:

Q1. Which of the following best describes a Python module?

a) A collection of directories containing Python code

b) A file containing Python definitions and statements with a .py extension

c) A built-in function in Python

d) A special variable used for initialization

Q2. What is the purpose of the __init__.py file in a Python package?

a) To store constants and variables only

b) To make Python treat the directory as a package

c) To automatically import the math module

d) To list all functions inside a module

Q3. Which statement is correct about the from…import syntax in Python?

a) It imports the entire module into the program.

b) It imports specific functions or variables from a module.

c) It only works for built-in modules.

d) It automatically lists all attributes of a module.

Q4. Which of the following modules is used for date and time manipulation in Python?

a) os

b) time

c) datetime
d) sys

Q5. Which function from the random module selects a random element from a list?
a) random()
b) randint()
c) choice()
d) shuffle()



14.6 SUMMING UP

- A module in Python is a .py file containing code such as functions, classes, or variables.
- Modules are imported using the import statement or from…import for specific components.
- The dir() function lists all names (functions, classes, variables) in a module.
- A package is a collection of modules organized in a directory with an __init__.py file.
- The __init__.py file makes a directory a package and can initialize package-level imports.
- Commonly used Standard Library modules include: math, random, os, datetime, sys, statistics, json, re, collections, time, urllib, itertools, functools, logging, and shutil.
- The math module provides mathematical functions and constants like sqrt(), pow(), and pi.
- The random module is used for generating random numbers, selecting random items, and shuffling sequences.
- The os module allows file and directory operations like creating, renaming, and removing files/folders.
- The datetime module helps with getting the current date/time, formatting, and performing date/time arithmetic.



14.7 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q1: b)

Ans. to Q2: b)

Ans. to Q3: b)

Ans. to Q4: c)

Ans. to Q5: c)

14.8 REFERENCES

- Python Documentation: https://docs.python.org/3/tutorial/modules.html

14.9 FURTHER READINGS

- Kanetkar, Y., & Kanetkar, A. (2023). Let us Python. BPB Publications.
- https://www.w3schools.com/python/

14.10 MODEL QUESTIONS

1. Explain the concept of a Python module. How do modules help in organizing and reusing code? Illustrate with an example.
2. Describe the steps involved in creating and importing a module in Python. Provide a code example using a simple function.
3. What is the purpose of the from…import statement? How does it differ from the regular import statement? Provide examples.
4. Explain the role of the dir() function in Python. How can it be used to explore the contents of a module?
5. Define a Python package. How is it different from a module? Explain with an example of a package containing multiple modules.
6. What is the significance of the init.py file in a Python package? How can it be used to import submodules or initialize package-level variables?
7. Discuss the functionalities of the math and random modules in Python with examples for each.

8. How can the os and datetime modules be used to work with the file system and manipulate dates/times? Provide practical examples.
9. Explain the usage of the sys and statistics modules in Python. Provide examples showing system-specific information and statistical calculations.
10. Describe the purpose and usage of modules such as json, re, collections, time, itertools, and functools in Python. Provide one example for each module.

****

UNIT 15: GUI PROGRAMMING

UNIT STRUCTURE

15.1 Learning Objectives

15.2 Introduction

15.3 GUI Programming

15.4 Tkinter library for GUI programming

15.5 Using Canvas

15.6 Using Frames

15.7 Let Us Sum Up

15.8 Answers to Check Your Progress

15.9 Further Reading

15.10 Model Questions

15.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- explain the concept of GUI programming
- use tkinter library for GUI programming in Python
- write programs to display various shapes like line, arc, oval, polygon in canvas
- write programs to display images in canvas
- write programs using various widgets like button, text, message, label and scrollbar
- write programs using various widgets like checkbutton, radiobutton and listbox

15.2 INTRODUCTION

In this unit, you will be introduced to the concept of GUI programming in Python. In the last unit, the concept of modules and packages was covered. Topics like creating and importing modules in Python and using third-party packages in Python were covered in detail in the last unit. This unit covers how tkinter library module is used for GUI programming in Python.

The two important parts of GUI programming canvas and frames are covered in detail in this unit. The unit covers how to add components of different shapes like line, arc, oval, rectangle, polygon and text in a canvas. In addition to these components, the unit also covers he addition of images to the canvas. The different widgets that can be used in a frame like button, label, text, message, radiobutton, checkbutton, and listbox are covered in detail in this unit. In the next unit, we will explore the concepts of advanced Python including multi-threading, networking, and web programming.

15.3 GUI PROGRAMMING

GUI (Graphical User Interface) programming allows users to interact with applications using graphical elements like buttons, menus, text boxes, etc., rather than typing commands in the console. There are various advantages to using GUI in our applications. Some of them are:

- GUI programs offer a user friendly way in which even a layman can use the application.
- GUI programs can also add beauty to the application by adding colors, images, menus and animations.
- GUI offers graphical components like buttons, labels etc.

In Python, GUI programming is widely supported through several libraries and modules. The most common module to create graphics program in Python is Tkinter. Let us look into tkinter module in detail in the next section.

15.4 TKINTER LIBRARY FOR GUI PROGRAMMING

Tkinter stands for "Toolkit Interface", which serves as Python's standard library for creating Graphical User Interfaces (GUIs). It acts as a bridge between Python and the Tk GUI toolkit, which is a part of the TCL/TK programming language.

With Tkinter, Python developers can access and use the powerful GUI features provided by the Tk module of TCL/TK. This interface allows the creation of various graphical elements such as buttons, labels, text boxes,

menus, etc. in a Python program. Tkinter simplifies the process of building GUI-based desktop applications by providing a Python-friendly way to work with the robust components of the underlying TK toolkit. Since Tkinter is included with most Python installations, it is widely used by for teaching GUI development in Python.

The first step in developing a GUI program is to create the root window. The root window is the top level window that provides rectangular space on the screen where we can display text, images, components etc. We allocate space for our use in the root window by creating a canvas or a frame. Canvas is used for displaying drawing like lines, arcs and various other shapes. Frame is used for displaying components like push buttons, check buttons, menus etc. These components are called widgets.

The Root Window:

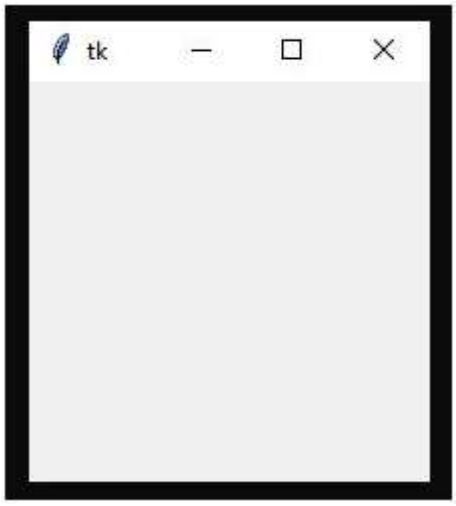
The space allocated to every GUI program is called the "root window" or the "top level window". This is the highest level GUI component in any tkinter application.

Example 1: Program to create root window

```
from tkinter import*
root = Tk()
root.mainloop()
```

The first line of the program is used to import all the components form the tkinter module. In the second line we get the root window by creating an object to Tk class. The third line helps to keep the application window open and waits for user interactions like button clicks, key presses, or window events (like closing or resizing). Without it, the GUI window would open and close immediately.

Output:

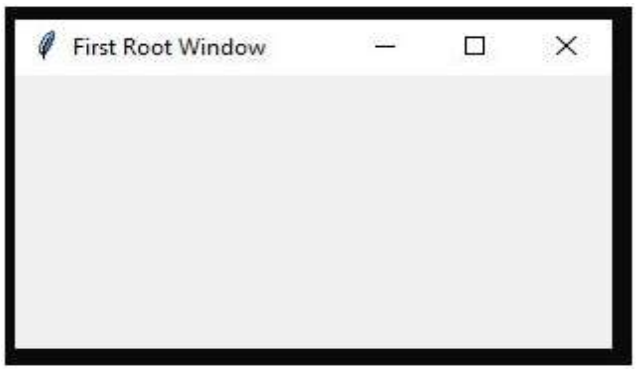


We can see in the output that the root window has a title bar and a default title "tk" and also has options for minimize, resize and close. We can close the window by clicking on the close "X" option. We can modify the code in Program 15.1 to add our own title and also set the size of the root window.

Example 2: Program to create root window with modifications

```
from tkinter import*
root = Tk()
root.title("First Root Window")
root.geometry("300x150")
root.mainloop()
```

Output:



In this output, we can see that we have modified the name of the title of the root window to "First Root Window" and set the size of the root window to 300x150 pixels.

Fonts and Colors:

Tkinter also has various options for adding different fonts and colors to the application. A font can be applied by using a tuple that contains the font family name, size and font style. For example:

fnt = ("Arial",-22,"bold")

In this example, the font style is "Arial" and the font size is 22. If the size is a positive number, it represents the size in points and if it is in negative then it represents size in pixels. The style of the font can be 'bold', 'italic', and 'underline'. All fonts are available in the tkinter.font module inside tkinter module.

Colors can be displayed by mentioning their names directly in the tkinter modoule. Colors name that are available for use include: red, green, blue, yellow, black, white, magenta, cyan, pink, purple, orange and many more. We can also specify the colors using hexadecimal format. For example, #000000 represents black while #00ff00 represents green where the colors are represented in the format #rrggbb for 8 bits per color representing red, green and blue.

15.5 USING CANVAS

A canvas is a rectangular area which can be used for drawing pictures like lines, circles, rectangles, polygons, arcs etc. We can also use canvas to include images.

To create a canvas, we need to create an object to Canvas class as shown in the example below:

c = Canvas(root, bg="purple", height=300, width=150, cursor="arrow")

Here, "c" is the object to the class "Canvas", "root" is the name of the parent window, "bg" represents background color, "height" and "width" represent the height and width of the canvas in pixels respectively. When the monitor screen resolution is 1360x768, it means that the screen can ac-

commodate 1360 pixels width-wise and 768-pixels height wise. The option "cursor" represents the shape of the cursor in the canvas. Some of the options for the shape of the cursor are: arrow, circle, clock, cross, dot, hand1, hand2, heart, pencil, mouse, star, watch among many others.

Once the canvas is created it must be added to the root window. This is done by using the pack() method. To add the above canvas object "c" we can write the code:

```
c.pack()
```

After the canvas is added, we can draw any shapes on the canvas.

Line:

```
l1 = c.create_line(40,40,200,200, width=4, fill="green")
```

To create a line, we use the create_line() method. This code creates a line by connecting the points (40,40) and (200,200) where the "width" represents the width of the line which is 4 here, and "fill" represents the color of the line which is given as green here. The default "width" value is 1.

Oval:

```
o1 = c.create_oval(10, 10, 120, 100, width=2, fill="pink", outline="cyan")
```

To create a oval, we use the create_oval() method. We give the top left (10,10) and the bottom lower coordinates (120,100) to create an oval in the rectangular area. If the rectangle has the same width and height then the oval becomes a circle. "fill" represents the color to fill the oval whereas "outline" represents the color of the border or outline of the oval.

Polygon:

```
p1 = c.create_polygon(80, 80, 200, 200, 300, 200, width=2, fill="purple",
outline="white", smooth=1,  activefill="lightblue")
```

To create a polygon, we use the create_polygon() method. The polygon is created using the points (80, 80), (200, 200), (300, 200). The option 'width' represents the width of the line. The option 'fill' represents the colour inside the polygon and the option 'outline' represents the colour of the border of the polygon. The option 'smooth' when 0 represents the sharp edges whereas when 1 represents smooth edges. The option 'activefill' represents the color of the polygon when the cursor is placed inside the polygon.

Rectangle:

```
r1 = c.create_rectangle(330, 10, 400, 100, width=5, fill="royalblue", outline="green")
```

To create a rectangle, we use the create_rectangle() method. We give the top left (330,10) and the bottom lower coordinates (400,100) to create a rectangle. The option 'fill' represents the colour of the rectangle and the option 'outline' gives the colour of the border of the rectangle.

Arc:

```
a1 = c.create_arc(100,100,400,300, width=4, start= 270, extent=180, outline="yellow", style="arc")
```

To create an arc, we use the create_arc() method. An arc represents part of an ellipse or a circle. Here, the arc is created in the rectangular space defined by coordinates (100,100) and (400,300). The option "width" represents the width of the arc. The option "start" is 270 represents where the arc will start and the option "extent" represents where the arc will extent. Here, the arc starts at 270 degrees and is then extended for another 180 degrees. The angles are taken in counter clock-wise direction. The option "outline" gives the colour of the arc. The option "style" represents the style of the arc where it can be arc, pie or chord.

Text:

```
fnt = ("Arial",-22,"bold")
t1 = c.create_text(150,320, text="Canvas", font=fnt, fill="white", activefill="blue")
```

To display text in the canvas, we use the create_text() method. The location (150,320) for the text to be displayed is given. The option "text" represents the text we want to appear on the canvas. The option "font" represents the font details selected and here we assign it value of tuple "fnt". Font details include the font family name, font size and font style. In this example, we use the font family name "Arial", font size "22" and font style as "bold". The option "fill" represents the colour of the text and the option

"activefill" represents the colour of the text when the cursor is placed over the text.

Example 3: Program to display various shapes in canvas

```
from tkinter import*
root = Tk()
root.title("Canvas")

c = Canvas(root, bg="purple", height=400, width=500, cursor="arrow")

# create a line
l1 = c.create_line(100, 350, 310, 350, width=4, fill="gray")
# create a oval

o1 = c.create_oval(10, 10, 120, 100, width=2, fill="pink", outline="cyan")
# create a polygon

p1 = c.create_polygon(80, 80, 200, 200, 300, 200, width=2, fill="purple",
outline="white", smooth=1,  activefill="lightblue")

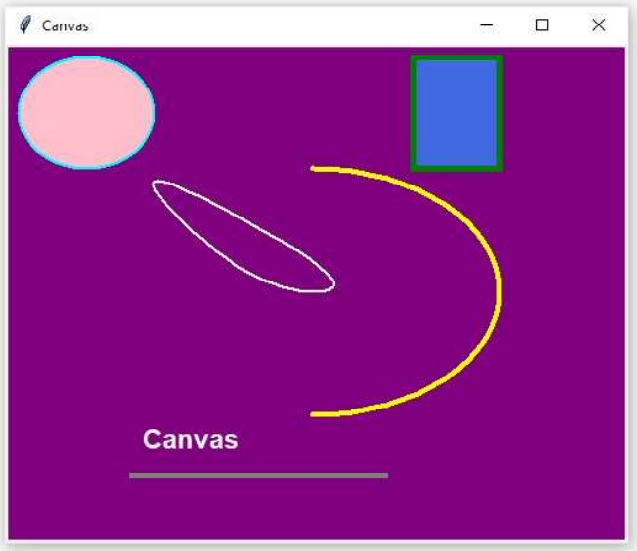
#create a rectangle
r1 = c.create_rectangle(330, 10, 400, 100, width=5, fill="royalblue",
outline="green")

#create an arc
a1 = c.create_arc(100,100,400,300, width=4, start= 270, extent=180,
outline="yellow", style="arc")
#create text
fnt = ("Arial",-22,"bold")
t1 = c.create_text(150,320, text="Canvas",
font=fnt,fill="white",activefill="blue")

c.pack()
```

root.mainloop()

Output:



Let us now see how to add images in canvas.

Images:

Image can be displayed in the canvas using create_image() method. But, first we need to load the image using PhotoImage class.

img_file = PhotoImage(file="logo.png")

i1 = c.create_image(200,200,anchor=NE,image=img_file)

The image is displayed relative to the coordinates given (200,200). The image can be displayed at any direction from this point using the option "anchor". The directions are represented by eight directions: NW, N, NE, E, SE, S, SW, W and CENTER. The option "image" represents the image file that is to be displayed on the canvas.

Example 4: Program to display image in canvas

```
from tkinter import*
root = Tk()
root.title("Canvas")
```

```
c = Canvas(root, bg="gray", height=400, width=500, cursor="arrow")

# display image
img_file = PhotoImage(file="logo.png")
i1 = c.create_image(200,200,anchor=NE,image=img_file)

fnt = ("Arial",-22,"bold italic")
t1 = c.create_text(150,50, text="Image in Canvas", font=fnt,fill="lightblue")

c.pack()
root.mainloop()
```

Output:





CHECK YOUR PROGRESS

Q1. What is the purpose of GUI programming?
Q2. What is a canvas?

15.6 FRAMES

A frame is similar to canvas that represents a rectangular area where some text or widgets can be displayed. The root window is also a frame. To create a frame, we create an object of Frame class.

f = Frame(root, bg="gray", height=400, width=500, cursor="arrow")

Here, "f" is the object to the class "Frame", "root" is the name of the parent window, "bg" represents background color, and "height" and "width" represent the height and width of the frame in pixels respectively. The option "cursor" represents the shape of the cursor in the frame.

Once the frame is created it must be added to the root window. This is done by using the pack() method. To add the above frame object "f" we can write the code:

f.pack()

Example 5: Program to display frame

```
from tkinter import*
root = Tk()
root.title("Frame")
f = Frame(root, bg="gray", height=400, width=500, cursor="arrow")
f.pack()
root.mainloop()
```

Output:



After the frame is added, we can now add widgets to the frame.

Widgets:

A widget is a GUI component that is displayed on the screen. Widgets are created as objects. Once a widget is created, it should be added to canvas or frame. Some of the important widgets in Python are:

- Button
- Label

- Message
- Text
- Scrollbar
- Radiobutton
- Checkbutton
- Listbox

Let us look at how we can create and use these widgets.

Button Widget:

The push button is a component that performs some actions when clicked. The button can be created as objects of Button class.

```
b = Button(f, text="Click Me", bg='pink', command=on_button_click)
b.pack()
```

Here, "b" is an object of Button class. "f" represents the frame for which the button is created as a child. The option "text" represents the text to be displayed on the button. The option "bg" represents the background colour for the button. The option "command" provides the link of the push button with the event handler function which is defined separately in the program. In this example, the function "on_button_click()" was defined to print a message on the screen if the button was clicked. The function is given below:

```
def on_button_click():
    print("Button was clicked!")
```

Example 6: Program to display Button widget in frame

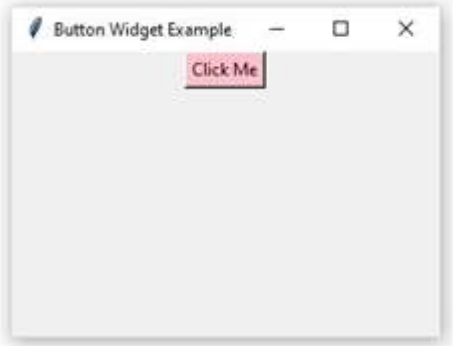
```
from tkinter import *
# Function to be called when button is clicked
def on_button_click():
    print("Button was clicked!")

root = Tk()
root.title("Button Widget Example")
root.geometry("300x200")
```

```
f = Frame(root, bg="lightblue", height=400, width=500)
f.pack()

# Create a Button inside the Frame with a click event
b = Button(f, text="Click Me", bg='pink', command=on_button_click)
b.pack()
root.mainloop()
```

Output:




```
C:\Users\Shruti\Desktop\Python_Programs>python button.py
Button was clicked!
```

Label Widget:

A label represents constant text that is displayed in the frame or container. It can be used to display one or more lines of text that cannot be modified. A label is created as an object of Label class.

```
l = Label(f, text="This is a Label Widget", font=("Arial", 12), fg="blue")
l.pack()
```

Here, "l" represents an object of class Label. "f" is the frame object to which the label "l" is a child. The "text" option represents the data that we want to display in the label of the frame. The "font" option represents the font family, size of the text. The option "fg" represents the foreground colour for the text.

Message Widget:

A message widget is similar to a label. But messages are generally used to display multiple lines of text whereas labels are used to display single line.

To create a message, we need to create an object of Message class.

```
m = Message(f, text="This is a Message Widget.\nThis is the second line
of the Widget.\n", fg="green", width=300)
m.pack()
```

Here, "m" is an object of Message class. The option "text" contains the message to be displayed. The option "width" represents the width of the message in pixels. We can also add options like "font", "fg" and "bg" for specifying the font style, colour of the text and background colour for the message text.

Text Widget:

The text widget is the same as a label or a message widget. But in text widgets, we have several options and can display multiple lines in different fonts and colours. We can also display images in Text widget. We can create a text widget by creating an object to Text class.

```
t = Text(root, height=5, width=40, font=("Arial", 12), fg="red",)
t.pack()
```

Here, "t" represents the object to Text class and "root" represent the object of root window. The "height" and "weight" options represent the height and width of the text in lines. The "font" option specifies that font that is chosen for the text and the option "fg" represents the colour of the text.

Once the text widget is created, we can insert any text using the insert() method.

```
t.insert(END, "This is a Text Widget.\n")
```

Here, the first argument "END" represents that the text will be added at the end of the previous text. We can also use the value "CURRENT" if we want to represent the text at the current cursor position. The second argument is the text that is to be added to the Text widget.

Scrollbar Widget:

A scroll bar widget is used to scroll the text in another widget. For example, if there are multiple lines, we may want to scroll the text in Text, Canvas, Frame or Listbox in either top to bottom or left to right manner using scroll bars. There are two types of scroll bars: horizontal and vertical. The horizontal scrollbar is used to view text from left to right and the vertical scrollbar

is used to view the text from top to bottom. To create a scroll bar, we have to create an object of Scrollbar class.

s = Scrollbar(root, orient=HORIZONTAL, command=t.xview)

Here, "s" represents an object of Scrollbar class which is a child of the root window. The option "orient" represents HORIZONTAL for horizontal scroll bars and VERTICAL for vertical scroll bars. The option "command" represents the method that is to be executed. The method xview is executed on the object "t", where "t" may represent a Text widget or Listbox. After creating the scrollbar, it should be attached to the Text or Listbox widget. This can be done by:

t.configure(xscrollcommand=s.set)

Here, "t" represents the text widget. The argument "xscrollcommand" calls the set() method of horizontal scroll bar. We can also call the vertical scroll bar using the "yscrollcommand". Finally, the scroll bar is attached to the root window using the pack() method.

s.pack(side=BOTTOM, fill=X)

The horizontal scrollbar is attached to the bottom of the widget and it spreads across X-axis.

Let us now look at the program that includes all these different widgets together.

Example 7: Program to display Label, Message, Text and Scrollbar widgets

```
from tkinter import *
root = Tk()
root.title("Widgets")
root.geometry("250x200")
f = Frame(root)
f.pack()

# Label Widget
l = Label(f, text="This is a Label Widget.", font=("Arial", 12), fg="blue")
l.pack()

# Message Widget
```

```
m = Message(f, text="This is a Message Widget.\nThis is the second line
of the Widget.\n", fg="green", width=300)
m.pack()

# Text Widget
t = Text(root, height=5, width=20, font=("Arial", 12), fg="red",)
t.pack()
t.insert(END, "This is a Text Widget.\n")

# Scrollbar Widget
s = Scrollbar(root, orient=HORIZONTAL, command=t.xview)
t.configure(xscrollcommand=s.set)
s.pack(side=BOTTOM, fill=X)

root.mainloop()
```

Output:



Let us now look at few other widgets that can be used in GUI programming.

Checkbutton Widget:

Check buttons or check boxes are used to select one or more options from available list of options. They are displayed in the form of square shaped boxes. When a check button is selected, a tick mark is displayed on the button. We can create check buttons by creating an object of class Checkbutton.

```
cvar = IntVar()
cb = Checkbutton(root, text="I agree to the terms", variable=cvar)
cb.pack()
```

But before we create an object, we need to create a variable object "cvar" of class IntVar(). This value of this variable will represent the information regarding the state of the check button, whether it has been clicked or not. If the value is 1, the check button has been selected; if it is 0 it has not been selected. The check button is then assigned to the root window. The option "text" represents the string that is to be displayed after the check button.

Radiobutton Widget:

A radio button is used to select only one option from a group of options. It is displayed in the form of round shaped button. When a radio button is selected, a dot appears on that particular radio button. We can create a radio button as an object of Radiobutton class.

```
rvar = StringVar(value="Python")
r1 = Radiobutton(root, text="Python", variable=rvar, value="Python")
r2 = Radiobutton(root, text="Java", variable=rvar, value="Java")
r3 = Radiobutton(root, text="C++", variable=rvar, value="C++")
```

But before we create an radio button object, we can create a variable object "rvar" of class StringVar(). This variable is assigned any one of the values from the group of options available for radio buttons. We have created three radio buttons in this example. The object "r1" of radio button class is a child of the root window. The option "text" represents the string tobe displayed after the radio button. The option "variable" represents the object of StringVar class.

```
r1.pack()
r2.pack()
r3.pack()
```

The three radio button objects are then assigned to the root window.

Listbox Widget:

A list box is used to display a list of items so that the user can select one or more of those items. To create a list box, we have to create an object of Listbox class.

lb = Listbox(root, height=4 , selectmode=MULTIPLE)

Here, "listbox" is the Listbox object. The option "height" represents the number of lines shown in the list box. The option "selectmode" represents how many items we want to select. The "selectmode" option can have values: BROWSE, SINGLE, MULTIPLE and EXTENDED. For MULTIPLE mode, we can select one or more items form the list. Once the list box is created, we can insert items into the list box using the insert() method.

```
courses = ["Data Structures", "Python Programming", "DBMS", "Computer Networks"]
for course in courses:
    lb.insert(END, course)
```

Here, first we have created a list named "courses" that contains four values in it. These values are then inserted into the list box one by one using a for loop and insert method.

```
lb.pack()
```

The pack() method is finally called to assign the listbox to the root window.

Example 8: Program to display Checkbutton, Radiobutton and Listbox widgets

```
from tkinter import *
#import tkinter as tk
root = Tk()
root.title("Widgets")
root.geometry("400x300")

# --- Checkbutton Widget ---
cvar = IntVar()
cb = Checkbutton(root, text="I agree to the terms", variable=cvar)
cb.pack()

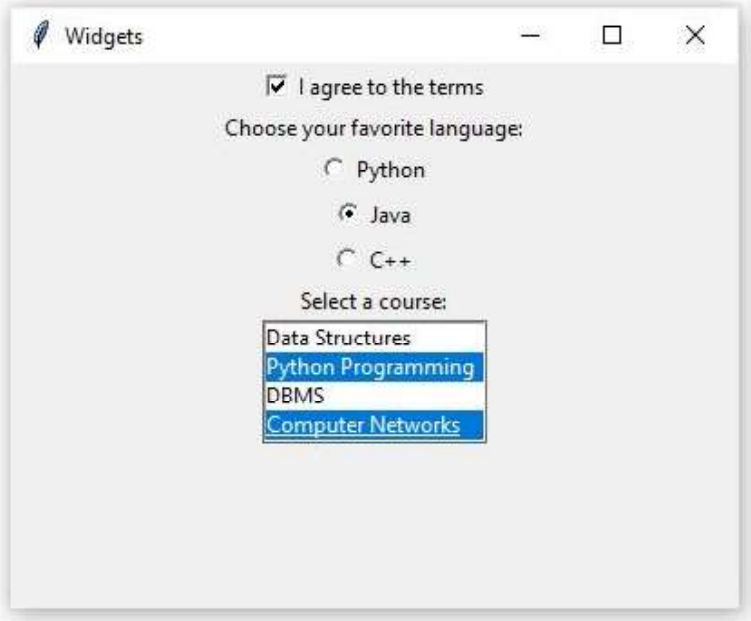
# --- Radiobutton Widgets ---
rvar = StringVar(value="Python")  # Default value
l1 = Label(root, text="Choose your favorite language:")
l1.pack()
```

```
r1 = Radiobutton(root, text="Python", variable=rvar, value="Python")
r2 = Radiobutton(root, text="Java", variable=rvar, value="Java")
r3 = Radiobutton(root, text="C++", variable=rvar, value="C++")
r1.pack()
r2.pack()
r3.pack()

# --- Listbox Widget ---
l2 = Label(root, text="Select a course:")
l2.pack()
lb = Listbox(root, height=4 , selectmode=MULTIPLE)
courses = ["Data Structures", "Python Programming", "DBMS", "Computer
Networks"]
for course in courses:
    lb.insert(END, course)
lb.pack()

root.mainloop()
```

Output:





CHECK YOUR PROGRESS

Q3. What are the two types of scrollbars?

Q4. What is a listbox? How is it created?



15.7 LET US SUM UP

- GUI (Graphical User Interface) programming allows users to interact with applications using graphical elements like buttons, menus, text boxes, etc., rather than typing commands in the console.
- Tkinter stands for "Toolkit Interface", which serves as Python's standard library for creating Graphical User Interfaces (GUIs).
- The first step in developing a GUI program is to create the root window.
- Canvas is used for displaying drwaing like lines, arcs and various other shapes.
- Frame is used for displaying components like push buttons, check buttons, menus etc. These components are called widgets.
- A canvas is a rectangular area which can be used for drawing pictures like lines, circles, rectangles, polygons, arcs etc.
- To create a canvas we need to create an object to Canvas class.
- Image can be displayed in the canvas using create_image() method.
- A frame is similar to canvas that represents a rectangular area where some text or widgets can be displayed.
- To create a frame, we create an object of Frame class.
- A widget is a GUI component that is displayed on the screen.
- Check buttons or check boxes are used to select one or more options from available list of options. They are displayed in the form of square shaped boxes.
- A radio button is used to select only one option from a group of options. It is displayed in the form of round shaped button.
- A list box is used to display a list of items so that the user can select one or more of those items.



15.8 ANSWERS TO CHECK YOUR PROGRESS

Ans to Q1: GUI (Graphical User Interface) programming allows users to interact with applications using graphical elements like buttons, menus, text boxes, etc., rather than typing commands in the console.

Ans to Q2: Canvas is a rectangular area which can be used for drawing pictures like lines, circles, rectangles, polygons, arcs etc. and even adding images.

Ans to Q3: The two types of scroll bars are: horizontal and vertical. The horizontal scrollbar is used to view text from left to right and the vertical scrollbar is used to view the text from top to bottom.

Ans to Q4: A list box is used to display a list of items so that the user can select one or more of those items. To create a list box, we have to create an object of Listbox class.



15.9 FURTHER READING

1) Kanetkar, Y., & Kanetkar, A. (2020). Let Us Python: Python Is Future, Embrace It Fast (Vol. 2). BPB Publications.



15.10 MODEL QUESTIONS

Q1. Define GUI. What are the advantages of GUI programming?

Q2. Using an example show how to create a root window.

Q3. Explain why Tkinter library is important in GUI programming using Python.

Q4. What are the different shapes that can be added to a canvas? Can images be also added to a canvas? If yes, how?

Q5. Explain how to add shapes like line and rectangle to a canvas using suitable examples.

Q6. What are some of the common widgets used in GUI programming?

Q7. Explain how to add button, label and radiobutton widgets to a frame using examples

Q8. Explain how to add text, scrollbar and listbox widgets to a frame using examples.

****

UNIT 16: ADVANCED PYTHON

UNIT STRUCTURE

16.1 Learning Objectives

16.2 Introduction

16.3 Multi-threading in Python

16.4 Networking in Python

16.5 Web Programming in Python

16.6 Using Libraries and Frameworks

16.7 Summing Up

16.8 Answers to Check Your Progress

16.9 References

16.10 Further Readings

16.11 Model Questions

16.1 LEARNING OBJECTIVES

By the end of this unit, you will be able to:

- Understand multi-threading and how to implement it using the threading module.
- Understand the basics of networking and create a simple server and client using the socket module.
- Explore web programming using Python's http.server module and frameworks like Flask.
- Use popular Python libraries and frameworks for advanced development.
- Follow best practices for writing clean, efficient, and maintainable Python code.

16.2 INTRODUCTION

Python is a versatile programming language that supports advanced topics such as multi-threading, networking, and web programming. These concepts allow you to build high-performance, scalable, and interactive appli-

cations. Additionally, Python offers a rich ecosystem of libraries and frameworks that simplify complex tasks. This unit will introduce you to advanced Python topics, demonstrate how to use libraries and frameworks, and provide best practices for Python programming and development.

16.3 MULTI-THREADING IN PYTHON

Multi-threading is a technique that allows a program to perform multiple tasks concurrently. It is useful for improving the performance of I/O-bound or CPU-bound tasks.

(a) The threading Module

Python provides the threading module to create and manage threads.

Example 1: Creating a Thread

```
import threading

def print_numbers():
    for i in range(5):
        print(i)

# Create a thread
thread = threading.Thread(target=print_numbers)

# Start the thread
thread.start()

# Wait for the thread to complete
thread.join()
print("Thread execution complete.")
```

Output:

```
0
1
```

```
2
3
4
Thread execution complete.
```

Explanation:

- A thread is created using the threading.Thread class.
- The start() method begins the thread's execution.
- The join() method ensures the main program waits for the thread to finish.

(b) Thread Synchronization

When multiple threads access shared resources, synchronization is required to avoid conflicts. The threading module provides tools like locks for synchronization.

Example 2: Using a Lock

```
import threading

counter = 0
lock = threading.Lock()

def increment():
    global counter
    for _ in range(100000):
        lock.acquire()
        counter += 1
        lock.release()

# Create threads
thread1 = threading.Thread(target=increment)
thread2 = threading.Thread(target=increment)
```

```
# Start threads
thread1.start()
thread2.start()

# Wait for threads to complete
thread1.join()
thread2.join()

print("Final counter value:", counter)
```

Output:

```
Final counter value: 200000
```

Explanation:

- A Lock object ensures that only one thread can modify the counter variable at a time.

16.4 NETWORKING IN PYTHON

Networking involves communication between devices over a network. Python provides the socket module for low-level networking.

The socket Module

The socket module allows you to create network connections.

Example 3: Creating a Simple Server and Client

Server Code:

```
import socket

# Create a socket object
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
# Bind the socket to a port
server_socket.bind(("127.0.0.1", 12345))

# Listen for incoming connections
server_socket.listen(1)
print("Server listening on port 12345...")

# Accept a connection
client_socket, addr = server_socket.accept()
print(f"Connection from {addr} established.")

# Send a message to the client
client_socket.send(b"Hello from the server!")

# Close the connection
client_socket.close()
```

Client Code:

```
import socket

# Create a socket object
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Connect to the server
client_socket.connect(("127.0.0.1", 12345))

# Receive data from the server
data = client_socket.recv(1024)
print("Received:", data.decode())

# Close the connection
client_socket.close()
```

Explanation:

- The server listens for incoming connections on port 12345.
- The client connects to the server and receives a message.

16.5 WEB PROGRAMMING IN PYTHON

Web programming involves creating applications that users can access through a browser. Python supports web development through built-in modules and web frameworks. Two common approaches are:

1. Using the built-in http.server module for simple HTTP servers.
2. Using a web framework like Flask for more advanced applications.

(a) Using the http.server Module

Python's http.server module allows you to create a simple HTTP server without installing extra packages. It's great for learning or serving static content.

Example 4: Basic HTTP Server

```
from http.server import BaseHTTPRequestHandler, HTTPServer

# Step 1: Create a handler class
class SimpleHandler(BaseHTTPRequestHandler):
    def do_GET(self):
        # Send HTTP status code 200 (OK)
        self.send_response(200)
        # Set the header content type
        self.send_header("Content-type", "text/html")
        self.end_headers()
        # Send a simple HTML response
        self.wfile.write(b"<h1>Hello from the server!</h1>")

# Step 2: Create an HTTP server
server = HTTPServer(("127.0.0.1", 8080), SimpleHandler)
```

```
print("Server running on port 8080...")
# Step 3: Run the server indefinitely
server.serve_forever()
```

Explanation:

- BaseHTTPRequestHandler is the base class for handling HTTP requests.
- do_GET() handles GET requests (browser visits).
- HTTPServer(("127.0.0.1", 8080), SimpleHandler) creates the server on localhost port 8080.
- Running this script allows you to visit http://127.0.0.1:8080 in a browser and see the HTML message.

Example 5: Serving Multiple Pages

```
from http.server import BaseHTTPRequestHandler, HTTPServer

class MultiPageHandler(BaseHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.send_header("Content-type", "text/html")
        self.end_headers()

        if self.path == "/":
            self.wfile.write(b"<h1>Home Page</h1>")
        elif self.path == "/about":
            self.wfile.write(b"<h1>About Page</h1>")
        else:
            self.wfile.write(b"<h1>404 Page Not Found</h1>")

server = HTTPServer(("127.0.0.1", 8080), MultiPageHandler)
print("Server running on port 8080...")
server.serve_forever()
```

Explanation:

- self.path contains the requested URL path.
- You can serve different content for /, /about, or any other route.

This shows the basic idea of routing, which is automated in frameworks like Flask.

(b) Introduction to Web Frameworks: Flask

Flask is a lightweight and beginner-friendly Python framework for building web apps. It simplifies routing, templates, and more.

Example 6: Basic Flask App

```
from flask import Flask

# Create a Flask app
app = Flask(__name__)

# Define route for the home page
@app.route("/")
def home():
    return "<h1>Welcome to Flask!</h1>"

# Run the Flask development server
if __name__ == "__main__":
    app.run(host="127.0.0.1", port=5000)
```

Explanation:

- @app.route("/") maps the URL / to the home() function.
- app.run() starts the server on 127.0.0.1:5000.
- Visit http://127.0.0.1:5000 in your browser to see the message.

Example 7: Multiple Routes and Dynamic Content

```
from flask import Flask
```

```
app = Flask(__name__)

# Home page
@app.route("/")
def home():
    return "<h1>Home Page</h1>"

# About page
@app.route("/about")
def about():
    return "<h1>About Page</h1>"

# Greeting page with a dynamic name
@app.route("/hello/<name>")
def hello(name):
    return f"<h1>Hello, {name}!</h1>"

if __name__ == "__main__":
    app.run(host="127.0.0.1", port=5000)
```

Explanation:

- Flask supports dynamic URLs using <variable> in routes.
- Visiting http://127.0.0.1:5000/hello/Alice will display Hello, Alice!.
- This shows how Flask handles both static and dynamic pages.

(c) Related Concepts

1. HTTP Methods:

- GET: Retrieve data (used above).
- POST: Send data to the server (forms, APIs).

2. Templates:

Flask can render HTML templates instead of returning strings.

Example:

```
from flask import Flask, render_template
app = Flask(__name__)
```

```
@app.route("/")
def home():
    return render_template("home.html")  # HTML file in 'templates' folder
```

- **Static Files:**

Serve CSS, JavaScript, or images using the static folder in Flask.

- **APIs:**

Flask is often used to build APIs by returning JSON:

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route("/data")
def data():
    return jsonify({"name": "Alice", "age": 25})
```

16.6 USING LIBRARIES AND FRAMEWORKS

When we write Python programs, we don't always have to start from scratch. There are libraries and frameworks created by other developers that we can use to save time.

- Library ? A collection of functions and tools you can use in your program.
- Framework ? A bigger structure that tells you how to build an application (like rules + tools).

Let's look at some popular ones

(a) NumPy (Numerical Python)

What it is:

NumPy helps us work with numbers, arrays (lists of numbers), and mathematical calculations faster.

Why use it:

Python lists are slow for math. NumPy arrays are much faster and give us lots of built-in math functions.

Example 8:

```
import numpy as np

# Create an array
arr = np.array([1, 2, 3, 4, 5])

# Add 10 to each element
print(arr + 10)   # [11 12 13 14 15]

# Multiply each element by 2
print(arr * 2)    # [ 2  4  6  8 10]

# Find mean (average)
print(np.mean(arr))  # 3.0
```

With just a few lines, NumPy does things that would take loops in normal Python.

(b) Pandas

What it is:

Pandas is used for data manipulation and analysis.

It lets us work with tables of data (like Excel spreadsheets).

Why use it:

Perfect for reading data from CSV/Excel, filtering rows, and analyzing information.

Example 9:

```
import pandas as pd

# Create a small dataset (like a mini Excel table)
data = {
   "Name": ["Alice", "Bob", "Charlie"],
   "Age": [25, 30, 35],
   "City": ["London", "Paris", "New York"]
}
```

```
df = pd.DataFrame(data)

# Show the table
print(df)

# Select a column
print(df["Age"])

# Filter rows (people older than 28)
print(df[df["Age"] > 28])
```

(c) Matplotlib

What it is:

Matplotlib is used for data visualization (drawing graphs and charts).

Why use it:

Helps us see patterns in data instead of just reading numbers.

Example 10:

```
import matplotlib.pyplot as plt

# Some sample data
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]

# Plot a line chart
plt.plot(x, y)
plt.title("Simple Line Graph")
plt.xlabel("X Axis")
plt.ylabel("Y Axis")
plt.show()
```

You'll see a straight line graph showing the relation between x and y.

(d) Introduction to Django

What it is:

Django is a web framework in Python.

It helps us build websites and web applications quickly.

Why use it:

- Takes care of user authentication (login/signup).
- Connects to databases easily.
- Helps make secure and scalable websites.

Example (basic idea) 11:

With Django, you can write code like this:

```
from django.http import HttpResponse

def hello(request):
    return HttpResponse("Hello, World! This is my first Django page.")
```

When you run the Django server and visit your site, you'll see:

Hello, World! This is my first Django page.

Django is used by big websites like Instagram and Pinterest!



CHECK YOUR PROGRESS

Choose the correct option:

1. Which Python module is primarily used for creating and managing threads?
   a) asyncio
   b) multiprocessing
   c) concurrent
   d) threading
2. What is the purpose of using a Lock object in multi-threading?
   a) To stop threads from completing
   b) To speed up thread execution
   c) To create new threads automatically
   d) To avoid conflicts when threads share resources

3. Which Python module is commonly used for creating low-level network connections?
   a) urllib
   b) networking
   c) socket
   d) requests
4. In web programming with Flask, what does the @app.route("/") decorator do?
   a) Defines a database connection
   b) Returns JSON data by default
   c) Starts the Flask server
   d) Maps a URL path to a function
5. Which of the following pairs is correct regarding Python libraries and their usage?
   a) Django - Numerical computations
   b) Matplotlib - Thread synchronization
   c) Pandas - Data analysis
   d) NumPy - Web development



16.7 SUMMING UP

- Python supports advanced features like multi-threading, networking, and web programming for scalable applications.
- Multi-threading uses the threading module for concurrent tasks, with locks ensuring safe access to shared data.
- Networking with the socket module enables client-server communication and data exchange.
- Web programming can be done using http.server for simple servers or Flask for dynamic web apps.
- NumPy, Pandas, and Matplotlib are popular libraries for numerical computing, data analysis, and visualization.
- Django is a powerful framework for building secure, database-driven, and scalable websites.



16.8 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q.1: d)

Ans. to Q.2: d)

Ans. to Q.3: c)

Ans. to Q.4: d)

Ans. to Q.5: c)

16.9 REFERENCES

- Python Documentation: https://docs.python.org/3/library/threading.html
- Flask Documentation: https://flask.palletsprojects.com/en/stable/
- Django: https://www.djangoproject.com/



16.10 FURTHER READINGS

- Kanetkar, Y., & Kanetkar, A. (2023). Let us Python. BPB Publications.
- https://www.w3schools.com/python/



16.11 MODEL QUESTIONS

1. Explain the concept of multi-threading in Python with an example. How does the threading module help in implementing it?
2. What is thread synchronization? Describe how a Lock object is used to prevent conflicts when multiple threads access shared resources.
3. Differentiate between the start() and join() methods in Python threading with suitable examples.
4. Describe how Python's socket module is used to create a simple client-server application. Illustrate with code.
5. What are the main differences between using Python's built-in http.server module and a framework like Flask for web programming?
6. Write a short note on how Flask handles routing. Explain with an example of static and dynamic routes.

7. Explain the importance of HTTP methods (GET and POST) in web programming. How are they used in Flask applications?
8. Discuss the role of NumPy, Pandas, and Matplotlib in Python programming. Provide one practical example for each.
9. What is Django? Explain why it is considered a powerful framework for building secure and scalable web applications.
10. Define libraries and frameworks in Python. How are they different? Give suitable examples to support your answer.

****