

CSBG004T

FUNDAMENTALS OF PYTHON



DR. C.V. RAMAN UNIVERSITY

Kargi Road, Kota, Bilaspur, (C. G.),

Ph. 07753-253801, 07753-253872

E-mail: info@cvru.ac.in, Website: www.cvru.ac.in

CSBG004T
FUNDAMENTALS OF PYTHON

Credit- 2

Subject Expert Team

Course Edited By:

Unit Written By:

Warning: All rights reserved, No part of this publication may be reproduced or transmitted or utilized or stored in any form or by any means now known or hereinafter invented, electronic, digital or mechanical, including photocopying, scanning, recording or by any information storage or retrieval system, without prior written permission from the publisher.

Edition: 2026

Publish by: DR. C.V. RAMAN UNIVERSITY

Kargi Road, Kota, Bilaspur, (C. G.),

Ph. 07753-253801, 07753-253872

STRUCTURE OF COURSE

Course Title	:	Fundamentals of Python
Course Type	:	Generic Elective Course (Sub III)
Course Code	:	CSBG004T
Course Credits	:	04
Semester	:	I

COURSE LEARNING OBJECTIVES:

By the end of this course, students will be able to:

- Identify the basic data types in Python (e.g., integers, floats, strings, booleans).
- Explain how variables, loops, and conditional statements work in Python.
- Describe the purpose of Python's core built-in functions and how to use them (e.g., print(), len(), range()).
- Use Python operators, control flow, and functions to solve simple computational problems.
- Design Python programs that combine multiple concepts, such as loops, functions, and conditionals, to solve more complex problems.

COURSE LEARNING OUTCOMES:

After completion of the course, the student would be able to

- CO-1: Elaborate the python installation, variables and operators.
- CO-2: Explain the use of data types in python programs.
- CO-3: Apply loops, decision statements, and functions in python.
- CO-4: Determine the class concepts, inheritance and modules of python.
- CO-5: Analyze packages, string and exception handling in python.

Course Content

BLOCK I	INTRODUCTION TO PYTHON PROGRAMMING	1-36
Unit 1	Basics of Python	03
Unit 2	Working with Python Code	13
Unit 3	Operators in Python	27
BLOCK II	BUILT-IN DATA TYPES AND THEIR METHODS	37-66
Unit 4	Numbers, Booleans, and Type Conversion	39
Unit 5	Strings and Their Operations	47
Unit 6	Collections – Lists, Tuples, Dictionaries, and Sets	55
BLOCK III	DECISION MAKING, LOOPS AND FUNCTIONS	65-102
Unit 7	Decision Making	67
Unit 8	Loops & Iteration	75
Unit 9	Functions & Lambda, Arrays	87
BLOCK IV	CLASSES CONCEPTS	103-150
Unit 10	Object-Oriented Programming (OOP) Basics	105
Unit 11	Scope and Modules	121
Unit 12	Working with Data	133
BLOCK V	FILE AND EXCEPTION HANDLING	151-196
Unit 13	File Handling in Python	153
Unit 14	Exception Handling in Python	169
Unit 15	String Formatting & Modular Programming	179

BLOCK I	INTRODUCTION TO PYTHON PROGRAMMING	1-36
Unit 1	Basics of Python	03
Unit 2	Working with Python Code	13
Unit 3	Operators in Python	27



This block introduces students to NumPy, one of the most important Python libraries used for numerical computing, scientific calculations, and data analysis. Students will learn the basics of NumPy arrays, understand how NumPy stores and processes large datasets efficiently, and explore various functions that make numerical operations faster and easier compared to regular Python lists. The block also covers essential techniques for manipulating data, reshaping arrays, slicing, indexing, and performing vectorized operations. By the end of this block, learners will have a strong foundation in NumPy and will be ready to apply it to real-world analytical and computational tasks.

UNIT 1

BASICS OF PYTHON

NOTE

UNIT STRUCTURE

- 1.1. Introduction
- 1.2. Objective
- 1.3. Why Python?
- 1.4. Python Installation
- 1.5. Python Syntax
- 1.6. Summary
- 1.7. Glossary
- 1.8. Answers to check your progress
- 1.9. Suggested readings - OER
- 1.10. Reference
- 1.11. Exercise

1.1 INTRODUCTION

Introduction to “Python Programming”

Python is one of the most popular programming languages because of its ease of use, readability, and versatility. Its user-friendly syntax enables individuals, regardless of experience level, to grasp and apply programming concepts with ease. Python is a flexible, high-level, general-purpose language that is used in many different domains, such as web development, data analysis, automation, artificial intelligence, and more. A key factor in its popularity is the vast range of libraries and community support it offers, which allow developers to perform complex tasks with minimal code.

The installation method for Python is simple and works with the majority of major operating systems, including Windows, macOS, and Linux.

Once installed, it is important to understand Python’s fundamental syntax and conventions, particularly the role of proper indentation and commenting. Python’s emphasis on code readability means that correct indentation is crucial for the code to function correctly. Comments play an equally vital role, enabling developers to document and clarify their code for better collaboration and future reference.

Variables form the foundation of Python programs, capable of holding various data types. Adhering to proper naming conventions ensures that the code remains clean and understandable. Additionally, variables can be declared as global, making them accessible throughout the application.

FUNDAMENTALS OF PYTHON

NOTE

Lastly, Python has a large collection of operators that are essential for carrying out logical operations and calculations. Arithmetic, assignment, comparison, logical, identity, membership, and bitwise operators are among them; they are all necessary for data manipulation and for facilitating program decision-making.

1.2 OBJECTIVE

After completing this unit, you will be capable of:

- Discuss the reasons behind Python's popularity as a programming language and explore its wide range of applications.
- Show how to install Python and execute Python programs.
- Comprehend Python's syntax and techniques for writing and formatting code correctly.

1.3 WHY “PYTHON”?

Python is one of the most widely used programming languages in a wide range of sectors for a number of important reasons.

- 1. Ease of Learning and Use:** Python is a great option for novices because to its straightforward and understandable syntax. Because of its design philosophy, which prioritizes code readability over languages like C++ or Java, programmers can express ideas in fewer lines of code.
- 2. Versatility:** Python is a general-purpose language used for web development, automation, machine learning, data analysis, and game development. Its flexibility allows developers to work across different fields using the same language.
- 3. Rich Ecosystem:** Python offers a vast collection of libraries and frameworks that simplify complex tasks. NumPy, Pandas, and Matplotlib support data analysis, while TensorFlow and PyTorch are key in machine learning. For web development, Django and Flask provide efficient solutions, and automation is made easier with tools like Selenium.
- 4. Strong Community Support:** The Python community is vibrant and helpful, providing a multitude of forums, tutorials, and tools. It is simpler for developers to learn, troubleshoot, and advance their abilities with this community-driven assistance system.
- 5. Cross-Platform Compatibility:** Python is cross-platform, which means that programs created on one OS will usually function perfectly on another. Its versatility enhances accessibility across a variety of devices and platforms.

1.4 PYTHON INSTALLATION

NOTE

Detailed instructions for setting up Python on Windows 10 or 11:

1. Get Python by visiting python.org, the official Python website.

- Select the latest stable version: The downloads page may contain the latest stable version of Python, such as Python 3.x. Click the button to get it for Windows.

2. Start the Installer

- Locate the downloaded file here: When the download is complete, find the installer in your "Downloads" folder; it will look like python-3.x.x.exe.
- First, install the installer: Double-clicking the installer will start the setup process.

3. Look at your options for installation.

- To use the "command line" to run "Python", tick the box labeled "Add Python to PATH." This is a crucial stage.
- Additional control over the installation can be obtained by selecting "Customize Installation." If not, you can select "Install Now."

4. Install Python

- The installation of Python will start, along with optional features like IDLE (Python's integrated development environment) and pip (Python's package installer).
- Hold off until the installation is finished. Usually, it only takes a few minutes.

5. Examine the setup

- Start the "Command Prompt" by: Press "Win + R", type 'cmd', then press 'Enter'.
- Check the version of Python: In the "Command Prompt", type "python -version" and press "Enter". By showing the "installed version" of Python, this ought to verify the installation.

6. Check pip version:

- To confirm that Python's package management, pip, was also installed, type pip --version.
- Learn Python Now, you can start Python programs from the Command Prompt or utilize IDLE, which is included with the installation. To initiate IDLE, search for and select "IDLE" from the Start menu.

NOTE

Your Windows computer has successfully installed Python!

This unit serves as a foundation for understanding the essential concepts and techniques in programming, providing both theoretical knowledge and practical insights into effective programming practices.

1.5 SYNTAX

Python Syntax

The rules that specify how Python programs should be written and how the Python interpreter should interpret them are known as Python syntax. It outlines the proper arrangement of operators, variables, keywords, and symbols to create legitimate instructions.

Fundamental Syntax Guidelines:

- 1. Case Sensitivity:** My variable and my variable are regarded as separate variables since Python can tell the difference between capital and lowercase characters.
- 2. Indentation:** Proper indentation, achieved using spaces or tabs at the start of lines, is essential in Python. It defines code blocks like function or loop bodies, with the standard being 4 spaces per level.
- 3. Statements:** Semicolons are not needed in Python to terminate statements; normally, a single statement is represented by each line.
- 4. Comments:** Multi-line comments are encapsulated in triple quotes, either `"""` or `"""`, whereas single-line comments start with `#`.
- 5. Variables and Assignments:** Variables in Python are dynamically typed and assigned values using the `=` operator.
- 6. Function Definition:** The `def` keyword is used to create functions. The function name and parameters are then placed in parenthesis.

Example:

```
# This is a comment
x = 10 # Assigning a value to a variable
y = 20

# Function definition with indentation
def add_numbers(a, b):
    sum = a + b
    return sum

# Calling the function
result = add_numbers(x, y)
print(result) # Output: 30
```

NOTE**In this example:**

- The numbers 10 and 20 are allocated to the variables 'x' and 'y'.
- Two numbers are added and their sum is returned by the defined function add_numbers.
- Indentation is used to define the scope of the function body.
- Comments help explain the code.
- The 'print function' outputs the result to the console.

**Activity 1.1**

1. Write down three key reasons why Python is a good choice for beginners and professionals.
2. Discuss these reasons with your station partner and come up with one unique use case where Python would be particularly effective (e.g., data analysis, web development, automation)

Materials Needed:

- Laptops or computers with Python installed (or internet access for installation steps)
- Whiteboard/Blackboard and markers
- Handouts for each station (containing instructions, examples, and tasks)

.....

.....

.....

.....

☑ Check Your Progress 1.1

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

1. Why is Python considered a beginner-friendly programming language?

.....

.....

.....

.....

2. What are the basic steps to install Python on a Windows system?

.....

.....

.....

.....

NOTE

1.6. SUMMARY

Python is a powerful and user-friendly programming language with a simple and readable **syntax**, making it ideal for beginners and professionals. Before coding, Python must be **installed**, allowing access to its interpreter and libraries. Python's syntax is straightforward, requiring minimal punctuation and indentation to define code blocks. Its versatility and ease of learning contribute to its widespread use across various fields, from web development to artificial intelligence.

1.7 GLOSSARY

- **Python** – A high-level, general-purpose programming language known for its readability.
- **Python Installation** – The process of downloading and setting up Python on a computer.
- **Interpreter** – A program that executes Python code line by line.
- **Syntax** – The set of rules that define how Python code is written and structured.
- **Indentation** – The use of whitespace to define code blocks in Python.
- **Readability** – The ease of understanding Python code due to its simple structure.
- **Versatility** – The ability of Python to be used in various fields like AI, web development, and automation.

1.8 ANSWERS TO CHECK YOUR PROGRESS

1. Why is Python considered a beginner-friendly programming language?

Answer: Python is considered beginner-friendly because of its **simple and readable syntax**, which closely resembles the English language. Unlike other programming languages that require complex syntax and extensive punctuation, Python uses **indentation to define code blocks**, making it easier to understand and write. Additionally, Python has a large **standard library** and an active community, providing ample resources for beginners to learn and develop applications efficiently.

2. What are the basic steps to install Python on a Windows system?

Answer:

To install Python on a Windows system, follow these steps:

NOTE

1. **Download Python:** Visit the official Python website (<https://www.python.org>) and download the latest stable version.
2. **Run the Installer:** Open the downloaded file and ensure you check the option "**Add Python to PATH**" before proceeding.
3. **Complete the Installation:** Click on "Install Now" and wait for the process to finish.
4. **Verify Installation:** Open the **Command Prompt** and type `python --version` or `python` to confirm Python is installed and running correctly.

1.8 SUGGESTED READINGS

Sr. No.	Title	URL	QR Code
1	Open Educational Resource	http://books.aisectionline.com/UnitUrls?unitId=1276	

1.9 REFERENCE

- Lutz, Mark. *Learning Python*. O'Reilly Media, 2013. A comprehensive guide to Python covering syntax, variables, operators, and core programming concepts.
- Zelle, John M. *Python Programming: An Introduction to Computer Science*. Franklin, Beedle & Associates Inc., 2010. Focuses on the basics of Python, including syntax, comments, and variable management.

1.10 EXERCISE**Part 1****A. Multiple Choice Questions (MCQs)**

1. **Which of the following is a key reason why Python is widely used?**
 - a) Complex syntax
 - b) High execution speed only
 - c) Readable and simple syntax
 - d) Requires a compiler to execute
2. **Where can you download Python for installation?**
 - a) www.python.org

NOTE

- b) www.installpython.com
- c) www.downloadpython.net
- d) www.pythoninstaller.com

3. Which command is used to check if Python is installed on your system?

- a) check python
- b) python -verify
- c) python -version
- d) verify python

4. Which of the following is true about Python syntax?

- a) Uses curly brackets {} to define code blocks
- b) Uses indentation instead of brackets
- c) Requires semicolons at the end of each statement
- d) Does not support functions

5. Which of the following statements about Python is false?

- a) Python is an interpreted language
- b) Python syntax is case-sensitive
- c) Python does not support object-oriented programming
- d) Python supports multiple programming paradigms

Answers with Explanations:

- 1. c) Readable and simple syntax** - Python is popular because its clean and readable syntax makes coding easier. This simplicity helps beginners learn programming quickly.
- 2. a) www.python.org** - Python's official and safest download source is its website python.org. Other links are not official and may be unsafe.
- 3. c) python -version** - The command `python --version` displays the installed Python version. It is the standard way to check if Python is installed.
- 4. b) Uses indentation instead of brackets** - Python uses indentation to define code blocks instead of curly braces. This keeps the code clean and readable.
- 5. c) Python does not support object-oriented programming** - This statement is false because Python fully supports object-oriented

NOTE

programming. It also supports procedural and functional paradigms.

B. Small answer type question

1. What makes Python different from other programming languages?
2. Explain the steps to install Python on a macOS system.
3. What is the role of indentation in Python syntax?

Part 2**C. Long answer type question (200 words)**

1. Discuss the key reasons why Python is considered one of the most popular programming languages. Explain its advantages and common applications.
2. Describe the step-by-step process of installing Python on different operating systems (Windows, macOS, and Linux). Also, explain how to set up Python for programming after installation.

Part 3**D. Long answer type question (300 words)**

1. Explain the key features of Python that make it one of the most preferred programming languages for beginners and professionals. Discuss its simplicity, interpreted nature, dynamic typing, and extensive libraries with suitable examples.
2. Describe the structure of a basic Python program. Explain concepts such as comments, indentation, variables, keywords, and input/output functions. Discuss why Python stresses readability and how it impacts program development.
3. Discuss the steps involved in installing Python, setting up the environment, and writing the first Python script. Explain how interactive mode and script mode work and compare their use cases.

UNIT 2

WORKING WITH PYTHON CODE

NOTE

UNIT STRUCTURE

- 2.1 Introduction
- 2.2 Objective
- 2.3 Indentation & Comments: Importance and usage
- 2.4 Variables in Python: Names, Outputs, and Global variables
- 2.5 Summary
- 2.6 Glossary
- 2.7 Answers to Check Your Progress.
- 2.8 Suggested readings - OER
- 2.9 Reference
- 2.10 Exercise

2.1 INTRODUCTION

Python is a widely used programming language known for its readability and simplicity. Two fundamental aspects of writing clean and efficient Python code are indentation and comments, which help structure the code and enhance its readability. Indentation in Python is mandatory, as it defines code blocks, ensuring proper execution, while comments provide explanations, making the code easier to understand and maintain. Another crucial concept is the use of variables, which store and manage data. Understanding variable naming conventions, displaying outputs, and distinguishing between local and global variables allows for better program organization and functionality. Mastering these concepts is essential for writing effective and maintainable Python programs.

2.2 OBJECTIVE

After completing this unit, you will be capable of:

- To understand the importance of indentation and comments in Python, emphasizing how indentation structures code blocks and how comments enhance readability and maintainability.
- To explore variables in Python, including proper naming conventions, displaying outputs, and differentiating between local and global variables for better code organization.
- To enable writing efficient and structured Python programs by applying best practices in indentation, commenting, and variable management.

NOTE**2.3 INDENTATION & COMMENTS: IMPORTANCE AND USAGE****INDENTATION**

The brackets {} that are frequently used in languages like C or Java are replaced with indentation to establish the structure and flow of the code.

It involves adding spaces or tabs at the start of a line of code and serves as a fundamental feature of the language. Code blocks in loops, functions, classes, and conditional expressions are indicated by indentation. Every line in the same block needs to have the same amount of indentation.

Key Points:

1. **Consistency:** Indentation must be consistent throughout a block of code. Typically, 4 spaces are used, though tabs can also be used. However, you should not mix spaces and tabs.
2. **Syntax Error:** An IndentationError or a SyntaxError will be raised for improper or inconsistent indentation. This is due to the fact that Python uses indentation to comprehend the program's hierarchy and flow.
3. **Scope and Blocks:** Indentation determines the scope of code blocks, such as defining the body of functions, loops, conditional statements, and classes.

Example:

```
# Correct indentation example
def greet(name):
    if name:
        print(f"Hello, {name}!") # Indented code inside the if statement
    else:
        print("Hello, Stranger!") # Indented code inside the else statement

greet("Alice")
```

In this instance:

- The code of greet function is appropriately indented beneath it.
- Both the if and else statements contain indented code blocks that run depending on the condition. The indentation within these blocks must remain consistent.

Incorrect Indentation Example:

```
def greet(name):
    if name:
        print(f"Hello, {name}!")
    else:
        print("Hello, Stranger!") # this line will raise an IndentationError
```

Python will give a "Indentation Error" because it expects the code under "else" to be indented to match the block's level, but the print ("Hello, Stranger!") statement in the wrong example is not.

NOTE

COMMENTS

They are used in Python to write notes, 'explain code', and temporarily disable particular areas. They are only used to help developers understand the code; the Python interpreter ignores them. Especially in large projects, comments are crucial to writing code that is readable and manageable.

Types of 'Comments':

1. **"Single-line Comments:"** Created using the '#' symbol. Everything following # on the same line is ignored by the interpreter.
2. **"Multi-line Comments:"** Python does not have a dedicated syntax for "multi-line comments" like some other languages. However, triple quotes (''' or ''') can be used to simulate them. These are commonly used as docstrings to document modules, functions, classes, and methods, but they can also act as multi-line comments when not assigned to any variable or object.

Example Usage:

- # Single-line comment explaining the code below: Indicates that the next line assigns a value to x.
- # The number 5 is allocated to variable x: explains the function of the variable x.
- Inside the greet function, a multi-line comment or docstring explains the function's purpose, parameters, and expected input.



Activity 2.1

1. List three reasons why Python is a popular programming language.
2. In small groups, discuss how Python's simplicity helps beginners and identify one real-world use case where Python would be highly effective (e.g., web development, data analysis, automation).
3. Follow the Python installation guide and verify that Python is installed on your system.

Materials Needed:

- Laptops or computers with Python installed (or access to install it).
- Python official website (www.python.org) for installation steps.
- Handouts with installation instructions and syntax examples.

NOTE

- Whiteboard/blackboard and markers for brainstorming.

.....

.....

.....

.....

2.4 VARIABLES IN PYTHON: NAMES, OUTPUTS, AND GLOBAL VARIABLES

What is Variables?

In Python, a variable serves as a reference to a value stored in memory. It allows data to be stored and manipulated throughout a program's execution. Unlike some other languages, Python does not require declaring a variable's type beforehand. Instead, it utilizes dynamic typing, meaning the type is automatically assigned at runtime based on the value provided.

Key Features of Python Variables:

1. **Dynamic Typing** – Python variables are not bound to a fixed data type. For instance, a variable can store an integer initially and later hold a string.
2. **Assignment with =** – The = operator is used to assign values to variables.
3. **No Pre-Declaration Required** – Variables can be defined simply by assigning a value without prior declaration.
4. **Automatic Type Recognition** – Python determines the variable's type dynamically based on the assigned value.

Example:

```
# Variable assignment
age = 30          # 'age' is assigned an integer value
name = "Alice"    # 'name' is assigned a string value
price = 19.99     # 'price' is assigned a float value
is_member = True  # 'is_member' is assigned a boolean value

# Changing variable types
age = "thirty"   # Now 'age' is a string, not an integer
```

In above example:

- Values of various types (integer, string, float, and boolean) are produced and assigned to the variables age, name, price, and is member.
- Python recognizes each variable's type automatically depending on the value that has been set to it.

- Variables can always be reassigned to different kinds. Age, for instance, was initially assigned as an integer before being changed to a string.

NOTE**Rules for Naming Variables:**

- **Alphanumeric and Underscores:** Only letters (a-z, A-Z), digits (0-9), and underscores (_) may be included in variable names. My_variable, for instance, is legitimate.
- **No Special Characters:** Variable names cannot contain spaces, @, \$, or %.
- **Cannot Begin with a Number:** Instead of beginning with a number, a variable name must begin with a letter or an underscore. Var1 is legitimate, while 1var is not, for instance.
- **Case Sensitivity:** The case of variable names matters. For instance, there are three distinct variables: myVariable, MyVariable, and myvariable.
- **Keywords:** You are not allowed to use any of Python's reserved keywords as variable names, including if, else, for, while, True, False, class, etc.

Naming Conventions:

- **Snake Case:** By convention, variable names are usually written in snake_case, meaning words are separated by underscores (_). For example, my_variable.
- **Camel Case:** Variable names occasionally employ camelCase, in which the first word is lowercase and the ones that follow are uppercase. Take myVariable, for instance.
- **Descriptive Names:** Using descriptive names, like total_amount or user_name, that express the purpose of the variable is a good idea.
- **Example:**

```
# Valid variable names
age = 25
first_name = "John"
last_name = "Doe"
totalAmount = 100.50

# Invalid variable names
1st_value = 10 # Starts with a number, this will raise a syntax error
my-name = "Alice" # Contains a special character (hyphen), this will raise a syntax error

# Case sensitivity
user = "Alice"
User = "Bob" # 'user' and 'User' are two distinct variables
```

NOTE**In above example:**

- The “variables” totalAmount, age, first_name, and last_name are legitimate and have the proper names.
- 1st_value and my-name are invalid due to naming rule violations.
- The case-sensitive nature of Python is demonstrated with the variables user and User, which are treated as two different variables.

Python Variable Value Assignment

In Python, variables are assigned values using the = operator. This operator inserts the value on the right into the variable on the left. Below is a description of how to set values for variables in Python:

Basic Assignment:

- 1. Single Value Assignment:** giving a variable a single value.

```
x = 10          # Assigns the integer 10 to variable x
name = "Alice" # Assigns the string "Alice" to variable name
```

- 2. Reassigning Values:** By giving a variable a new value, you can alter its value.

```
x = 10
x = 20 # Now x holds the value 20
```

Assign Multiple Values in variable

In a single line of code, many Python variables can have multiple values set to them. A useful technique for initializing multiple variables at once is numerous assignment. Python enables you to utilize a single value for several variables at once or assign values to variables one-to-one.

Multiple Assignment Rules:

- 1. One-to-One Assignment:** By using commas to separate the variables and their respective values, you can assign multiple values to multiple variables.
- 2. Same Value Assignment:** You can also assign a single value to several variables by using the assignment operator =.
- 3. Unpacking:** Python permits unpacking sequences (such as lists or tuples) into variables, provided that the number of variables is equal to the number of elements in the sequence.

Example 1: Assigning Multiple Values

```
# Assigning multiple values to multiple variables
a, b, c = 1, 2, 3
print(a) # Output: 1
print(b) # Output: 2
print(c) # Output: 3
```

In this example, the variables “a, b, and c” are each given the values 1, 2, and 3, in a single line.

Example 2: Assigning the Same Value to Multiple Variables

```
# Assigning the same value to multiple variables
x = y = z = 10
print(x) # Output: 10
print(y) # Output: 10
print(z) # Output: 10
```

Here, the value 10 is assigned to all three variables x, y, and z at once.

Example 3: Unpacking a Sequence

```
# Unpacking a list into variables
numbers = [4, 5, 6]
p, q, r = numbers
print(p) # Output: 4
print(q) # Output: 5
print(r) # Output: 6
```

The values 4, 5, and 6 are allocated to p, q, and r, respectively, once the list numbers are unpacked in this example. The number of variables on the left and the number of elements in the sequence on the right must match.

Output Variables

In Python, "outputting variables" refers to the process of publishing or displaying the values of variables to the console or other output streams. This is commonly accomplished by using the print() method, which outputs the output to the standard output (typically the console). Although there are alternative methods, such as formatting strings and writing to files, the “print() function” is the most straightforward and popular way to output variables.

Using the print() Function:**1. Basic Printing:**

```
name = "Alice"
age = 30
print(name) # Output: Alice
print(age) # Output: 30
```

NOTE

NOTE**2. Concatenation:**

```
first_name = "Alice"
last_name = "Smith"
print(first_name + " " + last_name) # Output: Alice Smith
```

3. Formatted Strings “(f-strings)”: (Python 3.6+)

```
name = "Alice"
age = 30
print(f"Name: {name}, Age: {age}") # Output: Name: Alice, Age: 30
```

4. Using the “format() Method:”

```
name = "Alice"
age = 30
print("Name: {}, Age: {}".format(name, age)) # Output: Name: Alice, Age: 30
```

5. Printing Multiple Variables:

```
x = 10
y = 20
z = 30
print(x, y, z) # Output: 10 20 30
```

6. Printing with Separator and End Parameters:

Separator: Specify a string to be inserted between variables.
python

```
x = 1
y = 2
z = 3
print(x, y, z, sep=", ") # Output: 1, 2, 3
```

End: Give the output a string to be appended at the end (a newline is the default).

“Global” and “Local Variables” in Python

In Python, “global” and “local variables” help manage data within various parts of a program, depending on their scope. Understanding these variables is essential for effectively handling data accessibility and modification in functions and other parts of the code.

“Global Variables:”

Definition: A global variable can be accessed from anywhere in the program, including within functions and classes, and is declared outside of all of them.

Scope: Once defined, these variables can be accessed from anywhere in the code due to their global scope.

Usage: Values that must be shared across several functions, including configuration information or constants, are frequently stored in global variables.

Example of Global Variables:

```
# Global variable
global_var = "I am global"

def print_global():
    print(global_var) # Accessing global variable

print_global() # Output: I am global
```

The variable `global_var` is available within the `print_global()` function in this example, while being declared outside the function.

Local Variables:

Definition: A local variable is one that is defined within a function or block and can only be accessed from within that specific context. It is created when the function executes and is removed once the function completes.

Scope: Local variables are confined to the function or block in which they are declared.

Usage: They are primarily used to store temporary data required only while the function or block is running.

Example of Local Variables:

```
def my_function():
    # Local variable
    local_var = "I am local"
    print(local_var) # Accessing local variable

my_function() # Output: I am local

# Trying to access local_var outside the function will raise an error
# print(local_var) # Uncommenting this line will cause a NameError
```

`local_var` is only accessible within `my_function()`, in this instance, where it is declared. If you try to access it outside of the function, an error will appear.

Modifying Global Variables Within Functions:

To modify a global variable within a function, the `global` keyword is required. This ensures the function accesses the existing global variable rather than creating a new local one.

NOTE

NOTE**Example of Modifying Global Variables:**

```
global_var = "Initial Value"

def modify_global():
    global global_var # Declare that we want to use the global variable
    global_var = "Modified Value"

modify_global()
print(global_var) # Output: Modified Value
```

In this example, `global_var` is modified inside `modify_global()` by using the `global` keyword. The change is reflected globally, so when `global_var` is printed after the function call, it shows the modified value.

☑ Check Your Progress 2.1

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

1. Why is indentation important in Python?

.....

2. What is a global variable in Python? How does it differ from a local variable?

.....

2.5 SUMMARY

Indentation and comments play a crucial role in Python programming. Unlike languages like C or Java that use brackets {}, Python relies on indentation to structure code blocks in loops, functions, and conditionals, ensuring consistency and readability. Improper indentation leads to syntax errors, affecting the program's execution. Comments, on the other hand, help developers understand the code by providing explanations. Python supports both single-line comments using # and multi-line comments using triple quotes (""" or """). Variables in Python store and manage data dynamically without explicit type declarations. They follow naming rules such as using alphanumeric characters and underscores while avoiding keywords and special characters. Python also supports multiple assignments and variable unpacking. Outputting variable values is commonly done using the `print()` function, with options like concatenation, formatted strings, and separators. Understanding the scope of variables is essential—local variables exist within functions, whereas global

variables can be accessed throughout the program. The global keyword allows modifying global variables within functions. Mastering these fundamental concepts ensures efficient, structured, and readable Python code.

NOTE

2.6 GLOSSARY

- **Indentation** – The use of spaces or tabs to structure Python code blocks, replacing {} used in other languages.
- **Syntax Error** – An error that occurs when Python code is incorrectly formatted, often due to improper indentation.
- **Comments** – Non-executable text in code used to explain logic, ignored by the Python interpreter.
- **Single-line Comment** – A comment written with #, affecting only the text following it on the same line.
- **Multi-line Comment** – A comment enclosed within triple quotes (''' or '''), often used as documentation.
- **Variable** – A named reference that stores data in memory and can be reassigned different values.

2.7 ANSWERS TO CHECK YOUR PROGRESS

1. Why is indentation important in Python?

Answer: Indentation in Python is used to define code blocks instead of using curly braces {} as in other languages. It improves code readability and ensures proper execution of control structures like loops and functions. Improper indentation will cause an Indentation Error, making it a crucial aspect of Python syntax.

2. What is a global variable in Python? How does it differ from a local variable?

Answer: A **global variable** is a variable declared outside of functions and accessible throughout the entire program. A **local variable**, on the other hand, is defined inside a function and can only be accessed within that function. Global variables can be modified inside a function using the global keyword.

2.8 SUGGESTED READINGS - OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource	http://books.aisectionline.com/User/UnitUrls?unitId=1277	

NOTE

2.9 REFERENCE

- Lutz, M. (2013). *Learning Python*. O'Reilly Media.
- Severance, C. (2016). *Python for Everybody: Exploring Data Using Python 3*. CreateSpace Independent Publishing.
- Van Rossum, G., & Drake, F. L. (2009). *The Python Language Reference Manual*. Network Theory Ltd.

2.10 EXERCISE

Part 1**A. Multiple Choice Questions (MCQs)****1. What is a key advantage of Python?**

- a) Complicated syntax
- b) High readability and simplicity
- c) Requires complex compilation
- d) No support for data science

2. Where can you download Python officially?

- a) www.python.org
- b) www.downloadpython.com
- c) www.python-install.net
- d) www.pythonsetup.com

3. Which of the following is true about Python syntax?

- a) Uses semicolons ; to separate statements
- b) Uses indentation instead of braces {}
- c) Requires explicit declaration of data types
- d) Does not support loops

4. How do you check if Python is installed on your system?

- a) `python -version`
- b) `check python`
- c) `verify python`
- d) `install python`

5. Which of the following is NOT a characteristic of Python?

- a) Case-sensitive syntax
- b) Compiled language
- c) Interpreted execution
- d) Supports multiple programming paradigms

NOTE**Answers with Explanations:**

1. **b) High readability and simplicity** - Python is known for its clean, easy-to-read syntax. This makes it beginner-friendly and faster to develop programs.
2. **a) `www.python.org`** - The official Python downloads are available only on `python.org`. Other sites are unofficial and not recommended.
3. **b) Uses indentation instead of braces `{}`** - Python uses indentation to define code blocks. This improves readability and enforces consistent structure.
4. **a) `python --version`** - Running `python --version` (or `python3 --version`) shows the installed Python version. If Python is not installed, the command will fail.
5. **b) Compiled language** - Python is primarily an interpreted language. It does not require full compilation like C or Java.

B. Short Answer Type Questions

1. Why is Python widely used across different industries?
2. Explain the role of indentation in Python syntax.
3. What are the main steps to install Python on a Windows system?

Part 2**C. Long Answer Type Questions (200 words)**

1. Discuss why Python is a preferred language for beginners and professionals. Explain its advantages and real-world applications.
2. Describe the complete process of installing Python on Windows, macOS, and Linux. Also, explain how to verify the installation and set up an environment for programming.

Part 3**D. Long answer type question (300 words)**

1. Explain the process of writing, saving, and executing Python programs. Discuss different code editors/IDEs, indentation rules, and common syntax errors students face in the beginning.
2. Discuss the importance of input and output functions in Python. Explain how the `input()`, `print()`, and type conversion functions are used to receive data and display results.

INTRODUCTION TO PYTHON PROGRAMMING

UNIT 3

OPERATORS IN PYTHON

NOTE

UNIT STRUCTURE

- 3.1 Introduction
- 3.2 Objective
- 3.3 Operators: “Arithmetic, Assignment, Comparison”
- 3.4 Logical, “Identity, Membership and Bitwise.”
- 3.5 Summary
- 3.6 Glossary
- 3.7 Answers to check your progress
- 3.8 Suggested Readings
- 3.9 Reference
- 3.10 Exercise
- 3.11 Feedback Form

3.1 INTRODUCTION

Operators are fundamental components in programming that perform various operations on variables and values. They allow programmers to manipulate data, compare values, and control the flow of execution. The main categories of operators include arithmetic operators (which handle mathematical calculations), assignment operators (used to assign values to variables), and comparison operators (which compare two values). Additionally, logical operators facilitate decision-making by combining multiple conditions, identity operators determine whether two variables reference the same object, membership operators check for the presence of a value in a sequence, and bitwise operators perform operations at the binary level. Understanding these operators is essential for writing efficient and functional code in any programming language.

3.2 OBJECTIVE

After completing this unit, you will be capable of:

- To understand the functionality and usage of arithmetic, assignment, and comparison operators.
- To explore logical operators and their role in decision-making.
- To analyze identity operators and their application in memory reference comparisons.
- To learn about membership operators and their use in sequence data types.

NOTE

- To examine bitwise operators and their significance in low-level programming and performance optimization.

3.3 OPERATORS: ARITHMETIC, ASSIGNMENT, COMPARISON

An “operator” in Python is a unique symbol or term that is used to manipulate variables or values. Programming relies heavily on operators, which allow you to compare data, assign values, do mathematical computations, and more. They are crucial for creating expressions and managing a program's flow, and they are used to work with data and variables.

Types of Operators in Python

1. **Arithmetic Operators:** These are used for performing mathematical operations.
 - **Addition (+):** Adds two values together.
 - Example: `result = 5 + 3` # result is 8
 - **Subtraction (-):** Subtracts one value from another.
 - Example: `result = 5 - 3` # result is 2
 - **Multiplication (*):** Multiplies two values.
 - Example: `result = 5 * 3` # result is 15
 - **Division (/):** Divides one value by another, always returning a float.
 - Example: `result = 5 / 3` # result is 1.6667
 - **Floor Division (//):** Divides one value by another, returning only the integer part of the quotient.
 - Example: `result = 5 // 3` # result is 1
 - **Modulus (%):** Returns the remainder from the division.
 - Example: `result = 5 % 3` # result is 2
 - **Exponentiation (**):** Raises one value to the power of another.
 - Example: `result = 5 ** 3` # result is 125
2. **Assignment Operators:** These are used to assign values to variables.
 - **Assignment (=):** Assigns a value to a variable.
 - Example: `x = 5`
 - **Addition Assignment (+=):** Adds and assigns the result.
 - Example: `x += 3` # equivalent to `x = x + 3`
 - **Subtraction Assignment (-=):** Subtracts and assigns the result.
 - Example: `x -= 2` # equivalent to `x = x - 2`
 - **Multiplication Assignment (*=):** Multiplies and assigns the result.
 - Example: `x *= 4` # equivalent to `x = x * 4`
 - **Division Assignment (/=):** Divides and assigns the result.
 - Example: `x /= 2` # equivalent to `x = x / 2`
 - **Floor Division Assignment (//=):** Performs floor division and assigns the result.

NOTE

- Example: `x //= 3` # equivalent to `x = x // 3`
 - **Modulus Assignment (%)**: Applies modulus and assigns the result.
 - Example: `x %= 3` # equivalent to `x = x % 3`
 - **Exponentiation Assignment (==)**: Applies exponentiation and assigns the result.
 - Example: `x **= 2` # equivalent to `x = x ** 2`
3. **Comparison Operators**: These are used to compare two values and return a boolean result (True or False).
- **Equal to (==)**: Checks if two values are equal.
 - Example: `result = (5 == 3)` # result is False
 - **Not equal to (!=)**: Checks if two values are not equal.
 - Example: `result = (5 != 3)` # result is True
 - **Greater than (>)**: Checks if one value is greater than another.
 - Example: `result = (5 > 3)` # result is True
 - **Less than (<)**: Checks if one value is less than another.
 - Example: `result = (5 < 3)` # result is False
 - **Greater than or equal to (>=)**: Checks if one value is greater than or equal to another.
 - Example: `result = (5 >= 3)` # result is True
 - **Less than or equal to (<=)**: Checks if one value is less than or equal to another.
 - Example: `result = (5 <= 3)` # result is False

3.4 OPERATORS: LOGICAL, IDENTITY, MEMBERSHIP AND BITWISE.

4. **Logical Operators**: These are used to combine conditional statements.
- **AND (and)**: Returns True if both conditions are true. Example: `result = (5 > 3) and (3 < 4)` # result is True
 - **OR (or)**: Returns True if at least one of the conditions is true. Example: `result = (5 > 3) or (3 > 4)` # result is True
 - **NOT (not)**: Returns True if the condition is false. Example: `result = not (5 > 3)` # result is False
- **Identity Operators**: These compare the memory locations of two objects.
- **Is (is)**: Checks if two variables point to the same object. Example: `result = (5 is 5)` # result is True
 - **Is Not (is not)**: Checks if two variables point to different objects. Example: `result = (5 is not 3)` # result is True
- **Membership Operators**: These test if a value is present in a sequence.
- **In (in)**: Checks if a value exists in a sequence. Example: `result = 3 in [1, 2, 3, 4]` # result is True
 - **Not In (not in)**: Checks if a value does not exist in a sequence. Example: `result = 5 not in [1, 2, 3, 4]` # result is True

NOTE

5. **Bitwise Operators:** These are used to perform operations on the binary representations of integers.

- **AND (&):** Performs a bitwise AND operation. Example: result = 5 & 3 # result is 1
- **OR (|):** Performs a bitwise OR operation. Example: result = 5 | 3 # result is 7
- **XOR (^):** Performs a bitwise XOR operation. Example: result = 5 ^ 3 # result is 6
- **NOT (~):** Performs a bitwise NOT operation. Example: result = ~5 # result is -6
- **Left Shift (<<):** Shifts bits to the left. Example: result = 5 << 1 # result is 10
- **Right Shift (>>):** Shifts bits to the right. Example: result = 5 >> 1 # result is 2



Activity 3.1

1. Write down three key reasons why Python is a good choice for beginners and professionals.
2. Discuss these reasons with your station partner and come up with one unique use case where Python would be particularly effective (e.g., data analysis, web development, automation).
3. Demonstrate your understanding by installing Python on your system and writing a simple "Hello, World!" program.

Materials Needed:

- Laptops or computers with Python installed (or internet access for installation steps).
- Whiteboard/Blackboard and markers.
- Handouts for each station (containing instructions, examples, and tasks).

.....

.....

.....

.....

☒ Check Your Progress 3.1

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

1. What is an arithmetic operator in Python? Give two examples.

.....

.....

.....

.....

2. What is a comparison operator in Python? Give two examples?

.....

NOTE

3.5 SUMMARY

Operators are essential components in programming that enable various operations on data. Arithmetic operators perform mathematical calculations, while assignment operators store values in variables. Comparison operators help in evaluating conditions by comparing values. Logical operators are used for combining multiple conditions, influencing decision-making in programs. Identity operators check if two variables refer to the same object in memory, whereas membership operators determine if a value exists in a sequence like a list or string. Bitwise operators perform operations at the binary level, useful in low-level programming. A solid understanding of these operators enhances coding efficiency and logical problem-solving.

3.6 GLOSSARY

- **Arithmetic Operators:** Perform mathematical operations like addition, subtraction, multiplication, and division.
- **Assignment Operators:** Assign values to variables using =, +=, -=, etc.
- **Comparison Operators:** Compare values using ==, !=, >, <, >=, and <=.
- **Logical Operators:** Combine multiple conditions using and, or, and not.
- **Identity Operators:** Check if two variables refer to the same object (is, is not).
- **Membership Operators:** Determine if a value exists in a sequence (in, not in).
- **Bitwise Operators:** Perform operations at the binary level, including &, |, ^, ~, <<, and >>.

3.7 ANSWERS TO CHECK YOUR PROGRESS

1. What is an arithmetic operator in Python? Give two examples.

Answer:

Arithmetic operators are used to perform basic mathematical operations like addition, subtraction, multiplication, and division.

NOTE**Example:**

```
a = 10
```

```
b = 5
```

```
print(a + b) # Addition → Output: 15
```

```
print(a * b) # Multiplication → Output: 50
```

2. What is a comparison operator in Python? Give two examples.**Answer:**

Comparison operators are used to compare two values and return a Boolean result (True or False).

Example:

```
x = 10
```

```
y = 5
```

```
print(x > y) # Greater than → Output: True
```

```
print(x == y) # Equal to → Output: False
```

3.8 SUGGESTED READINGS

Sr. No.	Title	URL	QR Code
1	Open Educational Resource	http://books.aisectionline.com/UnitUrls?unitId=1278	

3.9 REFERENCE

- Lutz, M. (2013). *Learning Python (5th Edition)*. O'Reilly Media.
- Severance, C. (2016). *Python for Everybody: Exploring Data in Python 3*.
- The Python Software Foundation. (n.d.). *Python Documentation*. Retrieved from <https://docs.python.org>

3.10 EXERCISE**Part 1****A. Multiple Choice Questions (MCQs)****1. What is one of the main reasons Python is popular?**

- Complicated syntax
- Easy to read and write code
- Requires a compiler
- Uses only functional programming

NOTE

2. Where should you download Python for installation?

- a) www.pythoninstall.com
- b) www.python.org
- c) www.pythondownload.net
- d) www.codepython.com

3. Which statement about Python syntax is true?

- a) Python uses semicolons at the end of every statement
- b) Python requires indentation to structure code
- c) Python does not allow whitespace in code
- d) Python uses {} to define code blocks

4. Which of the following is NOT an advantage of Python?

- a) Readable syntax
- b) Large community support
- c) Limited to web development only
- d) Supports multiple programming paradigms

5. Which command verifies Python installation?

- a) python -verify
- b) python check
- c) python -version
- d) verify python

Answers with Explanations:

1. **b) Easy to read and write code** - Python is popular because its simple, readable syntax makes coding faster and easier.
2. **b) www.python.org** - This is the official and safest website to download Python.
3. **b) Python requires indentation to structure code** - Python uses indentation instead of braces to define code blocks.
4. **c) Limited to web development only** - Python is versatile and used in many fields, not just web development.
5. **c) Python-version** - This command shows the installed Python version, confirming installation.

NOTE

B. Short Answer Type Questions

1. Explain the advantages of using Python over other programming languages.
2. Describe the steps to install Python on Windows and macOS.
3. What role does indentation play in Python syntax?

Part 2

C. Long Answer Type Questions (200 words)

1. Explain in detail why Python is a widely used programming language. Include its advantages, use cases, and major industries where it is applied.
2. Describe the complete process of installing Python on different operating systems. Include setting up an interpreter and verifying installation.

Part 3

D. Long Answer Type Questions (300 words)

1. Discuss the different types of operators in Python—arithmetic, comparison, logical, assignment, identity, and membership operators. Explain their functionality with examples.
2. Explain the concept of operator precedence and associativity in Python. Discuss how these rules affect the evaluation of complex expressions.
3. Describe the role of logical, relational, and bitwise operators in decision-making and computations. Provide examples where combined operators are used in programs



DR. C.V. RAMAN UNIVERSITY

Chhattisgarh, Bilaspur AN A/SECT GROUP UNIVERSITY

Approved by: PCI | AICTE | NISTE | BCI | Member of: ANU | Joint Committee: UGC | DSR | AICTE | Recognized by: BGC | A MAC Accredited University



DR. C. V. RAMAN UNIVERSITY (C.G.)

SLM FEEDBACK FORM

PROGRAMME : _____

COURSE : _____

BLOCK : _____

UNIT : _____

Dear Student,

While studying the units of this block, you may have found certain portions of the text difficult to comprehend. We wish to know your difficulties and suggestions, in order to improve the course. Therefore, we request you to fill up and send us the following questionnaire, which pertains to this block. If you find the space provided insufficient, kindly use a separate sheet.

Please mail to aisectcontent@aisect.org/ for online submission please scan QR code:

Questionnaire

1. Name : _____

2. Enrolment No. :

--	--	--	--	--	--	--	--	--	--	--

3. How many hours did you need for studying the units? :

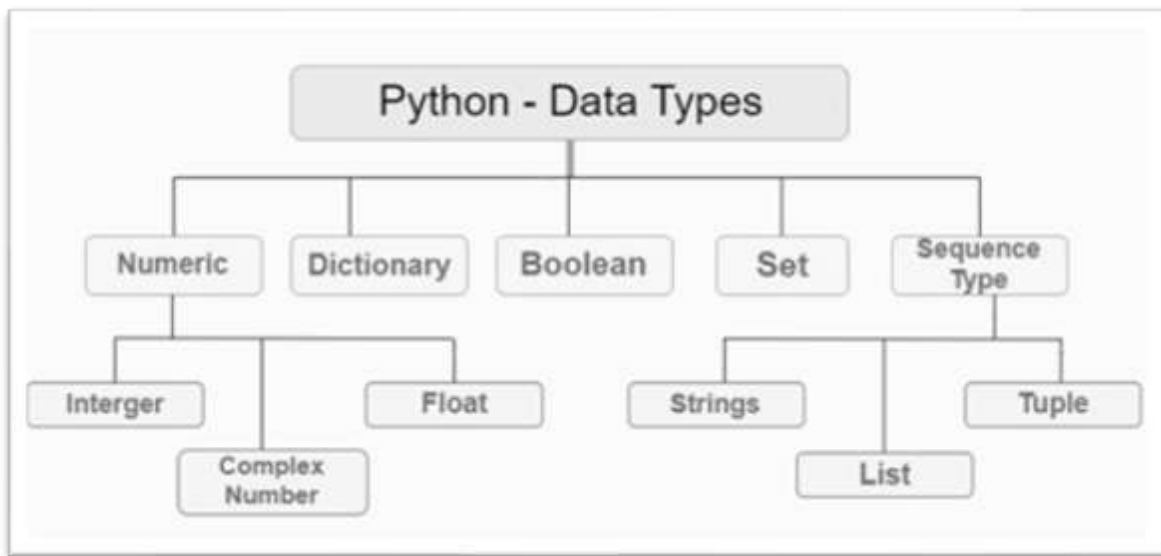
--

4. Please give your reactions to the following items based on your reading of the block:

Items	Excellent	Very Good	Good	Poor	Give specific Examples (If poor)
Presentation Quality					
Language and Style					
Illustrations					
Used (diagrams, tables, etc.)					
Conceptual Clarity					
Check Your Progress Questions					
Suggested Readings - OER					

5. Any other comments:

BLOCK II	BUILT-IN DATA TYPES AND THEIR METHODS	37-66
Unit 4	Numbers, Booleans, and Type Conversion	39
Unit 5	Strings and Their Operations	47
Unit 6	Collections – Lists, Tuples, Dictionaries, and Sets	55



This block helps students understand how Python stores, organizes, and manipulates different kinds of data. Since data is the core of any program, learning these built-in types is essential for problem-solving and real-world application development. Students will explore numeric types, boolean logic, strings, and widely used collection types such as lists, tuples, dictionaries, and sets. The block emphasizes practical usage, built-in methods, indexing, slicing, and operations that make Python a powerful and flexible language. By the end of this block, students will be able to choose the right data type for a given task, apply appropriate methods to process information, and write efficient, readable code for storing and manipulating data.

UNIT 4

NUMBERS, BOOLEANS, AND TYPE CONVERSION

NOTE

UNIT STRUCTURE

- 4.1 Introduction
- 4.2 Objective
- 4.3 Numbers (int, float, complex)
- 4.4 Casting
- 4.5 Booleans
- 4.6 Summary
- 4.7 Glossary
- 4.8 Answers to check your progress
- 4.9 Suggested readings - OER
- 4.10 Reference
- 4.11 Exercise

4.1 INTRODUCTION

“In Python, numbers are a fundamental data type used for mathematical and computational operations. Python supports three main types of numbers: integers (int) for whole numbers, floating-point numbers (float) for decimal values, and complex numbers (complex) for handling imaginary numbers. Type casting allows the conversion of one data type into another, such as converting a string to an integer or a float. Booleans (bool) represent True or False values and are crucial for decision-making and control flow in programming. Understanding these concepts helps in handling numerical data efficiently and making logical decisions in Python programs”.

4.2 OBJECTIVE

After completing this unit, you will be capable of:

- To understand the different numeric types in Python: integers, floats, and complex numbers.
- To learn how type casting works and how to convert between data types.
- To explore Boolean values and their role in logical operations and control structures.
- To develop a deeper understanding of how numerical and Boolean values interact in expressions and conditions.

NOTE

4.3 NUMBERS (INT, FLOAT, COMPLEX)

Three categories of numerical data are supported by Python: complex numbers, floating-point numbers (float), and integers (int).

- **int:** Represents whole numbers without a fractional part. Examples include 1, -3, and 10.
- **float:** Represents numbers with a decimal point, also known as real numbers. Examples include 3.14, -2.0, and 0.99.
- **complex:** Represents numbers that consist of a 'real' part and an 'imaginary' part. Examples include $2 + 3j$.

Example Program:

```
# Defining variables
a = 10    # int
b = 3.14  # float
c = 2 + 3j # complex

# Printing values and types
print(a, type(a))
print(b, type(b))
print(c, type(c))
```

Explanation:

1. The program defines three variables a, b, and c with different numeric types: int, float, and complex.
2. The print function displays the value of each variable along with its type, which is determined using the type() function.
3. The output will be:
10 <class 'int'>
3.14 <class 'float'>
(2+3j) <class 'complex'>

This clearly distinguishes integers, floating-point numbers, and complex numbers, demonstrating how Python handles different numeric types.

4.4 CASTING

The process of altering a variable's data type is called casting. This can be accomplished in Python by utilizing functions like str(), float(), and int(), among others. When working with different data types, casting is essential since Python can handle type conversions automatically in some situations but may need human intervention in others.

- **Implicit Casting:** When Python converts one data type to another, it does so automatically and without explicit guidance. For

example, Python smoothly transforms an integer into a float when doing a calculation that involves both an integer and a float.

- **Explicit Casting:** Occurs when the programmer deliberately converts a data type by using casting functions like `int()`, `float()`, or `str()`.

Example Program:

```
a = 10
b = 3.14
c = str(x)

print("Integer:", a)
print("Float:", b)
print("String:", c)
```

Explanation:

1. **a = 10:** Integer variable.
2. **b = 3.14:** Float variable.
3. **c = str(x):** The number an is converted to a string using the "str()" method.
4. The print statements display the original integer, float, and the string representation of a.

4.5 BOOLEANS

Booleans, which can only be True or False, are Python's representation of truth values. They are frequently used in conditional expressions like if and while and are essential to decision-making processes.

True: Represents a logical truth.

False: Represents a logical falsehood.

Booleans are often the result of comparison or logical operations. For example, the comparison `5 > 3` evaluates to True.

Example Program:

```
a = 10
b = 5
is_greater = a > b
print(is_greater)
```

Explanation:

`a = 10` and `b = 5`: Two integer variables.

`is_greater = a > b`: The comparison `10 > 5` evaluates to True.

The `print(is_greater)` statement outputs True because the condition is true.

NOTE

NOTE**Activity 4.1****1. Exploring Number Types**

- Create three variables: an integer, a float, and a complex number.
- Print the type of each variable using the `type()` function.

2. Type Casting Practice

- Convert an integer to a float and print the result.
- Convert a float to an integer and print the result.
- Convert a numeric string ("50") to an integer.

Materials Needed:

- A computer with Python installed
- A text editor or Python IDE (e.g., VS Code, PyCharm, Jupyter Notebook)

.....

.....

.....

.....

☑ Check Your Progress 4.1

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

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

```
x = 7.8
y = int(x)
print(y).
```

.....

.....

.....

.....

2. How does Python evaluate Boolean values for numbers?

.....

.....

.....

.....

4.6 SUMMARY

Python provides three primary numeric data types: integers for whole numbers, floats for decimal numbers, and complex numbers for imaginary values. Type casting enables the conversion of one data type into another, making it easier to work with different kinds of numerical inputs. Booleans, which represent True and False, play a key role in decision-making processes by evaluating conditions and controlling program flow. Mastering these concepts ensures efficient numerical computations and logical operations in Python programming.

4.7 GLOSSARY

- **Integer (int)** – A whole number without a decimal point (e.g., 10, -3).
- **Float (float)** – A number with a decimal point (e.g., 3.14, -0.5).
- **Complex (complex)** – A number with a real and an imaginary part (e.g., $2 + 3j$).
- **Casting** – Converting one data type into another (e.g., `int(3.5) → 3`).
- **Boolean (bool)** – A data type representing True or False.
- **Logical Expression** – An expression that evaluates to a Boolean value (True or False).
- **Control Flow** – The decision-making process in a program using Boolean values in conditions (e.g., `if`, `while`).

4.8 ANSWERS TO CHECK YOUR PROGRESS

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

```
x = 7.8
y = int(x)
print(y)
```

Answer:

The output will be 7, because `int()` removes the decimal part when converting a float to an integer.

2. How does Python evaluate Boolean values for numbers?

Answer:

In Python, any non-zero number is considered True, while 0 is considered False. For example, `bool(5)` returns True, whereas `bool(0)` returns False.

NOTE

NOTE**4.9 SUGGESTED READINGS**

Sr. No.	Title	URL	QR Code
1	Open Educational Resource	http://books.aisectionline.com/UnitUrls?unitId=1280	

4.9 REFERENCE

- Python Software Foundation. (2024). *Python 3 Documentation – Data Types and Operators*. Retrieved from <https://docs.python.org/3/>
- Sweigart, A. (2020). *Automate the Boring Stuff with Python (2nd ed.)*. No Starch Press.
- Matthes, E. (2019). *Python Crash Course (2nd ed.)*. No Starch Press.

4.10 EXERCISE**Part 1****A. Multiple Choice Questions (MCQs)**

- Which of the following is an example of an integer in Python?**
 - 3.14
 - "10"
 - 10
 - 5 + 2j
- Which function is used to convert a float into an integer?**
 - float()
 - int()
 - str()
 - bool()
- What is the output of bool(0) in Python?**
 - True
 - False
 - None
 - 0

4. Which number type is used to store complex numbers in Python?

- a) float
- b) int
- c) complex
- d) boolean

5. What is the result of bool(5) in Python?

- a) True
- b) False
- c) None
- d) 5

NOTE

Answers with Explanations:

1. **c) 10** - is a whole number without decimals, so Python treats it as an int type.
2. **b) int()** - converts a float to an integer by removing the decimal part (truncates the value).
3. **b) False** - In Python, 0 is considered a “falsy” value, so converting it to bool returns False.
4. **c) complex** - Python uses the complex type to store numbers with real and imaginary parts (like 5+2j).
5. **a) True** - Any non-zero number is considered “truthy” in Python, so bool(5) returns True.

B. Short Answer Type Questions

1. What is the difference between int, float, and complex in Python?
2. Explain how type casting works in Python with an example.
3. How are Boolean values used in conditional statements in Python?

Part 2

C. Long Answer Type Questions (200 words)

1. Explain the different numerical data types in Python with examples. Discuss how type casting helps in data manipulation.
2. What are Boolean values in Python? Explain their role in logical operations and decision-making with examples.

NOTE**D. Long Answer Type Questions (300 words)**

1. Explain Python's numeric data types (int, float, complex). Discuss how numbers are stored, manipulated, and used in mathematical expressions.
2. Describe Boolean values in Python. Discuss truth values, comparison operations, and how Booleans are used in logical expressions and decision-making.
3. Explain type conversion in Python. Differentiate between implicit and explicit conversion and discuss common conversion functions with examples.

UNIT 5

STRINGS AND THEIR OPERATIONS

NOTE

UNIT STRUCTURE

- 5.1 Introduction
- 5.2 Object
- 5.3 Strings (slicing, modify)
- 5.4 Strings (concatenate, format, escape)
- 5.5 Summary
- 5.6 Glossary
- 5.7 Answers to check your progress
- 5.8 Suggested readings - OER
- 5.9 Reference
- 5.10 Exercise

5.1 INTRODUCTION

Strings are one of the most important data types in Python, used to store and manipulate text. Python provides various operations to handle strings effectively, including slicing (extracting specific portions of a string), modifying (changing case, replacing characters, etc.), concatenating (joining multiple strings), formatting (inserting values dynamically into a string), and using escape characters (handling special characters within a string). Understanding these operations allows programmers to work efficiently with textual data and enhance the flexibility of their programs.

5.2 OBJECTIVE

After completing this unit, you will be capable of:

- To understand the concept of strings and how they are used in Python.
- To learn how to slice strings to extract specific portions.
- To explore different ways to modify strings, such as changing case and replacing text.
- To understand string concatenation and how to join multiple strings.
- To learn how string formatting works for inserting dynamic values.
- To use escape sequences to handle special characters in strings.

NOTE**5.3 STRINGS (SLICING, MODIFY)****What is String Slicing?**

String slicing allows us to extract a specific portion (or substring) from a string using indexing. Strings in Python are **indexed** from left to right (starting at 0) and from right to left (starting at -1).

The syntax for slicing a string is:

```
string[start:stop:step]
```

- start: The index where the slice starts (inclusive).
- stop: The index where the slice ends (exclusive).
- step: The interval of characters to take (optional).

Examples of String Slicing**1. Basic Slicing (Extracting a Substring)**

```
text = "Python Programming"
print(text[0:6]) # Output: Python
```

Explanation: The slice [0:6] extracts characters from index 0 to 5 (excluding 6).

2. Omitting Start or Stop

```
print(text[:6]) # Output: Python (same as text[0:6])
print(text[7:]) # Output: Programming (extracts from index 7 to end)
```

Explanation:

- [:6] extracts from the beginning (default 0) to index 5.
- [7:] extracts from index 7 to the end.

3. Using Negative Indexing

```
print(text[-11:]) # Output: Programming
print(text[: -8]) # Output: Python
```

Explanation:

- text[-11:] extracts from index -11 (backward count) to the end.
- text[: -8] extracts everything except the last 8 characters.

4. Using Step in Slicing

```
print(text[0:12:2]) # Output: Pto rg
```

Explanation:

- The step value 2 means every second character is taken (P, t, o, space, r, g).

5. Reversing a String using Slicing

```
print(text[::-1]) # Output: gnimmargorP nohtyP
```

Explanation:

- `[::-1]` means start from the end and move backward, effectively reversing the string.

NOTE

5.4 MODIFYING STRINGS

Unlike lists, strings in Python are **immutable**, meaning their characters cannot be changed directly. However, we can modify strings using various built-in methods that return a new modified string.

Examples of String Modification

1. Changing Case of Strings

```
message = "Hello World"
print(message.upper()) # Output: HELLO WORLD
print(message.lower()) # Output: hello world
print(message.title()) # Output: Hello World
print(message.capitalize()) # Output: Hello world
```

Explanation:

- `.upper()` converts all characters to uppercase.
- `.lower()` converts all characters to lowercase.
- `.title()` capitalizes the first letter of each word.
- `.capitalize()` capitalizes only the first letter of the string.

2. Replacing Substrings

```
text = "Python is fun"
new_text = text.replace("fun", "awesome")
print(new_text) # Output: Python is awesome
```

Explanation:

- `.replace("old", "new")` replaces all occurrences of "fun" with "awesome".

3. Removing Whitespaces

```
text = " Python "
print(text.strip()) # Output: Python
print(text.lstrip()) # Output: Python
print(text.rstrip()) # Output: Python
```

Explanation:

- `.strip()` removes spaces from both ends.
- `.lstrip()` removes spaces from the left.
- `.rstrip()` removes spaces from the right.

NOTE**4. Splitting and Joining Strings**

```
sentence = "Python is fun to learn"
words = sentence.split() # Splits based on spaces by default
print(words) # Output: ['Python', 'is', 'fun', 'to', 'learn']

joined_sentence = "-".join(words)
print(joined_sentence) # Output: Python-is-fun-to-learn
```

Explanation:

- .split() breaks a string into a list of words.
- .join() combines words into a single string with - as a separator.

5. Checking for Substrings

```
text = "I love Python"
print("Python" in text) # Output: True
print("Java" not in text) # Output: True
```

Explanation:

- "Python" in text checks if "Python" is present in the string.
- "Java" not in text returns True since "Java" is not in the string.

**Activity 5.1****1. String Slicing and Modification**

- Write a Python program that takes a string as input and:
 - o Extracts and prints the first five characters using slicing.
 - o Converts the entire string to uppercase.
 - o Replaces all occurrences of the letter 'a' with '@'.

2. String Concatenation and Formatting

- Create two string variables: first_name and last_name.
 - o Concatenate them to display the full name.
 - o Use format() to create a sentence like "My name is John Doe" dynamically.
 - o Use f-strings to display a similar sentence.

Materials Needed:

- Python interpreter (IDLE, VS Code, or Jupyter Notebook).
- Sample input strings to test the program.

```
.....
.....
.....
.....
```

☑ Check Your Progress 5.1

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

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

```
text = "Python Programming"
print(text[7:])
```

.....

2. How does the escape sequence `/t` work in Python strings?

.....

NOTE**5.5 SUMMARY**

Strings in Python offer various functionalities for efficient text manipulation. Slicing allows extracting specific parts of a string using indexing, while modification methods help in changing cases, replacing characters, or trimming whitespace. Concatenation enables joining multiple strings, making it useful for constructing meaningful text. String formatting provides dynamic insertion of values, allowing better readability and flexibility. Escape characters help handle special characters like newlines (`\n`) or quotes within a string. Mastering these operations enhances text processing and automation in Python programming.

5.6 GLOSSARY

- **String (str)** – A sequence of characters enclosed in quotes ("Hello", 'Python').
- **Slicing** – Extracting a portion of a string using indexing (`str[0:5]`).
- **Modify** – Changing a string's content using methods like `.upper()`, `.lower()`, `.replace()`, etc.
- **Concatenation** – Joining multiple strings using the `+` operator ("Hello" + "World" → "HelloWorld").
- **Formatting** – Inserting values dynamically into a string using `.format()`, f-strings (`f'Hello {name}'`), or `%` formatting.

NOTE**5.7 ANSWERS TO CHECK YOUR PROGRESS****1. What will be the output of the following code?**

```
text = "Python Programming"
print(text[7:])
```

Answer:

The output will be "Programming", as slicing [7:] extracts characters from index 7 to the end of the string.

2. How does the escape sequence \t work in Python strings?**Answer:**

The escape sequence \t inserts a **tab space** in a string. For example:

```
print("Hello\tWorld")
```

This prints:

```
nginx
```

```
CopyEdit
```

```
Hello World
```

where \t adds extra spacing between "Hello" and "World".

5.8 SUGGESTED READINGS - OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource	http://books.aisectionline.com/UnitUrls?unitId=1280	

5.9 REFERENCE

- Lutz, M. (2013). *Learning Python (5th Edition)*. O'Reilly Media.
- Zelle, J. (2016). *Python Programming: An Introduction to Computer Science (3rd Edition)*. Franklin, Beedle & Associates.
- Downey, A. (2015). *Think Python: How to Think Like a Computer Scientist*. O'Reilly Media.

5.10 EXERCISE**Part 1****A. Multiple Choice Questions (MCQs)****1. What will be the output of the following code?**

```
txt = "Python Programming"
print(txt[0:6])
```

NOTE

- a) Python Programming
- b) Python
- c) PyProg
- d) Pytho

2. Which of the following methods is used to remove whitespace from a string?

- a) .strip()
- b) .replace()
- c) .split()
- d) .remove()

3. What does \n represent in a string?

- a) New space
- b) Newline
- c) Tab space
- d) Escape character

4. Which method is used to format a string dynamically?

- a) .format()
- b) .join()
- c) .concat()
- d) .split()

5. Which of the following is the correct way to concatenate two strings?

- a) "Hello".add("World")
- b) "Hello" & "World"
- c) "Hello" + " " + "World"
- d) "Hello".merge("World")

Answers with Explanations:

- 1. b) Python** - txt[0:6] slices characters from index 0 to 5. So it extracts the first 6 letters
- 2. a) .strip()** - .strip() removes whitespace from the beginning and end of a string. It's commonly used for cleaning user input.
- 3. b) Newline** - \n is an escape sequence that moves the cursor to a new line. It's used for formatting multi-line text.
- 4. a) .format()** - .format() inserts values into placeholders {} inside a string. It helps create flexible and dynamic output.

NOTE

5. c) **"Hello" + " " + "World"** - Using the + operator joins strings together in Python. This also allows inserting spaces or other characters easily.

B. Short Answer Type Questions

1. Explain how string slicing works in Python with an example.
2. What are escape sequences in Python? Give three examples.
3. What is the difference between format() and f-strings in Python string formatting?

Part 2**C. Long Answer Type Questions (200 words)**

1. Explain string manipulation techniques in Python with examples. Discuss string slicing, modification, and concatenation in detail.
2. What are different ways to format strings in Python? Compare format(), f-strings, and concatenation, and explain when to use each.

Part 3**D. Long Answer Type Questions (300 words)**

1. Discuss the structure and properties of Python strings. Explain indexing, slicing, and immutability with examples.
2. Explain commonly used string methods such as ``upper()``, ``lower()``, ``replace()``, ``split()``, ``join()``, and formatting techniques. Discuss their importance in text processing.
3. Describe escape sequences and raw strings in Python. Explain how they help in handling special characters and multi-line strings

UNIT 6

COLLECTIONS – LISTS, TUPLES, DICTIONARIES, AND SETS

NOTE

UNIT STRUCTURE

- 6.1 Introduction
- 6.2 Object
- 6.3 List
- 6.4 Tuples
- 6.5 Dictionary
- 6.6 Set
- 6.7 Summary
- 6.8 Glossary
- 6.9 Answers to check your progress
- 6.10 Suggested readings
- 6.11 Reference
- 6.12 Exercise
- 6.13 Feedback form

6.1 INTRODUCTION

Collections in Python are specialized data structures used to store and manage multiple values efficiently. The four main types of collections are **Lists, Tuples, Dictionaries, and Sets**. **Lists** are ordered, mutable sequences that allow duplicate elements. **Tuples** are similar to lists but immutable, meaning their values cannot be changed after creation. **Dictionaries** store key-value pairs, allowing fast lookups and efficient data organization. **Sets** are unordered collections that do not allow duplicate elements, making them useful for mathematical operations like unions and intersections. Understanding these collections is essential for effective data handling and manipulation in Python.

6.2 OBJECTIVE

After completing this unit, you will be capable of:

- To understand the characteristics and use cases of Lists, Tuples, Dictionaries, and Sets.
- To learn how to create, access, and modify elements in these collections.
- To explore the differences between mutable (Lists, Dictionaries, Sets) and immutable (Tuples) collections.
- To use dictionaries for efficient key-value storage and retrieval.

NOTE

- To apply sets for eliminating duplicates and performing mathematical operations.

6.3 LIST

Python lists are ordered, mutable collections that can store a variety of data types.

Adding items, deleting items, splitting portions, and iterating over elements are common operations on lists.

- **Append:** Inserts an element at the list's end.
- **Remove:** Eliminates the first instance of a given element.
- **Slicing:** Extracts a defined portion of the list.
- **Iteration:** Enables traversing through all elements in the list.

Example Program:

```
fruits = ["apple", "banana", "cherry"]
# Append
fruits.append("orange")
print(fruits) # Output: ['apple', 'banana', 'cherry', 'orange']
# Remove
fruits.remove("banana")
print(fruits) # Output: ['apple', 'cherry', 'orange']
# Slicing
print(fruits[1:3]) # Output: ['cherry', 'orange']
```

Explanation:

1. **Append:** Adds "orange" to the list.
2. **Remove:** Removes "banana" from the list.
3. **Slicing:** Takes a section of the list (from index 1 to index 2) and extracts it.

6.4 C TUPLES

Python tuples are fixed sequences, meaning their values cannot be modified once created. They support operations like indexing and slicing, making them useful for grouping related information.

Example Program:

```
coordinates = (10, 20, 30)

# Accessing elements
# Output: 10
print(coordinates[0])

# Slicing
print(coordinates[1:3]) # Output: (20, 30)
```

Explanation:

1. **Accessing elements:** Retrieves the first item of the tuple.
2. **Slicing:** Extracts a portion of the tuple.

NOTE

6.5 DICTIONARY

Dictionaries are collections of key-value pairs that are not arranged in a certain order. Dictionaries are an excellent tool for lookups and categorizing related data because each key is distinct and correlates to a certain value.

Example Program:

```
student = {"name": "Alice", "age": 21, "grade": "A"}

# Accessing values
print(student["name"]) # Output: Alice

# Adding new key-value pair
student["major"] = "Computer Science"
print(student) # Output: {'name': 'Alice', 'age': 21, 'grade': 'A', 'major': 'Computer Science'}
```

Explanation:

1. **Accessing values:** Retrieves the value associated with the "name" key.
2. **Adding new key-value pair:** Adds a new entry for "major" in the dictionary.

6.6 SET

Collections of unique items without a set sequence are called sets. They are perfect for carrying out operations including union, intersection, and difference as well as keeping unique elements.

- **Adding Elements:** Use the add() method to add elements to a set.
- **Eliminating Elements:** A set's elements can be eliminated using the remove() method.
- **Set Operations:** You can work with sets using operations like difference (-), intersection (&), and union (|).

Example Program:

```
my_set = {1, 2, 3}
my_set.add(4) # Adding element
print(my_set)
another_set = {3, 4, 5}
intersection_set = my_set & another_set # Intersection
print(intersection_set)
```

NOTE**Explanation:**

1. **my_set = {1, 2, 3}**: Creates a 'set' with three elements.
2. **my_set.add(4)**: Adds 4 to the 'set'.
3. **print(my_set)**: Prints the updated set, which is {1, 2, 3, 4}.
4. **another_set = {3, 4, 5}**: Creates another set.
5. **intersection_set = my_set & another_set**: Finds the intersection of my_set and another_set, which is {3, 4}.
6. **print(intersection_set)**: Prints the intersection set.

**Activity 6.1****1. Understanding Data Collections**

Write a Python program to create a list, tuple, dictionary, and set with sample data. Print the contents of each collection.

2. Modifying and Accessing Data

- Modify the list by adding a new element.
- Attempt to modify the tuple and observe what happens.
- Retrieve a value from the dictionary using a key.
- Add a new unique value to the set.

Materials Needed:

- A computer with Python installed
- Any text editor or Python IDE (e.g., PyCharm, VS Code, Jupyter Notebook)

.....

☑ Check Your Progress 6.1

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

1. How are lists and tuples different in Python? Provide an example to illustrate the difference.

.....

2. How does a dictionary handle duplicate keys?

.....

NOTE

6.7 SUMMARY

Python provides four primary collection types: **Lists**, which are ordered, mutable sequences; **Tuples**, which are immutable and efficient for fixed data; **Dictionaries**, which store data in key-value pairs for fast access; and **Sets**, which contain unique elements and allow mathematical operations. Lists and dictionaries are widely used for dynamic data storage, while tuples provide read-only data integrity, and sets help eliminate duplicates. Mastering these collections improves data management, efficiency, and performance in Python programming.

6.8 GLOSSARY

- **List (list)** – An ordered, mutable sequence that allows duplicate elements (e.g., [1, 2, 3]).
- **Tuple (tuple)** – An ordered, immutable sequence (e.g., (1, 2, 3)).
- **Dictionary (dict)** – A collection that stores key-value pairs (e.g., {"name": "Alice", "age": 25}).
- **Set (set)** – An unordered collection of unique elements (e.g., {1, 2, 3}).
- **Mutable** – Can be modified after creation (Lists, Dictionaries, Sets).
- **Immutable** – Cannot be modified after creation (Tuples).
- **Key-Value Pair** – A structure in dictionaries where each key maps to a specific value.
- **Duplicate Elements** – Values that appear more than once (allowed in Lists, not in Sets).

6.7 ANSWERS TO CHECK YOUR PROGRESS

1. How are lists and tuples different in Python? Provide an example to illustrate the difference.

Answer:

Lists and tuples are both ordered collections of elements, but lists are **mutable**, meaning their content can be modified, while tuples are **immutable** and cannot be changed once created.

NOTE**Example:**

```
# List (Mutable)
my_list = [1, 2, 3]
my_list.append(4) # Adding a new element
print(my_list) # Output: [1, 2, 3, 4]

# Tuple (Immutable)
my_tuple = (1, 2, 3)

# my_tuple.append(4) # This will cause an error because tuples
# cannot be modified

print(my_tuple) # Output: (1, 2, 3)
```

2. How does a dictionary handle duplicate keys?**Answer:**

In a Python dictionary, **duplicate keys are not allowed**. If a dictionary is created with duplicate keys, the last assigned value will overwrite the previous one.

Example:

```
my_dict = {"name": "Alice", "age": 25, "name": "Bob"}
print(my_dict) # Output: {'name': 'Bob', 'age': 25}
```

Here, the second occurrence of "name" replaces the first one, so "Alice" is overwritten by "Bob".

6.8 SUGGESTED READINGS

Sr. No.	Title	URL	QR Code
1	Open Educational Resource	http://books.aisectionline.com/User/UnitUrls?unitId=1281	

6.9 REFERENCE

- *Python Crash Course* by Eric Matthes – No Starch Press
- *Automate the Boring Stuff with Python* by Al Sweigart
- *Python Programming for Beginners* - John Code

6.10 EXERCISE**NOTE****Part 1****A. Multiple Choice Questions (MCQs)****1. Which of the following data structures is mutable?**

- a) Tuple
- b) List
- c) Set
- d) Dictionary

2. Which Python collection type stores data as key-value pairs?

- a) List
- b) Tuple
- c) Dictionary
- d) Set

3. Which of the following statements about sets is true?

- a) Sets allow duplicate values
- b) Sets maintain order
- c) Sets are mutable
- d) Sets store unique elements

4. How do you create an empty dictionary in Python?

- a) {}
- b) []
- c) dict()
- d) Both a and c

5. Which method is used to add an element to a list?

- a) .append()
- b) .add()
- c) .insert()
- d) .update()

Answers with Explanations:

- 1. b) List** – Lists can be modified after creation (add, remove, update elements). Tuples are immutable, while sets & dictionaries are mutable but not in the same way for ordered data.
- 2. c) Dictionary** - Dictionaries store data in pairs like {key: value}. They allow fast lookups based on keys.

NOTE

3. **d) Sets store unique elements** - A set automatically removes duplicates and keeps only unique values. It is unordered and does not allow repeated items.
4. **d) Both a and c** - {} and dict() both create an empty dictionary in Python. They are two equivalent ways but {} is more commonly used.
5. **a) .append()** - .append() adds a new item at the end of the list. It is the most common method for single-value insertion.

B. Short Answer Type Questions (Without Answers)

1. What are the key differences between Lists and Tuples?
2. How does a dictionary handle duplicate keys?
3. Explain three real-life use cases of Sets in Python.

Part 2**C. Long Answer Type Questions (200 words)**

1. Compare and contrast Lists, Tuples, Dictionaries, and Sets in Python. Include examples and discuss their advantages and disadvantages.
2. Explain how Python's Dictionary data structure works. Discuss different dictionary methods such as get(), keys(), values(), items(), and update() with examples.

Part 3**D. Long Answer Type Questions (300 words)**

1. Compare and contrast Python's built-in collection types: lists, tuples, sets, and dictionaries. Explain their properties and use cases.
2. Discuss list operations such as insertion, deletion, slicing, searching, and sorting. Explain how lists support dynamic data manipulation.
3. Explain how dictionaries store data using key-value pairs. Discuss adding, modifying, deleting values, and iterating through dictionary items.



DR. C.V. RAMAN UNIVERSITY

Chhattisgarh, Bilaspur AN A/SECT GROUP UNIVERSITY

Approved by: PCI | AICTE | NISTE | BCI | Member of: ANU | Joint Committee: UGC | DSR | AICTE | Recognized by: BGC | A MAC Accredited University



DR. C. V. RAMAN UNIVERSITY (C.G.)

SLM FEEDBACK FORM

PROGRAMME : _____

COURSE : _____

BLOCK : _____

UNIT : _____

Dear Student,

While studying the units of this block, you may have found certain portions of the text difficult to comprehend. We wish to know your difficulties and suggestions, in order to improve the course. Therefore, we request you to fill up and send us the following questionnaire, which pertains to this block. If you find the space provided insufficient, kindly use a separate sheet.

Please mail to aisectcontent@aisect.org/ for online submission please scan QR code:

Questionnaire

1. Name : _____

2. Enrolment No. :

--	--	--	--	--	--	--	--	--	--	--

3. How many hours did you need for studying the units? :

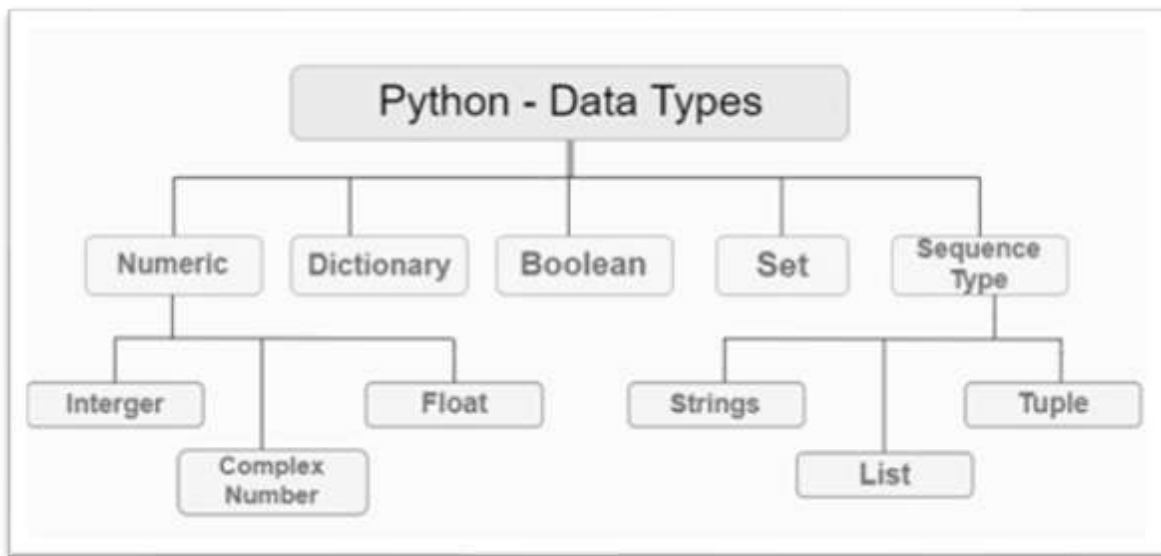
--

4. Please give your reactions to the following items based on your reading of the block:

Items	Excellent	Very Good	Good	Poor	Give specific Examples (If poor)
Presentation Quality					
Language and Style					
Illustrations					
Used (diagrams, tables, etc.)					
Conceptual Clarity					
Check Your Progress Questions					
Suggested Readings - OER					

5. Any other comments:

BLOCK III	DECISION MAKING, LOOPS AND FUNCTIONS	65-102
Unit 7	Decision Making	67
Unit 8	Loops & Iteration	75
Unit 9	Functions & Lambda	87



This block helps students understand how Python stores, organizes, and manipulates different kinds of data. Since data is the core of any program, learning these built-in types is essential for problem-solving and real-world application development. Students will explore numeric types, boolean logic, strings, and widely used collection types such as lists, tuples, dictionaries, and sets. The block emphasizes practical usage, built-in methods, indexing, slicing, and operations that make Python a powerful and flexible language. By the end of this block, students will be able to choose the right data type for a given task, apply appropriate methods to process information, and write efficient, readable code for storing and manipulating data.

UNIT 7

DECISION MAKING

NOTE

UNIT STRUCTURE

- 7.1 Introduction
- 7.2 Objective
- 7.3 If, Elif, Else
- 7.4 Short Hand If, Short Hand If..Else, Nested If
- 7.5 Summary
- 7.6 Glossary
- 7.7 Answers to Check Your Progress.
- 7.8 Suggested readings - OER
- 7.9 Reference
- 7.10 Exercise

7.1 INTRODUCTION

Conditional statements in Python allow decision-making by executing different blocks of code based on certain conditions. The main conditional statements are **if**, **elif**, **else**, which enable branching logic. The **if** statement checks a condition and executes the corresponding code block. The **elif** (else if) statement provides additional conditions to check if the first one is false. The **else** statement executes when none of the preceding conditions are met. Python also provides **shorthand if** and **shorthand if..else** for writing concise conditional expressions. Additionally, **nested if** allows multiple levels of conditions inside another if block, making complex decision-making possible.

7.2 OBJECTIVE

After completing this unit, you will be capable of:

- To understand the use of if, elif, and else statements for decision-making.
- To learn how to use shorthand if and shorthand if..else for concise conditions.
- To explore nested if statements for handling multi-level conditions.
- To apply these concepts to control the flow of execution in Python programs.

7.3 IF, ELIF, ELSE

IF...ELSE in Python

The if...else structure in Python gives programmers the ability to decide what to do and run certain code depending on predetermined criteria. It is a crucial tool for managing a program's execution flow.

NOTE

1. IF Statement The if statement evaluates the condition. If the condition evaluates to True, the relevant code is run; otherwise, it is ignored.

Syntax:

```
if condition:
```

```
    # code to execute if the condition is true
```

Example:

```
age = 20
```

```
if age >= 18:
```

```
    print("You are eligible to vote.")
```

Explanation:

- The condition age >= 18 is checked.
- If “age” is 18 or “greater”, it prints "You are eligible to vote."

2. IF...ELSE Statement : When the condition is ‘False’, the if...else phrase adds a different code block to run.

Syntax:

```
if condition:
```

```
    # code to execute if the condition is true
```

```
else:
```

```
    # code to execute if the condition is false
```

Example:

```
age = 16
```

```
if age >= 18:
```

```
    print("You are eligible to vote.")
```

```
else:
```

```
    print("You are not eligible to vote.")
```

Explanation:

- If age is less than 18, the else block will execute and print "You are not eligible to vote."

3. “IF...ELIF...ELSE Statement” When there are several conditions to examine, the elif statement—short for "else if"—is utilized. When the first condition is False, it permits the consideration of further criteria.

Syntax:

```
if condition1:
```

```
    # code for condition1
```

```
elif condition2:
```

NOTE

```
# code for condition2
else:
    # code if all conditions are false
```

Example:

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

Explanation:

- If score is 90 or more, "Grade: A" is printed.
- If not, but score is 80 or higher, "Grade: B" is printed.
- Otherwise, "Grade: C" is printed.

7.4 SHORT HAND IF, SHORT HAND IF..ELSE, NESTED IF

- 4. Short Hand IF** The shorthand if statement allows a concise, one-line conditional expression.

Syntax:

```
if condition: action
```

Example:"

```
x = 10
y = 5
if x > y: print("x is greater than y")
"
```

Explanation:

- The condition `x > y` is checked.
- If it is True, "x is greater than y" is printed in a single line.

- 5. Short Hand IF...ELSE** A shorthand if...else statement combines both branches into one line using the ternary operator.

Syntax:

```
value = action_if_true if condition else action_if_false
```

Example:

```
age = 20
message = "Adult" if age >= 18 else "Minor"
print(message)
```

NOTE**Explanation:**

- The condition `age >= 18` is checked.
- If True, the message is set to "Adult"; otherwise, it is set to "Minor."

6. “Nested IF” An if statement nested inside another ‘if’ statement allows for the processing of more complicated criteria.

Syntax:

```
if condition1:
    if condition2:
        # code for both conditions being true
```

Example:

```
num = 10
if num > 0:
    if num % 2 == 0:
        print("Positive and Even")
    else:
        print("Positive and Odd")
else:
    print("Non-positive")
```

Explanation:

- The program first determines whether `num > 0`.
- It determines whether `num % 2 == 0` if true.
- Based on the second condition, it prints "Positive and Even" or "Positive and Odd." If the first condition is false, "Non-positive" is printed.

**Activity 7.1****1. Understanding Basic Conditionals**

Write a Python program that takes a number as input from the user and checks whether it is positive, negative, or zero using if-elif-else.

2. Implementing Short-Hand If-Else

Modify the above program to use short-hand if-else for a concise version.

Materials Needed:

- Python installed on your system
- Text editor (VS Code, PyCharm, or IDLE)
- A computer or mobile device with an internet connection

.....

.....

.....

.....

NOTE**☑ Check Your Progress 7.1**

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

1. What is the difference between if-elif-else and nested if statements?

.....

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

```
x = 10
if x > 5:
    if x < 15:
        print("Within range")
    else:
        print("Out of range")
```

.....

7.5 SUMMARY

Conditional statements in Python enable decision-making by executing specific blocks of code based on conditions. The if statement checks a condition, while elif provides additional conditions. The else statement runs when no conditions are met. Python also offers **shorthand if** for single-line conditions and **shorthand if..else** for compact conditional expressions. **Nested if** allows multiple conditional checks within an if block, enabling more complex logic. Mastering these concepts improves program flow control and logical structuring in Python applications.

7.6 GLOSSARY

- **Conditional Statement** – A programming construct that makes decisions based on conditions.
- **if Statement** – Executes code when a condition is met.
- **elif Statement** – Checks multiple conditions sequentially.
- **else Statement** – Runs code if none of the conditions are met.
- **Short-Hand if** – A single-line syntax for if statements.
- **Short-Hand if..else** – A single-line conditional expression (x = a if condition else b).
- **Nested if** – An if statement inside another if, enabling complex decision-making.

NOTE**7.7 ANSWERS TO CHECK YOUR PROGRESS****1. What is the difference between if-elif-else and nested if statements?**

Answer: The if-elif-else statement is used when multiple conditions need to be checked sequentially. It ensures that only one condition is executed. In contrast, a **nested if** statement is when an if statement is placed inside another if, allowing multiple levels of condition checking. Nested if is used when one condition depends on another.

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

```
x = 10
if x > 5:
    if x < 15:
        print("Within range")
    else:
        print("Out of range")
```

Answer:

The output will be:

Within range

Since x = 10 satisfies both conditions (x > 5 and x < 15), the inner if condition executes and prints "Within range".

7.8 SUGGESTED READINGS - OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource	http://books.aisectionline.com/UnitUrls?unitId=1282	

7.9 REFERENCE

- Lutz, M. (2013). **Learning Python**. O'Reilly Media.
- Guttag, J. (2016). **Introduction to Computation and Programming Using Python**. MIT Press.

7.10 EXERCISE**Part 1****A. Multiple Choice Questions (MCQs)****1. Which Python statement is used to check a condition?**

- a) loop

NOTE

- b) b) if
- c) condition
- d) case

2. Which statement allows checking multiple conditions in Python?

- a) else
- b) elif
- c) endif
- d) switch

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

```
python
CopyEdit
x = 5
print("Positive") if x > 0 else print("Negative")
```

- a) Positive
- b) Negative
- c) Error
- d) None

4. Which of the following statements is true about nested if?

- a) Nested if cannot have multiple conditions.
- b) An if statement inside another if is called nested if.
- c) Nested if requires a loop to work.
- d) if cannot be used more than once in a program.

5. Which of the following is the correct syntax for a short-hand if statement?

- a) if (condition): print("Hello")
- b) print("Hello") if condition else print("Bye")
- c) if condition print("Hello")
- d) print("Hello"): if condition

Answers with Explanations:

- 1. b) if** – The if statement is used in Python to check a condition and execute code when the condition is true.
- 2. b) elif** - elif allows checking multiple conditions after an initial if statement.
- 3. a) Positive** - Since x = 5 and x > 0 is true, the expression prints "Positive".
- 4. b) An if statement inside another if is called nested if** - When one if statement is placed inside another, it is known as a nested if.

NOTE

5. b) `print("Hello") if condition else print("Bye")` - This is the correct syntax for Python's short-hand (ternary) if-else statement.

B. Short Answer Type Questions

1. Explain the difference between if-elif-else and nested if statements.
2. Write a Python program that uses short-hand if..else to determine whether a number is even or odd.
3. What is the benefit of using short-hand if statements in Python?

Part 2**C. Long Answer Type Questions (200 words)**

1. Explain the working of if, elif, and else statements with appropriate examples. Describe their role in controlling the flow of execution in a Python program.
2. Discuss the concept of nested if statements with an example program. Explain situations where nested if is useful and its advantages and limitations.

Part 3**D. Long Answer Type Questions (300 words)**

1. Explain conditional statements (`if`, `elif`, `else`) and discuss how they control program flow. Provide examples of nested and multiple conditions.
2. Discuss real-life scenarios where decision-making logic is used in programs. Explain how conditions help automate decisions.
3. Explain the role of relational and logical operators in decision-making. Discuss complex conditions with examples.

UNIT 8

LOOPS & ITERATION

NOTE

UNIT STRUCTURE

- 8.1 Introduction
- 8.2 Objective
- 8.3 While (Break, Continue)
- 8.4 For (Break, Continue, Range, Nested)
- 8.5 Summary
- 8.6 Glossary
- 8.7 Answers to check your progress.
- 8.8 Suggested readings - OER
- 8.9 Reference
- 8.10 Exercise

8.1 INTRODUCTION

Loops in Python allow repeated execution of a block of code until a certain condition is met. The **while** loop runs as long as a specified condition remains True, while the **for** loop iterates over a sequence such as a list, tuple, or range. Python provides control statements like **break** (to exit the loop prematurely) and **continue** (to skip the current iteration and proceed to the next one). The **range()** function helps generate a sequence of numbers for iteration. Additionally, **nested loops** allow loops to be placed inside other loops for handling complex iteration tasks.

8.2 OBJECTIVE

After completing this unit, you will be capable of:

- To understand the working of while and for loops in Python.
- To learn how to use break to terminate loops early.
- To explore continue to skip certain iterations without stopping the loop.
- To use the range() function for controlled iteration in for loops.
- To implement nested loops for handling multi-level iterations.

8.3 WHILE (BREAK, CONTINUE)

The While Loop in Python

A while loop allows a block of code to run continuously as long as the specified condition remains True. It's especially helpful when the total number of iterations is not known in advance.

Syntax of the While Loop:

While condition: '# Code block to execute'

FUNDAMENTALS OF PYTHON

NOTE

- Every iteration starts with a check of the condition.
- The code inside the loop is run if the condition evaluates to True.
- The loop ends if the condition turns out to be False.

Using the Break Statement in a 'While Loop'

The break statement immediately ends the loop's execution, regardless of whether the loop's condition is True or False.

Example 1: Using break to terminate a loop

```
count = 1
while count <= 5: # Loop continues as long as count <= 5
    print("Count:", count)
    if count == 3: # Check if count is 3
        print("Breaking the loop at count =", count)
        break # Exit the loop
    count += 1 # Increment count
print("Loop ended.")
```

Code Explanation:

1. The loop begins with the condition count <= 5.
2. The value of count is printed at each iteration.
3. When count reaches 3, the if statement becomes True, and the "break statement" is executed.
4. The loop exits immediately, even though the condition count <= 5 is still True.
5. Finally, the message "Loop ended." is printed outside the loop.

Output:

Copy code

Count: 1

Count: 2

Count: 3

Breaking the loop at count = 3

Loop ended.

Using the 'Continue Statement' in a While Loop

Use the continue statement to skip the current loop iteration and go directly to the next one. The loop does not end even when the remaining code for the current iteration is skipped.

NOTE

Example 2: Using continue to bypass certain values

```
count = 0

while count < 5:
    count += 1 # Increment count
    if count == 3: # Check if count equals 3
        print("Skipping count =", count)
        continue # Skip this iteration and proceed to the next one
    print("Count:", count)

print("Loop completed.")
```

Code Explanation:

1. The loop runs while count < 5.
2. At the beginning of the loop, count is increased.
3. The continue statement is carried out and the if condition is made True when count equals 3.
4. The continue statement skips the print("Count:", count) line and moves to the next iteration.
5. For all other values of count, they are printed as usual.

Output:

```
Count: 1
Count: 2
Skipping count = 3
Count: 4
Count: 5
Loop completed.
```

Differences Between break and continue

Break	Continue
Exits the loop completely.	The current iteration is skipped.
The loop terminates.	The loop moves on to the following iteration.
No further iterations happen.	Remaining iterations continue as usual.

Example 3: Combining break and continue in a While Loop

```
count = 0

while count < 7:
    count += 1 # Increment count
```

NOTE

```

if count == 3: # Skip printing 3
    print("Skipping count =", count)
    continue

if count == 5: # Stop the loop at 5
    print("Breaking the loop at count =", count)
    break

print("Count:", count)

print("Loop finished.")

```

Code Explanation:

1. The loop runs as long as `count < 7`.
2. The `continue` statement skips the printing of `count = 3` and proceeds to the 'next iteration.'
3. The 'break statement' terminates the 'loop' when `count = 5`.
4. After exiting the loop, the program prints "Loop finished."

Output:

```

Count: 1
Count: 2
Skipping count = 3
Count: 4
Breaking the loop at count = 5
Loop finished.

```

8.4 FOR (BREAK, CONTINUE, RANGE, NESTED)

The 'for Loop' in Python

To iterate over a sequence (such as a list, tuple, string, or range), use a for loop, which repeats a block of code for each item in the series.

Syntax of for loop:

for the variable in the sequence:

Code to run for every iteration Here:

- The **variable** takes each value from the sequence, one by one.
- The loop executes as long as there are items in the sequence.

1. 'break' in a 'for Loop'

The "**break statement**" is used to "**exit the loop immediately**", even if there are more items in the sequence.

Example 1: Using break to stop the loop

```
for number in range(1, 6): # Iterates through numbers 1 to 5
```

NOTE

```
print("No:", number)
if number == 3: # Condition to stop the loop
    print("Breaking the loop at no =", number)
    break # Exit the loop
print("Loop ended.")
```

Explanation of Code:

1. The loop iterates through the numbers 1 to 5 (using range).
2. The loop is terminated by breaking the if condition when the number equals 3.
3. The code after the loop ("Loop ended.") executes.

Output:

```
No: 1
No: 2
No: 3
Breaking the loop at no = 3
Loop ended.
```

2. continue in a for Loop

The continue statement advances to the following iteration by avoiding the current one.

Example 2: Using continue to skip values

```
for number in range(1, 6): # Iterates through numbers 1 to 5
    if number == 3: # Condition to skip
        print("Skipping number =", number)
        continue # In this iteration, skip the remaining code.
    print("No:", number)
print("Loop completed.")
```

Explanation of Code:

1. The loop iterates through numbers 1 to 5.
2. When number equals 3, the continue statement skips the print("Number:", number) line.
3. The next iteration is reached via the loop.

Output:

```
No: 1
No: 2
Skipping number = 3
No: 4
No: 5
Loop completed.
```

NOTE**3. range() Function**

The "range()" technique is frequently used with for loops and produces a sequence of numbers.

Syntax:

"range(start, stop, step)"

- **"start"**: Start value (optional, default is 0).
- **"stop"**: Stopping value (exclusive).
- **"step"**: Increment (optional, default is 1).

Example 3: Using range() with for loop

```
# Print even numbers from 2 to 10
for number in range(2, 11, 2): # Start=2, Stop=11, Step=2
    print("Even num:", number)
```

Explanation of Code:

1. range(2, 11, 2) generates the numbers: 2, 4, 6, 8, 10.
2. The for loop iterates through these numbers, and each value is printed.

Output:

```
"Even num: 2"
"Even num: 4"
"Even num: 6"
"Even num: 8"
"Even num: 10"
```

4. Nested "for Loops"

Every time the outer loop iterates, the inner loop continues until the end, forming a nested loop.

Example 4: Using nested for loops to print a pattern

```
for i in range(1, 4): # Outer loop (rows)
    for j in range(1, 4): # Inner loop (columns)
        print(f'({i}, {j})', end=" ") # Print coordinates
    print() # New line after each row
```

Explanation of Code:

1. The outer loop (i) iterates from 1 to 3 (3 rows).
2. The inner loop (j) iterates from 1 to 3 for each value of i.
3. For each pair (i, j), the coordinates are printed.
4. To keep the output on the same line, use end=" ".
5. After the inner loop advances to the following line, print() is called.

NOTE

Output:

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

5. Combining break, continue, and Nested Loops

Example 5: Break and Continue in Nested Loops

```
for i in range(1, 4): # Outer loop
    for j in range(1, 4): # Inner loop
        if j == 2: # Skip inner loop iteration when j=2
            print("Skipping inner loop at j =", j)
            continue
        if i == 3: # Break outer loop when i=3
            print("Breaking outer loop at i =", i)
            break
        print(f'i={i}, j={j}')
    if i == 3: # Ensures outer loop breaks after the condition
        break
```

Explanation of Code:

1. The outer loop runs through i values 1 to 3.
2. The inner loop runs through j values 1 to 3.
3. When j == 2, Continue bypasses the inner loop's current iteration.
4. When i == 3, break exits both loops.

Output:

```
i=1, j=1
Skipping inner loop at j = 2
i=1, j=3
i=2, j=1
Skipping inner loop at j = 2
i=2, j=3
Skipping inner loop at j = 2
Breaking outer loop at i = 3
```



Activity 8.1

1. Objective:

- To understand how break and continue affect loop execution.
- To practice writing while and for loops with break and continue.

2. Instructions:

- Write a Python program that prints numbers from 1 to 10 but stops the loop when it reaches 6 using the break statement.

NOTE

- Modify the program to skip number 6 using the continue statement instead of break.

Materials Needed:

- Computer with Python installed
- Python editor (IDLE, VS Code, or Jupyter Notebook)

.....

.....

.....

.....

☒ Check Your Progress 8.1

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

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

```
for i in range(1, 6):
    if i == 3:
        break
    print(i)
```

.....

.....

.....

.....

2. What happens when the continue statement is used inside a loop?

.....

.....

.....

.....

8.5 SUMMARY

Loops in Python help automate repetitive tasks efficiently. The while loop runs until its condition becomes False, whereas the for loop iterates over sequences like lists and ranges. The break statement allows exiting a loop early, while continue skips the current iteration without stopping the loop. The range() function generates sequences of numbers for iteration, and **nested loops** allow a loop inside another loop for handling complex patterns. Mastering loops and control statements improves efficiency in writing Python programs.

8.6 GLOSSARY

- **While Loop (while)** – Executes a block of code as long as a condition remains True.
- **For Loop (for)** – Iterates over a sequence (list, tuple, string, range).

NOTE

- **Break Statement (break)** – Exits a loop before it completes all iterations.
- **Continue Statement (continue)** – Skips the current iteration and moves to the next.
- **Range (range())** – Generates a sequence of numbers, commonly used in for loops.
- **Nested Loops** – A loop inside another loop, used for multi-level iteration.
- **Iteration** – The process of executing a loop's code repeatedly.
- **Loop Control Statements** – Special statements (break, continue) that modify loop execution flow.

8.7 ANSWERS TO CHECK YOUR PROGRESS.

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

```
for i in range(1, 6):
    if i == 3:
        break
    print(i)
```

Answer:

The output will be:

1
2

Since break is encountered when i == 3, the loop stops immediately.

2. What happens when the continue statement is used inside a loop?

Answer:

When continue is used inside a loop, it **skips the current iteration** and moves to the next one. The loop does not stop but simply ignores the remaining code in that iteration.

Example:

```
for i in range(1, 6):
    if i == 3:
        continue
    print(i)
```

Output:

CopyEdit

1
2
4
5

Here, when i == 3, continue skips the print statement and moves to i == 4.

NOTE**8.8 SUGGESTED READINGS - OER**

Sr. No.	Title	URL	QR Code
1	Open Educational Resource	http://books.aisectionline.com/UnitUrls?unitId=1283	

8.9 REFERENCE

- "Python Crash Course" – Eric Matthes
- "Learning Python" – Mark Lutz
- "Automate the Boring Stuff with Python" – Al Sweiga

8.10 EXERCISE**Part 1****A. Multiple Choice Questions (MCQs)**

- What will happen when the break statement is used inside a loop?**
 - It continues to the next iteration
 - It terminates the loop immediately
 - It skips the next two iterations
 - It restarts the loop
- Which of the following correctly defines a for loop with a range function?**
 - for i in range(1, 5):
 - for i in (1, 5):
 - for range(1, 5) in i:
 - for i = range(1, 5):
- What does the continue statement do in a loop?**
 - Terminates the loop
 - Skips the current iteration and moves to the next one
 - Breaks out of the loop completely
 - Repeats the same iteration
- What will be the output of the following code?**

```
for i in range(3):
    for j in range(2):
```

NOTE

print(i, j)

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

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

i = 0

while i < 5:

i += 1

if i == 3:

continue

print(i)

- a) 1, 2, 3, 4, 5
- b) 1, 2, 4, 5
- c) 1, 2, 3, 5
- d) 1, 3, 4, 5

Answers with Explanations:

1. **b) It terminates the loop immediately** – break stops the loop at once and control moves outside the loop.
2. **a) for i in range(1, 5):** - This is the correct syntax to loop from 1 to 4 using range().
3. **b) Skips the current iteration and moves to the next one** - continue skips the remaining code in the current loop iteration.
4. **a) 0 0, 0 1, 1 0, 1 1, 2 0, 2 1** - The outer loop runs 3 times and the inner loop runs 2 times for each value.
5. **b) 1, 2, 4, 5** - When i becomes 3, continue skips printing that value.

B. Short Answer Type Questions

1. Explain the difference between break and continue in Python loops.
2. How does the range() function work in a for loop? Give an example.
3. What are nested loops, and how do they work?

Part 2

C. Long Answer Type Questions (200 words)

1. Explain the working of while loops and for loops in Python. Discuss how break and continue statements affect loop execution with examples.

NOTE

2. Describe nested loops in Python with examples. How are they useful in real-world programming scenarios?

Part 3

D. Long Answer Type Questions (300 words)

1. Discuss the use of loops in Python. Explain ``for`` and ``while`` loops with syntax, examples, and real-life applications.
2. Explain nested loops and their importance. Provide programs that demonstrate patterns, matrices, and multi-level iteration.
3. Describe loop control statements (``break``, ``continue``, ``pass``). Explain how they modify loop behavior with examples.

UNIT 9

FUNCTIONS & LAMBDA, ARRAYS

NOTE

UNIT STRUCTURE

- 9.1 Introduction
- 9.2 Objective
- 9.3 Functions (Creating, Calling Parameters Passing, Keyword Arguments, Recursion)
- 9.4 Lambda
- 9.5 Arrays
- 9.6 Summary
- 9.7 Glossary
- 9.8 Answers to check your progress
- 9.9 Suggested readings - OER
- 9.10 Reference
- 9.11 Exercise
- 9.12 Feedback form

9.1 INTRODUCTION

Functions in Python are reusable blocks of code designed to perform specific tasks. They help organize code, reduce redundancy, and improve readability. Python allows users to **create** functions using the `def` keyword and **call** them when needed. Functions accept **parameters** to pass values and return results. Python supports **keyword arguments**, allowing arguments to be passed with explicit names for better clarity. **Recursion** is a technique where a function calls itself to solve complex problems. Additionally, **lambda functions** (also called anonymous functions) are single-line functions used for short, simple operations without a formal `def` declaration.

9.2 OBJECTIVE

After completing this unit, you will be capable of:

- To understand how to create and call functions in Python.
- To explore different ways of passing parameters into functions.
- To learn about keyword arguments for better function clarity.
- To understand recursion and its role in solving complex problems.
- To use lambda functions for concise and efficient coding.

NOTE**9.3 FUNCTIONS (CREATING, CALLING PARAMETERS PASSING, KEYWORD ARGUMENTS, RECURSION)****Functions in Python**

A chunk of reusable code that carries out a particular purpose is called a function. By breaking a program up into smaller, more manageable components, functions enhance readability and code reuse.

Syntax of a Function

```
def function_name(parameters):
```

```
    # Code block
```

```
    return value
```

- **def:** A function is defined by a keyword.
- **function_name:** The function's name.
- **parameters:** Optional inputs sent to the function.
- **return:** An optional method for returning a value.

Creating and Calling Functions**Example 1: Creating and Calling a Function**

```
# Function to greet a user
```

```
def greet():
```

```
    print("Hello! Welcome to Python.")
```

```
# Calling the function
```

```
greet()
```

Explanation of Code:

1. The function `greet()` is defined using `def`. It contains a `print` statement.
2. To execute the function, we call it using `greet()`.

Output:

```
Hello! Welcome to Python.
```

Passing Parameters to a Function

Parameters are variables passed to a function. They allow us to pass data and make the function dynamic.

Example 2: Passing Parameters

```
# Function to add two numbers
```

```
def add_numbers(a, b):
```

```
    result = a + b
```

```
    print("Sum:", result)
```

```
# Calling the function with arguments
```

```
add_numbers(5, 3)
```

Explanation of Code:

1. a and b are parameters of the add_numbers function.
2. The function takes two arguments (5 and 3), adds them, and prints the result.

Output:

Sum: 8

Keyword Arguments

The sequence of arguments is unimportant when you use keyword arguments to pass arguments using the parameter name.

Example 3: Using Keyword Arguments

```
# Function to display user details
def display_user(name, age):
    print(f'Name: {name}')
    print(f'Age: {age}')

# Calling the function with keyword arguments
display_user(age=25, name="John")
```

Explanation of Code:

1. The function defines the arguments name and age.
2. Name="John" and age="25" are supplied as keyword parameters when the function is called.
3. As long as parameter names are provided, the argument order is irrelevant.

Output:

Name: John
Age: 25

Default Parameters

A default parameter has a default value, which is used if no argument is provided.

Example 4: Default Parameters

```
# Function to greet a user with default name
def greet_user(name="Guest"):
    print("Hello,", name)

# Calling the function
greet_user("Alice") # Passing argument
greet_user() # Without argument
```

Explanation of Code:

1. The parameter name has a default value of "Guest".
2. If no argument is passed, the default value is used.

NOTE

NOTE**Output:**

Hello, Alice
Hello, Guest

6. Recursion

When a function calls itself, recursion takes place. It is frequently employed to resolve issues that can be divided into more manageable subissues.

Example 5: Factorial using Recursion

Function to calculate factorial using recursion

```
def factorial(n):
```

```
    if n == 0: # Base case
```

```
        return 1
```

```
    else:
```

```
        return n * factorial(n - 1) # Recursive call
```

```
# Calling the function
```

```
result = factorial(5)
```

```
print("Factorial of 5 is:", result)
```

Explanation of Code:

1. The factorial function calculates the factorial of n.
2. **Base case:** If $n == 0$, return 1 (factorial of 0 is 1).
3. **Recursive case:** The function calls itself with $n-1$.
4. When n reaches 0, the recursion stops.

Execution Steps:

- $\text{factorial}(5) \rightarrow 5 * \text{factorial}(4)$
- $\text{factorial}(4) \rightarrow 4 * \text{factorial}(3)$
- $\text{factorial}(3) \rightarrow 3 * \text{factorial}(2)$
- $\text{factorial}(2) \rightarrow 2 * \text{factorial}(1)$
- $\text{factorial}(1) \rightarrow 1 * \text{factorial}(0) \rightarrow 1$

Final result: $5 \times 4 \times 3 \times 2 \times 1 = 120$
 $= 1205 \times 4 \times 3 \times 2 \times 1 = 120$.

Output:

Factorial of 5 is: 120

9.4 LAMBDA

Lambda Functions in Python

In Python, a lambda function is a compact, anonymous function defined without the def keyword. These functions are ideal for performing simple tasks where defining a full function would be redundant. A lambda function can take multiple arguments but can only contain a single expression.

NOTE**What is a Lambda Function?**

A lambda function is defined using the lambda keyword, followed by the parameters and an expression. The expression is evaluated and returned when the lambda function is invoked.

Syntax:

lambda arguments: expression

- **arguments:** These are the input parameters, just like in a regular function.
- **expression:** A single expression that is evaluated and returned automatically, without using the return keyword.

1. Defining a Lambda Function

A lambda function can be defined inline and used immediately.

Example 1: A Lambda Function to Add Two Numbers

Lambda function to add two numbers

```
add = lambda x, y: x + y
```

Calling the lambda function

```
result = add(5, 3)
```

```
print("Sum:", result)
```

Explanation:

1. The lambda function lambda x, y: x + y adds the two arguments x and y.
2. It is assigned to the variable add.
3. The function is called with the values 5 and 3.
4. The sum (8) is printed.

Output:

Sum: 8

2. Lambda Function Without Assigning to a Variable

Lambda functions can be invoked directly without assigning them to a variable.

Example 2: Inline Lambda Function

Directly calling a lambda function

```
print("Product:", (lambda x, y: x * y)(4, 5))
```

Explanation:

1. The lambda function lambda x, y: x * y computes the product of x and y.
2. It is immediately invoked with the arguments 4 and 5.
3. The product (20) is printed.

Output:

Product: 20

NOTE**3. Using Lambda with the filter() Function**

The filter() function filters elements from a list based on a specified condition. Lambda functions are commonly used within filter() to define the filtering criteria.

Example 3: Filtering Even Numbers from a List

```
# List of numbers
numbers = [1, 2, 3, 4, 5, 6]

# Use filter() with a lambda function
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))

print("Even numbers:", even_numbers)
```

Explanation:

1. The filter() function applies the lambda function lambda x: x % 2 == 0 to each element in the list to check if the number is even.
2. The filtered list is converted to a list and printed.

Output:

Even numbers: [2, 4, 6]

4. Using Lambda with the map() Function

The map() function applies a given function to every item in a sequence. Lambda functions are often used with map() for simple transformations.

Example 4: Squaring Each Number in a List

```
# List of numbers
numbers = [1, 2, 3, 4, 5]

# Use map() with a lambda function
squared_numbers = list(map(lambda x: x ** 2, numbers))

print("Squared numbers:", squared_numbers)
```

Explanation:

1. The map() function applies the lambda function lambda x: x ** 2 to each element in the list to square the numbers.
2. The result is converted to a list and printed.

Output:

Squared numbers: [1, 4, 9, 16, 25]

5. Using Lambda with the reduce() Function

The reduce() function (from the functools module) reduces a sequence to a single value by applying a function cumulatively.

Example 5: Calculating the Product of a List

```
from functools import reduce

# List of numbers
numbers = [1, 2, 3, 4]
```

NOTE

```
# Use reduce() with a lambda function
product = reduce(lambda x, y: x * y, numbers)
print("Product of all numbers:", product)
```

Explanation:

1. The reduce() function applies the lambda function lambda x, y: x * y to each element of the list cumulatively.
2. The final product (24) is printed.

Output:

Product of all numbers: 24

6. Lambda Functions with Multiple Arguments

Lambda functions can accept multiple arguments and perform operations inline.

Example 6: Finding the Largest of Three Numbers

```
# Lambda function to find the largest of three numbers
largest = lambda x, y, z: max(x, y, z)
```

```
# Calling the lambda function
result = largest(10, 25, 15)
print("Largest number:", result)
```

Explanation:

1. The lambda function lambda x, y, z: max(x, y, z) finds the largest value among the three arguments.
2. The result (25) is printed.

Output:

Largest number: 25

Advantages of Lambda Functions

1. **Concise:** Lambda functions are written in a single line, making the code more compact.
2. **Anonymous:** They are unnamed, ideal for temporary operations.
3. **Versatile:** Easily used with functions like map(), filter(), and reduce().

Limitations of Lambda Functions

1. **Single Expression:** Lambda functions can only contain a single expression.
2. **Reduced Readability:** For complex operations, lambda functions can be harder to read.
3. **Limited Functionality:** They do not support statements such as loops or multiple lines.

NOTE**9.5 ARRAYS (ELEMENT ACCESSING AND ADDING, LOOPING, LENGTH, METHODS).****Arrays in Python**

An **array** is a collection of elements of the same data type stored at contiguous memory locations. Arrays are useful for storing and managing large datasets where operations like indexing, adding, looping, and accessing elements are common.

In Python, arrays are not built into the language by default but can be created using the **array** module or **NumPy** library. Lists, however, can serve similar purposes as arrays because they support dynamic size and multiple data types.

Definition of an Array

A data structure called an array holds several identically typed elements in a single variable. Arrays prioritize type consistency and memory efficiency over Python lists, which can store a variety of data types.

Python arrays require the import of the array module in order to be used:

```
import array
```

Creating an Array

Arrays are created using the `array.array()` function. The syntax is:

```
array(typecode, [elements])
```

- **typecode:** 'i' for integers and 'f' for floats are examples of characters that indicate the data type of array items.
- **elements:** A list of initial elements for the array.

Example 1: Creating an Array

```
import array

# Creating an integer array
arr = array.array('i', [1, 2, 3, 4, 5])
print("Array elements:", arr)
```

Explanation:

1. The array module is imported.
2. An array `arr` is created with the typecode 'i' (integer).
3. The array elements `[1, 2, 3, 4, 5]` are initialized and displayed.

Output:

```
Array elements: array('i', [1, 2, 3, 4, 5])
```

Accessing Elements in an Array

Array elements can be accessed using **indexing**. Indexing starts at 0 for the first element.

NOTE**Example 2: Accessing Array Elements**

```
import array
# Creating an array
arr = array.array('i', [10, 20, 30, 40, 50])

# Accessing elements using indices
print("First element:", arr[0]) # Accessing first element
print("Third element:", arr[2]) # Accessing third element
```

Explanation:

1. The array `arr` is created with integers.
2. `arr[0]` accesses the first element (10).
3. `arr[2]` accesses the third element (30).

Output:

```
First element: 10
Third element: 30
```

Adding Elements to an Array

Inserting Elements into an Array Elements can be inserted into an array using the following methods:

- **append()**: Appends a single element to the end of the array.
- **extend()**: Adds multiple elements from an iterable to the array.
- **insert()**: Inserts an element at a specified index in the array.

Example 3: Adding Elements

```
import array

# Creating an array
arr = array.array('i', [1, 2, 3])

# Adding elements
arr.append(4) # Append a single element
arr.extend([5, 6]) # Extend with multiple elements
arr.insert(1, 10) # Insert 10 at index 1

print("Updated array:", arr)
```

Explanation:

1. `append(4)` adds 4 at the end.
2. `extend([5, 6])` adds multiple elements [5, 6] at the end.
3. `insert(1, 10)` inserts 10 at index 1.

Output:

```
Updated array: array('i', [1, 10, 2, 3, 4, 5, 6])
```

Looping Through an Array

You can iterate through array elements using loops, such as for loops.

NOTE**Example 4: Looping Through an Array**

```
import array

# Creating an array
arr = array.array('i', [10, 20, 30, 40])

# Looping through the array
print("Array elements:")
for element in arr:
    print(element)
```

Explanation:

1. A for loop is used to iterate through arr.
2. Each element is printed on a new line.

Output:

```
Array elements:
10
20
30
40
```

5. Length of an Array

The len() function is used to determine the total number of elements in an array.

Example 5: Finding the Length of an Array

```
import array

# Creating an array
arr = array.array('i', [1, 2, 3, 4, 5, 6])

# Finding the length of the array
print("Length of the array:", len(arr))
```

Explanation:

1. The len() function returns the number of elements in the array.
2. Here, the array contains 6 elements.

Output:

```
Length of the array: 6
```

6. Array Methods

Below are some frequently utilized array methods:

Method	Description
append(x)	Adds the element x to the end of the array.
extend(iterable)	Adds all elements from the given iterable.
insert(i, x)	Inserts the element x at the specified index i.

pop(i)	Removes and returns the element at index i.
remove(x)	Deletes the first occurrence of element x.
index(x)	Returns the index of the element x.
reverse()	Reverses the order of elements in the array.

NOTE**Example 6: Using Array Methods**

```
import array

# Creating an array
arr = array.array('i', [1, 2, 3, 4, 5])

# Using methods
arr.pop(2) # Removes element at index 2
arr.remove(4) # Removes the first occurrence of 4
arr.reverse() # Reverses the array

print("Modified array:", arr)
```

Explanation:

1. The pop(2) method removes the element located at index 2, which is the third element.
2. remove(4) removes the first occurrence of 4.
3. reverse() reverses the array.

Output:

Modified array: array('i', [5, 2, 1])

**Activity 9.1****1. Understanding Function Creation and Calling**

- Create a Python function named greet() that takes a name as an argument and prints "Hello, [name]!".

2. Exploring Lambda Functions

- Write a lambda function to multiply a number by 5 and test it with different values.

Materials Needed:

- Computer with Python installed
- Python IDE or text editor (e.g., VS Code, PyCharm, Jupyter Notebook)
- Reference materials (Python documentation, books, or online tutorials)

.....

.....

.....

.....

NOTE**☑ Check Your Progress 9.1**

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

1. What is the key difference between a normal function and a lambda function in Python?

.....

2. How does recursion work in Python? Provide an example.

.....

9.6 SUMMARY

Functions in Python allow modular and reusable code by defining tasks that can be executed multiple times. They accept **parameters** to work with dynamic data and can return results. **Keyword arguments** enable explicit naming of parameters, improving readability. **Recursion** allows functions to call themselves for problem-solving, commonly used in mathematical computations like factorials and Fibonacci series. **Lambda functions** provide a compact way to define small, one-time-use functions. Mastering these concepts helps in writing efficient and structured Python programs.

9.7 GLOSSARY

- **Function (def)** – A block of reusable code defined using the def keyword.
- **Calling a Function** – Executing a function using its name followed by parentheses (function_name()).
- **Parameter** – A variable defined in a function to receive input values.
- **Argument** – The actual value passed to a function when calling it.
- **Keyword Argument** – An argument passed with an explicit parameter name (func(name="Alice")).
- **Recursion** – A function calling itself to solve a problem iteratively.
- **Lambda Function (lambda)** – A small, anonymous function written in one line (lambda x: x * 2).
- **Return Statement (return)** – Used to send back a result from a function.

9.8 ANSWERS TO CHECK YOUR PROGRESS

NOTE

1. What is the key difference between a normal function and a lambda function in Python?

Answer: A normal function in Python is defined using the `def` keyword and can contain multiple expressions. It requires a function name and an explicit return statement if returning a value.

Example:

python

CopyEdit

```
def square(x):
    return x * x
```

A lambda function, on the other hand, is an **anonymous, single-expression function** defined using the `lambda` keyword. It does not require a function name or return.

Example:

```
square = lambda x: x * x
```

Lambda functions are useful for short, one-time operations where defining a full function may not be necessary.

2. How does recursion work in Python? Provide an example.

Answer: Recursion occurs when a function calls itself within its definition. It is commonly used for problems that can be broken into smaller subproblems, such as calculating factorials or Fibonacci sequences.

Example of a recursive function to calculate factorial:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

```
print(factorial(5)) # Output: 120
```

In this example, `factorial(5)` calls `factorial(4)`, which calls `factorial(3)`, and so on until `factorial(0)`, which returns 1, stopping the recursion.

9.9 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource	http://books.aisectionline.com/UnitUrls?unitId=1284	

NOTE**9.10 REFERENCE**

- **Python Crash Course** by Eric Matthes
- **Automate the Boring Stuff with Python** by Al Sweigart
- **Python Programming: An Introduction to Computer Science** by John Zelle
- **Python for Data Analysis** by Wes McKinney

9.11 EXERCISE**Part 1****A. Multiple Choice Questions (MCQs)****1. Which keyword is used to define a function in Python?**

- a) function
- b) def
- c) func
- d) define

2. Which statement is true about lambda functions?

- a) Lambda functions can have multiple expressions
- b) Lambda functions require the return keyword
- c) Lambda functions are anonymous functions with a single expression
- d) Lambda functions cannot take arguments

3. How do you call a function named my_function in Python?

- a) call my_function()
- b) my_function()
- c) run my_function()
- d) execute(my_function)

4. What is recursion in Python?

- a) A function calling another function
- b) A function calling itself
- c) A function with multiple return statements
- d) A function that only works inside loops

5. Which module is used to work with arrays in Python?

- a) math
- b) os
- c) array
- d) sys

Answers with Explanations:

1. **b) def** – The def keyword is used to define a function in Python.
2. **c) Lambda functions are anonymous functions with a single expression** - Lambda functions are unnamed and can contain only one expression.
3. **b) my_function()** - A function is called by writing its name followed by parentheses.
4. **b) A function calling itself** - Recursion occurs when a function calls itself to solve a problem.
5. **c) array** - The array module is used to create and work with array data structures.

NOTE**B. Short Answer Type Questions**

1. What are the differences between normal functions and lambda functions in Python?
2. How does parameter passing work in Python functions? Explain with examples.
3. What are the advantages and disadvantages of recursion?

Part 2**C. Long Answer Type Questions (200 words)**

1. Explain the concept of function parameters, including positional arguments, keyword arguments, and default values, with appropriate examples.
2. Describe arrays in Python, including how they are different from lists. Provide examples of array creation, manipulation, and accessing elements.

Part 3**D. Long Answer Type Questions (200 words)**

1. Explain the concept of functions in Python. Discuss defining functions, returning values, and the importance of code reusability.
2. Describe lambda (anonymous) functions. Discuss how they differ from normal functions and provide examples of their usage.
3. Explain Python arrays and how they differ from lists. Discuss the need for the `array` module and common array operations.



DR. C.V. RAMAN UNIVERSITY

Chhattisgarh, Bilaspur AN A/SECT GROUP UNIVERSITY

Approved by: PCI | AICTE | NISTE | BCI | Member of: AN | Joint Committee: UGC | DSR | AICTE | Recognized by: BGC | A MAC Accredited University



DR. C. V. RAMAN UNIVERSITY (C.G.)

SLM FEEDBACK FORM

PROGRAMME : _____

COURSE : _____

BLOCK : _____

UNIT : _____

Dear Student,

While studying the units of this block, you may have found certain portions of the text difficult to comprehend. We wish to know your difficulties and suggestions, in order to improve the course. Therefore, we request you to fill up and send us the following questionnaire, which pertains to this block. If you find the space provided insufficient, kindly use a separate sheet.

Please mail to aisectcontent@aisect.org/ for online submission please scan QR code:

Questionnaire

1. Name : _____

2. Enrolment No. :

--	--	--	--	--	--	--	--	--	--	--

3. How many hours did you need for studying the units? :

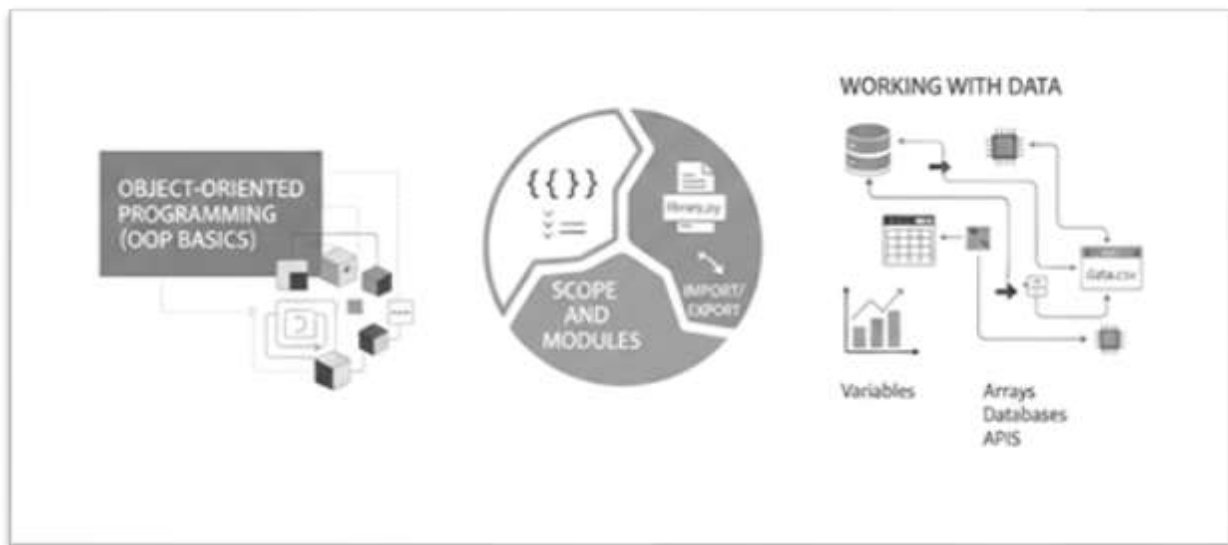
--

4. Please give your reactions to the following items based on your reading of the block:

Items	Excellent	Very Good	Good	Poor	Give specific Examples (If poor)
Presentation Quality					
Language and Style					
Illustrations					
Used (diagrams, tables, etc.)					
Conceptual Clarity					
Check Your Progress Questions					
Suggested Readings - OER					

5. Any other comments:

BLOCK IV	CLASSES CONCEPTS	103-150
Unit 10	Object-Oriented Programming (OOP) Basics	105
Unit 11	Scope and Modules	121
Unit 12	Working with Data	133



This block introduces Object-Oriented Programming (OOP), a widely used programming paradigm that makes software more organized, modular, and scalable. Students learn how to design and use classes and objects to model real-world entities, which enhances the way programs are structured. The block explains essential OOP principles, variable scope rules, and how Python modules allow programmers to split their code into manageable components. Additionally, students learn how Python handles different data within classes and modules, helping them build well-structured applications. After completing this block, students will be able to design object-oriented solutions and write programs that are easier to maintain, expand, and reuse.

UNIT 10

OBJECT-ORIENTED PROGRAMMING (OOP) BASICS

NOTE

UNIT STRUCTURE

- 10.1 Introduction
- 10.2 Objective
- 10.3 Classes/Objects (Create, Delete, Modify)
- 10.4 Inheritance
- 10.4 Iterators
- 10.5 Summary
- 10.6 Glossary
- 10.7 Answers to check your progress
- 10.8 Suggested readings - OER
- 10.9 Reference
- 10.10 Exercise

10.1 INTRODUCTION

Object-oriented programming (OOP) in Python revolves around **classes** and **objects**, allowing for structured and reusable code. A **class** is a blueprint for creating objects, while an **object** is an instance of a class. Python allows users to **create**, **modify**, and **delete** objects dynamically. **Inheritance** enables a class to derive properties and behaviors from another class, promoting code reuse. **Iterators** are objects that implement the `__iter__()` and `__next__()` methods, allowing sequential access to elements, commonly used in loops and iterable data structures like lists and tuples.

10.2 OBJECTIVE.

After completing this unit, you will be capable of:

- To understand how to create, modify, and delete classes and objects in Python.
- To explore **inheritance** and its role in code reusability.
- To learn about **iterators** and how they enable sequential access to elements.
- To apply OOP concepts to improve code organization and maintainability.

10.3 CLASSES/OBJECTS (CREATE, DELETE, MODIFY)

Classes and Objects in Python

Definition:

A class in Python is a template used to create objects, which are instances of the class. Classes contain both functions (methods) that

NOTE

manipulate the data and data (attributes). Objects are particular instances of a class that adhere to the structure specified by the class while storing their own distinct data.

Creating Classes and Objects**Example Code:**

```
# Define a class
class Car:
    # Constructor to initialize attributes
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    # Method to display car details
    def display_details(self):
        return f'{self.year} {self.make} {self.model}'

# Create an object (instance of the class)
my_car = Car("Toyota", "Camry", 2020)

# Access object attributes and methods
print(my_car.display_details())
```

Step-by-Step Explanation:**1. Class Definition:**

The class keyword is used to define the Car class. Each object produced has its attributes (make, model, and year) initialized by the constructor, a unique `__init__` method.

2. Object Creation:

The `my_car` object is created by calling `Car("Toyota", "Camry", 2020)`. This initializes the attributes `make`, `model`, and `year` for the `my_car` instance.

3. Accessing Attributes and Methods:

The `display_details()` method is called on the `my_car` object, returning the string representation of the car's details.

Modifying Object Attributes**Example Code:**

```
# Modify an object's attribute
my_car.year = 2021
print(my_car.display_details()) # Updated year
```

Step-by-Step Explanation:**1. Direct Modification:**

The `year` attribute of the `my_car` object is updated from 2020 to 2021 using `my_car.year = 2021`.

2. Result:

Calling `display_details()` again reflects the updated year attribute.

Deleting Object Attributes and Objects**Example Code:**

```
# Delete an object's attribute
del my_car.year

# Try to access the deleted attribute
try:
    print(my_car.year)
except AttributeError as e:
    print("Error:", e)

# Delete the object
del my_car

# Try to access the deleted object
try:
    print(my_car)
except NameError as e:
    print("Error:", e)
```

Step-by-Step Explanation:**1. Deleting an Attribute:**

The `del` keyword removes the `year` attribute from the `my_car` object. Attempting to access `my_car.year` afterward raises an `AttributeError`.

2. Deleting an Object:

The `del` keyword deletes the entire `my_car` object. Any further reference to `my_car` raises a `NameError`.

NOTE

10.4 INHERITANCE

Inheritance in Python

Definition: Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a class, known as the child or derived class, to inherit properties and methods from another class, referred to as the parent or base class. This helps in reusing code and creating a class hierarchy with shared behaviors.

Understanding Inheritance: In Python, a child class is defined by passing the parent class as an argument during class creation. The child class inherits all the attributes and methods of the parent class and can introduce its own unique attributes or methods, or even override existing ones.

NOTE**Basic Example of Inheritance**

```
# Parent Class
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return f'{self.name} makes a sound.'

# Child Class
class Dog(Animal):
    def speak(self):
        return f'{self.name} barks.'

# Create instances
generic_animal = Animal("Generic Animal")
dog = Dog("Buddy")

# Access methods
print(generic_animal.speak()) # Output: Generic Animal makes a
                              # sound.
print(dog.speak())           # Output: Buddy barks.
```

Step-by-Step Explanation:**1. Parent Class Definition:**

The Animal class includes a constructor (`__init__`) to initialize the name attribute and a method `speak()`.

2. Child Class Definition:

The Dog class inherits from Animal and overrides the `speak()` method to specify behavior for dogs.

3. Creating Instances:

The `generic_animal` instance uses the `speak()` method from the parent class, while the `dog` instance uses the overridden method from the child class.

Adding New Methods and Attributes in the Child Class

```
# Child Class with additional method
class Cat(Animal):
    def __init__(self, name, color):
        super().__init__(name) # Call parent class constructor
        self.color = color

    def description(self):
        return f'{self.name} is a {self.color} cat.'

# Create instance of Cat
cat = Cat("Whiskers", "white")
print(cat.description()) # Output: Whiskers is a white cat.
print(cat.speak())       # Output: Whiskers makes a sound.
```

Step-by-Step Explanation:**1. Extending the Constructor:**

The Cat class introduces a new attribute, color. The `super().__init__(name)` invokes the parent class constructor to initialize the name attribute.

2. New Method:

The `description()` method is unique to the Cat class, adding additional functionality.

3. Result:

The Cat class instance can access both inherited (`speak()`) and newly defined methods (`description()`).

Multiple Inheritance

Python allows multiple inheritance, where a child class can inherit from multiple parent classes.

```
class Flyer:
```

```
    def fly(self):
        return "This animal can fly."
```

```
class Swimmer:
```

```
    def swim(self):
        return "This animal can swim."
```

```
# Child Class inheriting from multiple parents
```

```
class Duck(Flyer, Swimmer):
```

```
    pass
```

```
# Create instance of Duck
```

```
duck = Duck()
```

```
print(duck.fly()) # Output: This animal can fly.
```

```
print(duck.swim()) # Output: This animal can swim.
```

Step-by-Step Explanation:**1. Parent Classes:**

Flyer and Swimmer define the methods `fly()` and `swim()`, respectively.

2. Child Class:

The Duck class inherits from both Flyer and Swimmer, gaining access to both methods.

3. Result:

The Duck class instance can perform actions defined in both parent classes.

Overriding Methods in Multiple Inheritance

When parent classes have methods with the same name, Python uses the Method Resolution Order (MRO) to decide which method to invoke.

```
class Parent1:
```

NOTE

NOTE

```

def greet(self):
    return "Hello from Parent1!"

class Parent2:
    def greet(self):
        return "Hello from Parent2!"

class Child(Parent1, Parent2):
    pass

child = Child()
print(child.greet()) # Output: Hello from Parent1!

```

Step-by-Step Explanation:**1. Conflict in Method Names:**

Both Parent1 and Parent2 have a greet() method.

2. MRO Resolution:

Python resolves the method call based on the order in which the parent classes are listed. Since Parent1 is listed first, its greet() method is called.

Types of Inheritance in Python

1. Single Inheritance
2. Multiple Inheritance
3. Multilevel Inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance

Each type is discussed below with examples.

1. Single Inheritance

In single inheritance, a child class inherits from only one parent class.

```

class Animal:
    def sound(self):
        return "This is a generic animal sound."

```

```

class Dog(Animal):
    def sound(self):
        return "Bark!"

```

```

# Create objects
animal = Animal()
dog = Dog()

```

```

print(animal.sound()) # Output: This is a generic animal sound.
print(dog.sound())   # Output: Bark!

```

Step-by-Step Explanation:**1. Parent Class:**

The Animal class has a sound() method.

2. Child Class:

The Dog class inherits from Animal and overrides the sound() method to provide its specific implementation.

3. Object Creation:

Calling sound() on a Dog object invokes the overridden method, while calling it on an Animal object invokes the parent's method.

2. Multiple Inheritance

In multiple inheritance, a child class inherits from more than one parent class.

```
class Engine:
```

```
    def engine_type(self):
        return "Diesel engine"
```

```
class Body:
```

```
    def body_type(self):
        return "Sedan"
```

```
class Car(Engine, Body):
```

```
    def details(self):
        return f'{self.engine_type()} in a {self.body_type()}'
```

```
# Create an object
```

```
car = Car()
```

```
print(car.details()) # Output: Diesel engine in a Sedan
```

Step-by-Step Explanation:**1. Parent Classes:**

Engine and Body each define different methods.

2. Child Class:

The Car class inherits from both Engine and Body, combining functionality from both.

3. Object Creation:

The Car object can access methods from both parent classes.

3. Multilevel Inheritance

In multilevel inheritance, a class inherits from a child class, forming a chain of inheritance.

```
class Vehicle:
```

```
    def type(self):
        return "General Vehicle"
```

NOTE

NOTE

```

class Car(Vehicle):
    def type(self):
        return "Car"

class ElectricCar(Car):
    def type(self):
        return "Electric Car"

# Create an object
tesla = ElectricCar()
print(tesla.type()) # Output: Electric Car

```

Step-by-Step Explanation:**1. Inheritance Chain:**

ElectricCar inherits from Car, which inherits from Vehicle.

2. Overriding:

Each class provides its own version of the type() method.

3. Object Creation:

Calling type() on an ElectricCar object invokes the method defined in ElectricCar.

4. Hierarchical Inheritance

In hierarchical inheritance, multiple child classes inherit from a single parent class.

```

class Shape:
    def info(self):
        return "This is a shape."

class Circle(Shape):
    def area(self, radius):
        return 3.14 * radius * radius

class Rectangle(Shape):
    def area(self, length, breadth):
        return length * breadth

# Create objects
circle = Circle()
rectangle = Rectangle()

print(circle.info()) # Output: This is a shape.
print("Circle area:", circle.area(5)) # Output: Circle area: 78.5
print("Rectangle area:", rectangle.area(4, 6)) # Output: Rectangle area: 24

```

Step-by-Step Explanation:**1. Parent Class:**

Shape provides a common method.

2. Child Classes:

Circle and Rectangle inherit from Shape and introduce specific functionality.

3. Object Creation:

Instances of Circle and Rectangle can access both inherited and unique methods.

5. Hybrid Inheritance

Hybrid inheritance combines two or more inheritance types.

```
class A:
```

```
    def method_A(self):
        return "Method from Class A"
```

```
class B(A):
```

```
    def method_B(self):
        return "Method from Class B"
```

```
class C:
```

```
    def method_C(self):
        return "Method from Class C"
```

```
class D(B, C):
```

```
    def method_D(self):
        return "Method from Class D"
```

```
# Create an object
```

```
d = D()
```

```
print(d.method_A()) # Output: Method from Class A
```

```
print(d.method_B()) # Output: Method from Class B
```

```
print(d.method_C()) # Output: Method from Class C
```

```
print(d.method_D()) # Output: Method from Class D
```

Step-by-Step Explanation:**1. Parent and Child Classes:**

D inherits from both B and C, while B inherits from A.

2. Method Resolution:

Python determines the order of method calls based on the Method Resolution Order (MRO).

NOTE

NOTE

10.5 ITERATORS

Iterators in Python**Definition:**

In Python, an object called an iterator allows elements in a collection (such a list, tuple, or dictionary) to be traversed sequentially without requiring an explicit counter to be kept track of. Two methods are implemented by iterators:

1. `__iter__()`: Gives back the actual iterator object.
2. `__next__()`: Gives back the subsequent item in the series. It raises a `StopIteration` exception if there are no more items.

Python's for loops and numerous other sequence handling features are built on iterators.

Creating and Using Iterators**Example Code:**

```
# Simple example of an iterator
my_list = [1, 2, 3, 4]
```

```
# Get an iterator object
iterator = iter(my_list)
```

```
# Traverse using the iterator
print(next(iterator)) # Output: 1
print(next(iterator)) # Output: 2
print(next(iterator)) # Output: 3
print(next(iterator)) # Output: 4
```

```
# Calling next() beyond this will raise StopIteration
```

Step-by-Step Explanation:

1. **iter() Function:** The `iter(my_list)` converts `my_list` into an iterator object.
2. **next() Function:** The next item in the iterator is retrieved using the `next()` function. A `StopIteration` exception is raised when `next()` is used after all elements have been used.

Custom Iterator

By establishing a class with the `__iter__()` and `__next__()` methods, you can construct a custom iterator.

Example Code:

```
class MyNumbers:
    def __init__(self, start, end):
        self.current = start
        self.end = end
```

NOTE

```
def __iter__(self):
    return self

def __next__(self):
    if self.current <= self.end:
        number = self.current
        self.current += 1
        return number
    else:
        raise StopIteration
```

```
# Create an iterator
numbers = MyNumbers(1, 5)
for num in numbers:
    print(num)
```

Step-by-Step Explanation:

1. **Initialization:** The class `MyNumbers` initializes with start and end values to define the range.
2. The iterator object itself is returned by the `__iter__()` method.
3. The `__next__()` method advances the counter and returns the current number. The counter rises `StopIteration` if it surpasses end.
4. **Usage in for Loop:** The for loop automatically handles calling `__iter__()` and `__next__()`.

Using Iterators with Generators

A **generator** is a simpler way to create iterators using `yield`. Generators are functions that pause their execution and resume from where they left off.

Example Code:

```
def my_generator():
    yield 1
    yield 2
    yield 3

# Use the generator
for value in my_generator():
    print(value)
```

Step-by-Step Explanation:

1. **Generator Function:** The `my_generator()` function uses `yield` to produce values one at a time.
2. **Usage in Loops:** The generator's output is iterated over by the for loop.

NOTE**Practical Use Cases of Iterators****1. File Reading:**

```
with open('example.txt', 'r') as file:
    for line in file:
        print(line.strip())
```

Explanation: File objects in Python are iterators that yield one line at a time.

2. Infinite Sequence (e.g., Fibonacci):

```
class Fibonacci:
    def __init__(self):
        self.a, self.b = 0, 1

    def __iter__(self):
        return self

    def __next__(self):
        self.a, self.b = self.b, self.a + self.b
        return self.a

fib = Fibonacci()
for i in range(5):
    print(next(fib))
```

**Activity 10.1****1. Creating and Modifying Classes**

- Write a Python class named Car with attributes brand and model.
- Create an object of the Car class and modify its attributes.

2. Implementing Inheritance

- Create a child class ElectricCar that inherits from Car and adds a new attribute battery_capacity.
- Create an object of ElectricCar and display all attributes.

Materials Needed:

- A computer with Python installed
- A text editor (e.g., VS Code, PyCharm) or Python IDE
- Internet access for documentation reference

.....

.....

.....

.....

NOTE**☑ Check Your Progress 10.1**

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

1. What is the purpose of creating objects in Python, and how are they related to classes?

.....

2. How does inheritance improve code efficiency in Python?

.....

10.6 SUMMARY

Classes and objects are fundamental in Python's OOP model, providing a structured way to create reusable components. Objects are instances of classes that can be created, modified, and deleted dynamically. **Inheritance** allows new classes to inherit attributes and methods from existing ones, reducing redundancy. **Iterators** simplify data access by enabling objects to be looped over using `__iter__()` and `__next__()` methods. Understanding these concepts improves Python programming efficiency, making code modular and scalable.

10.7 GLOSSARY

- **Class (class)** – A blueprint for creating objects, defining properties and methods.
- **Object** – An instance of a class that holds specific data and behaviors.
- **Create Object** – Instantiating a class to form an object (`obj = ClassName()`).
- **Modify Object** – Updating object attributes (`obj.name = "New Value"`).
- **Delete Object (del)** – Removing an object using the `del` keyword (`del obj`).
- **Inheritance** – A mechanism where a class derives attributes and methods from another class.
- **Parent Class (Superclass)** – The class whose properties are inherited.

NOTE

- **Child Class (Subclass)** – A class that inherits from another class.
- **Iterator** – An object that enables sequential iteration using `__iter__()` and `__next__()`.
- **Iterable** – An object that can return an iterator (e.g., lists, tuples, dictionaries).

10.8 ANSWERS TO CHECK YOUR PROGRESS**1. What is the purpose of creating objects in Python, and how are they related to classes?**

Answer: Objects in Python are **instances of a class**, meaning they represent a specific realization of the class blueprint. A class defines the structure and behavior (attributes and methods), while objects store actual data and interact with other objects. Creating objects allows reusability and modularity in programming, making it easier to manage and manipulate data.

2. How does inheritance improve code efficiency in Python?

Answer: Inheritance allows a **child class to inherit attributes and methods** from a **parent class**, reducing code duplication. Instead of rewriting common functionalities, a subclass can extend or modify the behavior of the parent class, making the code **more efficient, reusable, and easier to maintain**.

10.9 SUGGESTED READINGS

Sr. No.	Title	URL	QR Code
1	Open Educational Resource	http://books.aisectionline.com/UnitUrls?unitId=1285	

10.10 REFERENCE

- **"Python Crash Course"** by Eric Matthes
- **"Automate the Boring Stuff with Python"** by Al Sweigart
- **"Learning Python"** by Mark Lutz

10.11 EXERCISE**Part 1****A. Multiple Choice Questions (MCQs)****1. What is an object in Python?**

- a) A function that prints output

NOTE

- b) A variable that stores a single value
- c) An instance of a class
- d) A module

2. Which keyword is used to define a class in Python?

- a) def
- b) object
- c) class
- d) new

3. How do you delete an object in Python?

- a) del object_name
- b) remove object_name
- c) delete(object_name)
- d) discard object_name

4. What is the purpose of inheritance in Python?

- a) To define new functions
- b) To allow a child class to reuse properties and methods from a parent class
- c) To store data permanently
- d) To execute a function multiple times

5. Which method is used to return an iterator object?

- a) __next__()
- b) __iter__()
- c) __getitem__()
- d) __call__()

Answers with Explanations:

- 1. c) An instance of a class** – An object is a real-world entity created from a class in Python.
- 2. c) class** - The class keyword is used to define a class in Python.
- 3. a) del object_name** - The del keyword removes the object reference from memory.
- 4. b) To allow a child class to reuse properties and methods from a parent class** - Inheritance enables code reuse and hierarchical class relationships.
- 5. b) __iter__()** - The __iter__() method returns the iterator object itself.

NOTE**B. Short Answer Type Questions**

1. What are the advantages of using object-oriented programming in Python?
2. Explain the difference between an instance variable and a class variable.
3. How does an iterator work in Python, and what methods must be implemented in an iterator class?

Part 2**C. Long Answer Type Questions (200 words)**

1. Explain the process of creating, modifying, and deleting classes and objects in Python with examples. How does encapsulation contribute to security in object-oriented programming?
2. Discuss the concept of inheritance in Python. Explain different types of inheritance with examples. How does inheritance promote code reusability and efficiency?

Part 3**D. Long Answer Type Questions (300 words)**

1. Discuss the principles of OOP including class, object, encapsulation, inheritance, and polymorphism. Explain how they help structure programs.
2. Explain the process of creating classes and objects in Python. Discuss constructors, instance variables, and methods.
3. Describe real-world examples where OOP is used. Explain how modeling improves software design.

UNIT 11

SCOPE AND MODULES

NOTE

UNIT STRUCTURE

- 11.1 Introduction
- 11.2 Objective
- 11.3 Scope
- 11.4 Modules (Random, Math, Statistics)
- 11.5 Summary
- 11.6 Glossary
- 11.7 Answers to check your progress
- 11.8 Suggested readings - OER
- 11.9 Reference
- 11.10 Exercise

11.1 INTRODUCTION

In Python, **scope** defines the visibility and lifetime of variables in a program, determining where a variable can be accessed. Python follows different scopes, including **local**, **global**, **enclosed**, and **built-in scopes**. **Modules** in Python are collections of functions and variables that can be imported to extend functionality without rewriting code. Some commonly used built-in modules are **Random**, which deals with generating random numbers, **Math**, which provides mathematical functions, and **Statistics**, which helps perform statistical calculations like mean and median. Understanding scope and modules enhances efficiency and code organization in Python programming.

11.2 OBJECTIVE

After completing this unit, you will be capable of:

- To understand the concept of **variable scope** and how it affects accessibility.
- To explore **local**, **global**, **enclosed**, and **built-in scopes** in Python.
- To learn about **modules** and how they provide additional functionality.
- To study key built-in modules: **Random**, **Math**, and **Statistics** for numerical operations.
- To use modules effectively to simplify complex programming tasks.

11.3. SCOPE

Scope in Python

Definition:

In Python, scope describes the area of a program where a specific variable can be accessed or changed. Writing clear, effective code and avoiding mistakes brought on by variable shadowing, accidental

NOTE

modifications, or name conflicts require an understanding of scope. Python's search order for variables is governed by a set of rules known as LEGB (Local, Enclosing, Global, Built-in).

Types of Scope in Python**1. Local Scope:**

variables that are specified inside a code block or function.

2. Enclosing Scope:

variables that are defined in a function that encloses another function (nested functions, for example).

3. Global Scope:

variables that are defined outside of any function or block at the top level of a script or module.

4. Built-in Scope:

Variables and functions that are available in Python by default, such as `print()` and `len()`.

LEGB Rule (Local, Enclosing, Global, Built-in)

When a variable is accessed, Python searches for it in the following order:

- 1. Local:** First, Python looks in the current function's scope.
- 2. Enclosing:** If not found, Python searches in the enclosing function's scope (if any).
- 3. Global:** If still not found, Python searches in the global scope.
- 4. Built-in:** Finally, Python searches in the built-in scope (i.e., standard Python functions and exceptions).

Local Scope

Variables defined within a function belong to the function's local scope. These variables cannot be accessed outside the function.

Example Code:

```
def my_function():
    x = 10 # x is local to my_function
    print("Inside the function:", x)

my_function()
# print(x) # This would result in an error as x is not accessible outside
the function.
```

Step-by-Step Explanation:

1. The variable `x` is declared inside the `my_function()` function, making it local to that function.
2. Attempting to access `x` outside the function causes a `NameError` because `x` exists only within the function's scope.

Enclosing Scope

The enclosing scope applies to variables in an outer function when there is a nested function. An inner function can access variables from its enclosing function.

Example Code:

```
def outer_function():
    y = 20 # y is in the enclosing scope
    def inner_function():
        print("Inside the inner function:", y) # Access variable from the
        outer function
    inner_function()

outer_function()
```

Step-by-Step Explanation:

1. **Outer Function:** The variable `y` is defined within `outer_function()`.
2. **Inner Function:** The nested `inner_function()` accesses `y` from the enclosing scope.
3. The code outputs 20 as `inner_function()` successfully retrieves the value of `y` from the outer scope.

Global Scope

A variable declared in the global scope is accessible throughout the script, including within functions, unless shadowed by a local variable with the same name.

Example Code:

```
z = 30 # z is a global variable

def print_z():
    print("Inside the function:", z)

print("In global scope:", z) # Accessible in global scope
print_z() # Accessible within the function
```

Step-by-Step Explanation:

1. The variable `z` is declared in the global scope, outside any function.
2. The global variable `z` can be accessed both in the global context and inside the `print_z()` function.
3. Output:
In global scope: 30
Inside the function: 30

Built-in Scope

Python's built-in scope contains pre-defined functions and variables that are always accessible, such as `len()`, `max()`, and exceptions like `TypeError`.

NOTE

NOTE**Example Code:**

```
print(len("Hello")) # len() is a built-in function
```

Step-by-Step Explanation:

1. len() is a built-in Python function available for use without requiring an import statement.
2. Calling len("Hello") computes the length of the string "Hello", producing an output of 5.

Modifying Global Variables Inside Functions

To modify a global variable from within a function, the global keyword must be used. Without it, Python will treat the variable as a local one, creating a new instance within the function.

Example Code:

```
x = 50 # Global variable

def modify_global():
    global x # Declare x as global
    x = 100 # Modify the global variable

modify_global()
print(x) # Output: 100
```

Step-by-Step Explanation:

1. The global variable x is initialized with the value 50.
2. Within modify_global(), the global keyword is used to indicate that the function should modify the global x rather than creating a new local variable.
3. The value of x is updated to 100, and this change is reflected globally, producing an output of 100.

11.4. MODULES (RANDOM, MATH, STATISTICS)

Python Modules: Random, Math, and Statistics**Understanding Modules**

A module in Python is essentially a file containing Python code that can define functions, classes, and variables. Modules help organize code into logical groupings and enable code reuse. Python includes a variety of built-in modules that provide essential functionalities like generating random numbers, performing mathematical calculations, and conducting statistical analyses. The random, math, and statistics modules are among the most commonly used ones.

1. The random Module

The random module is designed to generate pseudo-random numbers for various purposes. It enables tasks such as generating random

values, selecting random elements from sequences, and shuffling sequences.

Key Functions:

- **random():** Generates a random floating-point number between 0 and 1.
- **randint(a, b):** Returns a random integer between a and b (inclusive).
- **choice(sequence):** Picks a random element from a given sequence.
- **shuffle(sequence):** Randomly rearranges the elements in a sequence in place.

Example Code:

```
import random
# Generate a random float between 0 and 1
print("Random float:", random.random())

# Generate a random integer between 1 and 10
print("Random integer:", random.randint(1, 10))

# Choose a random element from a list
colors = ['red', 'blue', 'green', 'yellow']
print("Random color:", random.choice(colors))

# Shuffle the list
random.shuffle(colors)
print("Shuffled list:", colors)
```

Explanation of the Code:

1. **random.random():** Produces a random float within the range [0, 1).
2. **random.randint(1, 10):** Generates a random integer between 1 and 10.
3. **random.choice(colors):** Randomly selects one color from the list.
4. **random.shuffle(colors):** Rearranges the order of elements in the list.

Example Output:

```
Random float: 0.3830952894779466
Random integer: 7
Random color: green
Shuffled list: ['yellow', 'blue', 'green', 'red']
```

2. The math Module

The math module provides access to a wide range of mathematical functions and constants, such as trigonometric functions, logarithms, and the value of pi.

NOTE

NOTE**Key Functions:**

- **sqrt(x)**: Computes the square root of x.
- **factorial(x)**: Returns the factorial of x.
- **sin(x)**: Calculates the sine of x (input in radians).
- **pi**: A constant representing the value of pi (approximately 3.14159).

Example Code:

```
import math

# Calculate square root of 25
print("Square root of 25:", math.sqrt(25))

# Calculate factorial of 5
print("Factorial of 5:", math.factorial(5))

# Calculate sine of 90 degrees (convert to radians first)
print("Sine of 90 degrees:", math.sin(math.radians(90)))

# Access the value of pi
print("Value of pi:", math.pi)
```

Explanation of the Code:

1. **math.sqrt(25)**: Computes the square root of 25, yielding 5.
2. **math.factorial(5)**: Returns 120, which is the factorial of 5 ($5 * 4 * 3 * 2 * 1$).
3. **math.sin(math.radians(90))**: Converts 90 degrees to radians and calculates its sine.
4. **math.pi**: Retrieves the constant value of pi.

Example Output:

```
Square root of 25: 5.0
Factorial of 5: 120
Sine of 90 degrees: 1.0
Value of pi: 3.141592653589793
```

3. The statistics Module

The statistics module contains tools for performing statistical calculations on numeric data, such as finding averages, medians, and measures of spread.

Key Functions:

- **mean(data)**: Calculates the average of the data.
- **median(data)**: Finds the median (middle value) of the data.
- **stdev(data)**: Computes the standard deviation, indicating the data's spread.
- **variance(data)**: Calculates the variance, the square of the standard deviation.

Example Code:

```
import statistics

# Define a list of numbers
data = [10, 20, 30, 40, 50]

# Calculate the mean
print("Mean:", statistics.mean(data))

# Calculate the median
print("Median:", statistics.median(data))

# Calculate the standard deviation
print("Standard Deviation:", statistics.stdev(data))

# Calculate the variance
print("Variance:", statistics.variance(data))
```

Explanation of the Code:

1. **statistics.mean(data)**: Calculates the arithmetic mean: $(10 + 20 + 30 + 40 + 50) / 5 = 30$.
2. **statistics.median(data)**: Finds the middle value of the sorted list, which is 30.
3. **statistics.stdev(data)**: Measures how data points deviate from the mean.
4. **statistics.variance(data)**: Returns the variance, the square of the standard deviation.

Example Output:

```
Mean: 30
Median: 30
Standard Deviation: 15.811388300841896
Variance: 250.0
```

**Activity 11.1****1. Understanding Scope in Python**

- Write a Python program that demonstrates the difference between local and global scope.
- Modify a global variable inside a function using the global keyword.

2. Working with Modules

- Use the random module to generate a random number between 1 and 100.
- Use the math module to find the square root of a user-inputted number.

NOTE

NOTE

- Use the statistics module to calculate the mean and median of a list of numbers.

Materials Needed:

- A computer with Python installed
 - A code editor (e.g., VS Code, PyCharm, Jupyter Notebook)
 - Python Documentation for reference
-
-
-
-

☑ Check Your Progress 11.1

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

1. What is the difference between local and global scope in Python?

.....

.....

.....

.....

2. How can the random module be used to simulate rolling a six-sided die? Provide an example.

.....

.....

.....

.....

11.5 SUMMARY

Scope in Python determines the accessibility of variables within different parts of a program, categorized into **local, global, enclosed, and built-in scopes**. Proper use of scope ensures data security and prevents unintended modifications. Modules are pre-written Python files that enhance functionality and allow code reuse. The **Random** module generates random numbers, the **Math** module provides mathematical operations, and the **Statistics** module offers statistical computations. Mastering scope and modules helps in writing efficient and well-structured Python programs.

11.6 GLOSSARY

- **Scope** – The area where a variable is accessible within a program.
- **Local Scope** – Variables declared inside a function, accessible only within that function.

NOTE

- **Global Scope** – Variables declared outside functions, accessible throughout the program.
- **Enclosed Scope** – Variables in nested functions, accessible within inner functions.
- **Built-in Scope** – Predefined variables and functions provided by Python.
- **Module (import)** – A file containing Python functions and variables that can be reused.
- **Random Module (random)** – Provides functions to generate random numbers (e.g., `random.randint(1, 10)`).
- **Math Module (math)** – Contains mathematical functions like square root and trigonometry (`math.sqrt(25)`).
- **Statistics Module (statistics)** – Provides statistical functions like mean and median (`statistics.mean([1,2,3])`).

11.7 ANSWERS TO CHECK YOUR PROGRESS**1. What is the difference between local and global scope in Python?**

Answer: A **local scope** refers to variables declared inside a function and accessible only within that function. A **global scope** refers to variables declared outside any function, making them accessible throughout the entire program. If a function needs to modify a global variable, the `global` keyword must be used.

2. How can the random module be used to simulate rolling a six-sided die? Provide an example.

Answer: The `random.randint(1,6)` function can be used to simulate rolling a six-sided die, as it generates a random integer between 1 and 6.

Example:

```
import random
roll = random.randint(1,6)
print(f"You rolled a {roll}")
```

11.8 SUGGESTED READINGS - OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource	http://books.aisectionline.com/UnitUrls?unitId=1286	

NOTE

11.9 REFERENCE

- **Automate the Boring Stuff with Python** by Al Sweigart
- **"Python Cookbook"** by David Beazley and Brian K. Jones
- **Official Python Documentation:** <https://docs.python.org/3/>

11.10 EXERCISE

Part 1**A. Multiple Choice Questions (MCQs)**

- 1. Which module is used to generate random numbers in Python?**
 - a) math
 - b) statistics
 - c) random
 - d) numpy
- 2. Which function from the math module is used to find the square root of a number?**
 - a) sqrt()
 - b) root()
 - c) square()
 - d) pow()
- 3. What is the purpose of the statistics module in Python?**
 - a) To perform file operations
 - b) To generate random numbers
 - c) To perform statistical calculations
 - d) To manage system configurations
- 4. Which keyword is used to import a module in Python?**
 - a) include
 - b) import
 - c) module
 - d) define
- 5. What will `random.randint(1,10)` return?**
 - a) A float number between 1 and 10
 - b) A random integer between 1 and 9
 - c) A random integer between 1 and 10
 - d) An error

Answers with Explanations:

1. **c) random** – The random module is used to generate random numbers in Python.
2. **a) sqrt()** - The sqrt() function from the math module returns the square root of a number.
3. **c) To perform statistical calculations** – The statistics module is used for mean, median, mode, etc.
4. **b) import** – The import keyword is used to include a module in a program.
5. **c) A random integer between 1 and 10** - randint(1,10) returns an integer including both 1 and 10.

B. Short Answer Type Questions

1. What is the difference between a local and global scope in Python?
2. Explain the use of the random.choice() function with an example.
3. What is the purpose of the math.ceil() and math.floor() functions?

Part 2**C. Long Answer Type Questions (200 words each)**

1. Explain the different types of scopes in Python (local, global, enclosed, and built-in) with examples.
2. Discuss the key functions of the random, math, and statistics modules, including practical applications of each.

Part 3**D. Long Answer Type Questions (300 words)**

1. Explain variable scope in Python. Discuss local, global, and nonlocal variables with examples.
2. Describe the purpose and advantages of modules in Python. Explain how importing modules improves code reusability.
3. Discuss the creation of user-defined modules. Explain how to write, save, import, and use custom modules.

NOTE

UNIT 12

WORKING WITH DATA

NOTE

UNIT STRUCTURE

- 12.1 Introduction
- 12.2 Objective
- 12.3 Dates
- 12.4 Math
- 12.5 JSON
- 12.6 RegEx
- 12.5 Summary
- 12.6 Glossary
- 12.7 Answers to check your progress
- 12.8 Suggested readings - OER
- 12.9 Reference
- 12.10 Exercise
- 12.11 Feedback form

12.1 INTRODUCTION

Python provides built-in modules to handle various types of data efficiently. The **Dates** module (datetime) allows working with dates and times, including formatting and manipulation. The **Math** module (math) provides mathematical functions like trigonometry, logarithms, and square roots. The **JSON** module (json) helps encode and decode JSON data, enabling seamless interaction with web APIs and data storage. The **RegEx** module (re) is used for pattern matching and text processing, allowing efficient searching and modification of strings. These modules play a crucial role in handling different data types effectively in Python.

12.2 OBJECTIVE

After completing this unit, you will be capable of:

- To understand how to manipulate and format **dates** using the datetime module.
- To explore the **math** module for performing advanced mathematical operations.
- To learn how to work with **JSON** data for encoding and decoding structured information.
- To apply **regular expressions (RegEx)** for pattern matching and text processing in Python.
- To use these modules to enhance functionality and simplify complex tasks.

NOTE

12.3 DATES

Working with Dates in Python

Overview:

Managing dates and times is a frequent task in Python, whether it's calculating time intervals, formatting date strings, or handling time zones. Python's built-in datetime module offers versatile classes for date and time manipulation. With this module, you can perform tasks such as parsing dates, date arithmetic, formatting, and comparisons. Key classes in the datetime module include datetime, date, time, and timedelta.

The datetime Module

The datetime module provides several essential components:

1. **datetime.date**: Represents a calendar date (year, month, day).
2. **datetime.time**: Represents a specific time (hour, minute, second).
3. **datetime.datetime**: Combines date and time into a single object.
4. **datetime.timedelta**: Represents a time span, i.e., the difference between two dates or times.
5. **datetime.now()**: Fetches the current local date and time.
6. **datetime.strptime()**: Converts a string to a datetime object based on a specified format.

1. Retrieving the Current Date and Time

The datetime.now() function returns the current date and time as a datetime object, including microseconds.

Example Code:

```
import datetime

# Get the current date and time
current_datetime = datetime.datetime.now()
print("Current Date and Time:", current_datetime)
```

Explanation:

1. datetime.datetime.now() retrieves the current local date and time.
2. The output includes precise details down to microseconds.

Output:

```
sql
Copy code
Current Date and Time: 2024-12-18 10:30:45.123456
```

2. Creating Specific Date and Time Objects

You can create custom date or time objects using the datetime.date and datetime.time classes.

NOTE**Example Code:**

```
import datetime

# Create a specific date
specific_date = datetime.date(2024, 12, 18)
print("Specific Date:", specific_date)

# Create a specific time
specific_time = datetime.time(10, 30, 45)
print("Specific Time:", specific_time)
```

Explanation:

1. `datetime.date(2024, 12, 18)` creates a date object for December 18, 2024.
2. `datetime.time(10, 30, 45)` creates a time object for 10:30:45 AM.

Output:

```
sql
Copy code
Specific Date: 2024-12-18
Specific Time: 10:30:45
```

3. Formatting Dates and Times

The `strftime()` method formats datetime objects into string representations based on a specified format.

Example Code:

```
import datetime

# Get current datetime
current_datetime = datetime.datetime.now()

# Format the datetime object
formatted_datetime = current_datetime.strftime("%Y-%m-%d %H:%M:%S")
print("Formatted Date and Time:", formatted_datetime)
```

Explanation:

1. `strftime("%Y-%m-%d %H:%M:%S")` converts the datetime object into a string using the specified format:
 - `%Y`: Four-digit year (e.g., 2024)
 - `%m`: Two-digit month (e.g., 12)
 - `%d`: Two-digit day of the month (e.g., 18)
 - `%H`: Two-digit hour (24-hour clock)
 - `%M`: Two-digit minute
 - `%S`: Two-digit second

NOTE**Output:**

sql

Copy code

Formatted Date and Time: 2024-12-18 10:30:45

4. Parsing Strings to datetime Objects

The `strptime()` function parses a date string and converts it into a datetime object.

Example Code:

```
import datetime
```

```
# Parse a date string into a datetime object
```

```
date_string = "2024-12-18 10:30:45"
```

```
parsed_datetime = datetime.datetime.strptime(date_string, "%Y-%m-%d %H:%M:%S")
```

```
print("Parsed Date and Time:", parsed_datetime)
```

Explanation:

1. `strptime(date_string, "%Y-%m-%d %H:%M:%S")` converts the string into a datetime object based on the format provided.

Output:

```
Parsed Date and Time: 2024-12-18 10:30:45
```

5. Performing Date Arithmetic with timedelta

The `timedelta` class enables operations like adding or subtracting days, hours, or minutes.

Example Code:

```
import datetime
```

```
# Create a date object
```

```
current_date = datetime.date(2024, 12, 18)
```

```
# Perform arithmetic
```

```
future_date = current_date + datetime.timedelta(days=10)
```

```
past_date = current_date - datetime.timedelta(days=5)
```

```
print("Current Date:", current_date)
```

```
print("Date 10 days in the future:", future_date)
```

```
print("Date 5 days in the past:", past_date)
```

Explanation:

1. `datetime.timedelta(days=10)` represents a 10-day duration.
2. Adding or subtracting `timedelta` objects to/from a date adjusts the date accordingly.

Output:

yaml

Copy code

Current Date: 2024-12-18
 Date 10 days in the future: 2024-12-28
 Date 5 days in the past: 2024-12-13

6. Comparing Dates

You can directly compare dates using operators such as `<`, `>`, or `==`.

Example Code:

```
import datetime

# Create two date objects
date1 = datetime.date(2024, 12, 18)
date2 = datetime.date(2024, 12, 19)

# Compare dates
print("Is date1 earlier than date2?", date1 < date2)
print("Are the dates equal?", date1 == date2)
```

Explanation:

1. `date1 < date2`: Checks if date1 is earlier than date2.
2. `date1 == date2`: Compares whether the two dates are identical.

Output:

```
Is date1 earlier than date2? True
Are the dates equal? False
40
```

12.4 MATH

Mathematics in Python

Overview

The **math module** in Python offers a wide range of mathematical functions and constants. These include basic operations such as exponentiation, logarithms, and trigonometry, as well as advanced computations like factorials and the greatest common divisor (GCD). As part of Python's standard library, the math module is optimized for performance and serves as an essential tool for numerical and scientific programming tasks.

It provides constants like **π (pi)** and **e (Euler's number)**, alongside functions for trigonometric calculations, working with floating-point numbers, and performing algebraic operations.

Key Features of the math Module

1. Basic Mathematical Operations

- `math.sqrt(x)`: Computes the square root of x.
- `math.pow(x, y)`: Raises x to the power of y.
- `math.fmod(x, y)`: Returns the remainder of x divided by y.

2. Trigonometric Functions

- `math.sin(x)`: Calculates the sine of x (input in radians).

NOTE

- `math.cos(x)`: Computes the cosine of `x` (in radians).
- `math.tan(x)`: Determines the tangent of `x` (in radians).
- `math.radians(x)`: Converts an angle from degrees to radians.
- `math.degrees(x)`: Converts an angle from radians to degrees.

3. Exponential and Logarithmic Functions

- `math.exp(x)`: Computes **e** raised to the power of `x`.
- `math.log(x, base)`: Calculates the logarithm of `x` to the specified base.
- `math.log10(x)`: Returns the base-10 logarithm of `x`.

4. Factorial and Combinatorics

- `math.factorial(x)`: Computes the factorial of `x`.
- `math.comb(n, k)`: Calculates the number of ways to select `k` items from `n` items.
- `math.perm(n, k)`: Computes the number of arrangements of `k` items chosen from `n`.

5. Constants

- `math.pi`: The mathematical constant π (approximately 3.14159).
- `math.e`: Euler's number, **e** (approximately 2.71828).

Example Code: Using the math Module**1. Basic Mathematical Operations**

```
import math

# Square root
sqrt_value = math.sqrt(25)
print("Square root of 25:", sqrt_value)

# Exponentiation
pow_value = math.pow(2, 3)
print("2 raised to the power of 3:", pow_value)

# Modulo operation
mod_value = math.fmod(10, 3)
print("10 modulo 3:", mod_value)
```

Explanation:

1. `math.sqrt(25)` calculates the square root, which results in 5.0.
2. `math.pow(2, 3)` computes 2^3 , resulting in 8.0.
3. `math.fmod(10, 3)` determines the remainder of $10 \div 3$, returning 1.0.

Output:

```
Square root of 25: 5.0
2 raised to the power of 3: 8.0
10 modulo 3: 1.0
```

NOTE**2. Trigonometric Functions**

```
import math

# Sine, Cosine, and Tangent
angle_rad = math.radians(90) # Convert 90 degrees to radians
sin_value = math.sin(angle_rad)
cos_value = math.cos(angle_rad)
tan_value = math.tan(angle_rad)

print("Sine of 90 degrees:", sin_value)
print("Cosine of 90 degrees:", cos_value)
print("Tangent of 90 degrees:", tan_value)
```

Explanation:

1. `math.radians(90)` converts 90° to radians (1.5708 radians).
2. `math.sin(angle_rad)` calculates the sine of 90°, yielding 1.0.
3. `math.cos(angle_rad)` computes the cosine of 90°, approximately 0.0.
4. `math.tan(angle_rad)` computes the tangent, which is undefined for 90° and returns a very large value.

Output:

```
Sine of 90 degrees: 1.0
Cosine of 90 degrees: 6.123233995736766e-17
Tangent of 90 degrees: 1.633123935319537e+16
```

3. Exponential and Logarithmic Functions

```
import math

# Exponential
exp_value = math.exp(2)
print("e raised to the power of 2:", exp_value)

# Logarithms
log_value = math.log(100, 10) # Logarithm with base 10
log10_value = math.log10(100) # Base-10 logarithm
print("Logarithm of 100 with base 10:", log_value)
print("Log base 10 of 100:", log10_value)
```

Explanation:

1. `math.exp(2)` computes e^2 , approximately 7.3891.
2. `math.log(100, 10)` calculates the logarithm of 100 to base 10, which equals 2.
3. `math.log10(100)` also returns the base-10 logarithm, which is 2.

Output:

```
e raised to the power of 2: 7.3890560989306495
Logarithm of 100 with base 10: 2.0
Log base 10 of 100: 2.0
```

NOTE**4. Factorial and Combinatorics**

```
import math
```

```
# Factorial
```

```
factorial_value = math.factorial(5)
```

```
print("Factorial of 5:", factorial_value)
```

```
# Combinations
```

```
combinations_value = math.comb(5, 3)
```

```
print("Combinations of 5 choose 3:", combinations_value)
```

Explanation:

1. `math.factorial(5)` calculates $5!$, which equals 120.
2. `math.comb(5, 3)` determines the number of ways to choose 3 items from 5, which is 10.

Output:

Factorial of 5: 120

Combinations of 5 choose 3: 10

12.5 JSON

JSON in Python**Definition:**

JSON (JavaScript Object Notation) is a lightweight, easy-to-use format for exchanging data. It is simple for humans to read and write while also being straightforward for machines to parse and generate. JSON is commonly used for data storage and transmission between a server and a web application or different components of an application. In Python, the `json` module provides tools for handling JSON data. This module includes functions to encode Python objects into JSON format and decode JSON data back into Python objects.

The Python `json` module allows you to:

- Convert Python objects into JSON format using `json.dumps()`.
- Transform JSON data into Python objects using `json.loads()`.
- Read from and write JSON data to files with `json.dump()` and `json.load()`.

Key Functions in the `json` Module

- **`json.dumps(obj)`:**
Converts a Python object (e.g., dictionary, list, tuple) into a JSON-formatted string.
- **`json.loads(json_string)`:**
Parses a JSON string and converts it into a corresponding Python object.

NOTE

- **json.dump(obj, file):**
Serializes a Python object and writes it to a file in JSON format.
- **json.load(file):**
Reads JSON data from a file and deserializes it into a Python object.

1. Encoding Python Objects to JSON

The `json.dumps()` function is used to transform a Python object into a JSON-formatted string.

Example Code:

```
import json

# Python dictionary
person = {
    "name": "Alice",
    "age": 25,
    "is_student": False,
    "courses": ["Math", "Physics", "Computer Science"]
}

# Convert to JSON string
person_json = json.dumps(person)
print("JSON String:", person_json)
```

Explanation:

1. The `person` dictionary stores information about an individual, including name, age, student status, and courses.
2. The dictionary is transformed into a “JSON” string using `json.dumps(person)`.
3. The resulting output is a JSON-formatted “string”.

Output:

```
{"name": "Alice", "age": 25, "is_student": false, "courses": ["Math", "Physics", "Computer Science"]}
```

2. Decoding JSON to Python Objects

The `json.loads()` function is used to parse a JSON string into a Python object.

Example Code:

```
import json

# JSON string
person_json = '{"name": "Alice", "age": 25, "is_student": false, "courses": ["Math", "Physics", "Computer Science"]}'

# Convert JSON to Python object
person = json.loads(person_json)
print("Python Dictionary:", person)
```

NOTE**Explanation:**

1. person_json contains JSON-formatted data as a string.
2. The JSON string is transformed into a 'Python dictionary' using "json.loads(person_json)."
3. The output is the Python representation of the JSON data.

Output:

```
{'name': 'Alice', 'age': 25, 'is_student': False, 'courses': ['Math', 'Physics', 'Computer Science']}
```

3. Writing JSON to a File

The json.dump() function is used to serialize a Python object and save it as a JSON file.

Example Code:

```
import json

# Python dictionary
person = {
    "name": "Alice",
    "age": 25,
    "is_student": False,
    "courses": ["Math", "Physics", "Computer Science"]
}

# Use open("person.json", "w") to write to a JSON file as json_file:

json.dump(person, json_file)
```

Explanation:

1. The information to be saved is in the person dictionary.
2. The dictionary is serialized and sent to a file called person.json using json.dump(person, json_file).
3. The JSON-formatted data will be kept in the file.

Output in person.json:

```
{
  "name": "Alice",
  "age": 25,
  "is_student": false,
  "courses": ["Math", "Physics", "Computer Science"]
}
```

4. Reading "JSON" from a File

JSON data can be read from a file and transformed back into a Python object using the json.load() function.

Example Code:

```
import json

# Read from JSON file
with open("person.json", "r") as json_file:
    person = json.load(json_file)

print("Python Dictionary:", person)
```

Explanation:

1. The JSON data is read from the file person by "json.load(json_file).JSON."
2. It deserializes the data into a Python dictionary.
3. The output displays the dictionary retrieved from the file.

Output:

```
{'name': 'Alice', 'age': 25, 'is_student': False, 'courses': ['Math', 'Physics', 'Computer Science']}
```

5. Handling Nested JSON Objects

The json module can work with complex JSON structures, such as nested objects or arrays.

Example Code:

```
import json

# Nested dictionary
person = {
    "name": "Alice",
    "age": 25,
    "address": {
        "street": "123 Main St",
        "city": "Wonderland"
    },
    "courses": ["Math", "Physics", "Computer Science"]
}

# Convert to JSON string with indentation
person_json = json.dumps(person, indent=4)
print("Nested JSON String:")
print(person_json)
```

Explanation:

1. The person dictionary includes a nested dictionary for the address.
2. json.dumps(person, indent=4) converts the nested structure into a neatly formatted JSON string with indentation.
3. The output displays the JSON structure in a readable format.

Output:

```
{
    "name": "Alice",
```

NOTE

NOTE

```

"age": 25,
"address": {
    "street": "123 Main St",
    "city": "Wonderland"
},
"courses": [
    "Math",
    "Physics",
    "Computer Science"
]
}

```

12.6 REGEX

Regular Expressions (RegEx) in “Python”

A “Regular Expression”: What Is It?

A regular expression (RegEx) is a sequence of characters that defines a search pattern. It's commonly used for tasks like extracting data, replacing text, validating user input, and searching for substrings. Python's `re` module allows for advanced string manipulation and pattern matching.

Regular expressions are popular in text processing, data validation, and extraction due to their ability to describe complex string patterns succinctly. The `re` module in Python provides an effective way to perform these operations.

Key Functions in the “re Module”

1. `re.match(pattern, string)`

- Verifies whether the pattern matches the string's beginning. If it does, it returns a match object; if not, it returns `None`.

2. `re.search(pattern, string)`

- Looks for a pattern match throughout the entire string. If a match is discovered, it returns the first match object; if not, it returns `None`.

3. `re.findall(pattern, string)`

- Provides a list of all the string's non-overlapping matches.
- An empty list is returned if no matches are discovered.

4. `re.finditer(pattern, string)`

- Gives back an iterator that produces match objects for every string match.

5. re.sub”(pattern, replacement, string)”

- Uses the given replacement string to replace every instance of the pattern.

6. re.split(pattern, string)

- Wherever the pattern appears, the string is split, and the resultant segments are returned as a list.

NOTE**Activity 12.1****1. Understanding Dates and Time**

- Write a Python program that prints the current date and time.
- Format the date as YYYY-MM-DD and extract the year separately.

2. Using Math and JSON Modules

- Create a program that calculates the square root of a number using the math module.
- Convert a Python dictionary into a JSON string and back to a dictionary.

3. Regular Expressions in Python

- Write a Python script that uses RegEx to extract all email addresses from a given text.

Materials Needed:

- Python installed on your system
- A text editor (e.g., VS Code, PyCharm, Jupyter Notebook)
- A sample text file (for RegEx email extraction task)

.....

.....

.....

.....

☑ Check Your Progress 12.1

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

1. How do you convert a Python dictionary to a JSON string and back?

.....

.....

.....

.....

NOTE

2. How can you extract all numbers from a given text using Regular Expressions in Python?

.....

12.7 SUMMARY

Python offers specialized modules to handle dates, mathematical operations, JSON data, and text pattern matching. The **datetime** module helps with date and time manipulation, while the **math** module provides essential mathematical functions. The **json** module facilitates the conversion of Python objects to JSON format and vice versa, enabling seamless data exchange. The **re** module allows pattern-based text processing through **regular expressions**. Mastering these modules helps streamline data handling and processing in Python applications.

12.8 GLOSSARY

- **Dates (datetime)** – A module that handles date and time operations (`datetime.datetime.now()`).
- **Math (math)** – A module providing mathematical functions like square root (`math.sqrt(16)`).
- **JSON (json)** – A module used to encode/decode JSON data (`json.dumps(obj)`, `json.loads(json_string)`).
- **RegEx (re)** – A module for pattern matching in text (`re.search(r"\d+", "123abc")`).
- **Datetime Object** – An instance representing a date and time (`datetime.datetime(2024, 3, 6)`).
- **Pattern Matching** – Finding specific sequences in text using regular expressions.
- **Encoding (Serialization)** – Converting a Python object into a JSON string (`json.dumps(obj)`).
- **Decoding (Deserialization)** – Converting a JSON string into a Python object (`json.loads(json_string)`).

12.9 ANSWERS TO CHECK YOUR PROGRESS

1. How do you convert a Python dictionary to a JSON string and back?

Answer: In Python, you can use the `json.dumps()` method to convert a dictionary to a JSON string and `json.loads()` to convert it back to a dictionary.

NOTE

```
import json
data = {"name": "Alice", "age": 25}
json_string = json.dumps(data) # Convert to JSON string
print(json_string) # Output: {"name": "Alice", "age": 25}

dict_data = json.loads(json_string) # Convert back to dictionary
print(dict_data["name"]) # Output: Alice
```

2. How can you extract all numbers from a given text using Regular Expressions in Python?

Answer: You can use the `re.findall()` method with the pattern `\d+` to extract all numbers from a string.

```
import re
text = "My phone number is 12345 and my age is 30."
numbers = re.findall(r"\d+", text) # Finds all numbers
print(numbers) # Output: ['12345', '30']
```

12.10 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource	http://books.aisectionline.com/UnitUrls?unitId=1287	

12.11 REFERENCE

- Sweigart, A. (2019). *Automate the Boring Stuff with Python* (2nd ed.). No Starch Press.
- Beazley, D. (2009). *Python Essential Reference* (4th ed.). Addison-Wesley.

12.12 EXERCISE

Part 1

A. Multiple Choice Questions (MCQs)

1. Which module in Python is used to work with dates?

- time
- datetime
- calendar
- dateutils

NOTE

2. Which of the following is a correct way to load a JSON string in Python?
 - a) `json.load(my_string)`
 - b) `json.loads(my_string)`
 - c) `json.parse(my_string)`
 - d) `json.read(my_string)`
3. Which function is used to find the square root of a number in the math module?
 - a) `math.pow(x, 2)`
 - b) `math.sqrt(x)`
 - c) `math.square(x)`
 - d) `math.sqr(x)`
4. What is the purpose of the `re.search()` function in Python's RegEx module?
 - a) To replace a matched string
 - b) To return all occurrences of a pattern
 - c) To find the first occurrence of a pattern
 - d) To split a string based on a pattern
5. Which function is used to extract the current date in Python?
 - a) `datetime.date.today()`
 - b) `datetime.now()`
 - c) `date.today()`
 - d) `datetime.datetime.today()`

Answers with Explanations:

1. **b) datetime** – The datetime module provides classes to work with dates, times, and date-time combinations.
2. **b) json.loads(my_string)** - `json.loads()` is used to convert a JSON string into a Python dictionary or object.
3. **b) math.sqrt(x)** - The `math.sqrt()` function returns the square root of a given number.
4. **c) To find the first occurrence of a pattern** - `re.search()` scans the string and returns the first match of the pattern if found.
5. **a) datetime.date.today()** - `datetime.date.today()` returns the current date without the time part.

B. Short Answer Type Questions

1. What is JSON, and how does Python handle JSON data?
2. Explain the difference between `match()` and `search()` functions in the `re` module.
3. How can you perform date arithmetic in Python using the `datetime` module?

Part 2**C. Long Answer Type Questions (200 words)**

1. Explain the importance of the `datetime` module in Python. Discuss its key functions with examples, such as retrieving the current date, formatting dates, and performing date arithmetic.
2. Describe the use of Regular Expressions (RegEx) in Python. Explain different functions such as `match()`, `search()`, `findall()`, and `sub()` with suitable examples.

Part 3**D. Long Answer Type Questions (300 words)**

1. Explain different data sources that Python programs work with. Discuss structured and unstructured data.
2. Discuss how Python handles data internally using variables, objects, and data structures.
3. Describe the process of reading, processing, and storing data using basic Python features.

NOTE



DR. C.V. RAMAN UNIVERSITY

Chhattisgarh, Bilaspur AN A/SECT GROUP UNIVERSITY

Approved by: PCI | AICTE | NISTE | BCI | Member of: ANU | Joint Committee: UGC | DSR | AICTE | Recognized by: BGC | A MAC Accredited University



DR. C. V. RAMAN UNIVERSITY (C.G.)

SLM FEEDBACK FORM

PROGRAMME : _____

COURSE : _____

BLOCK : _____

UNIT : _____

Dear Student,

While studying the units of this block, you may have found certain portions of the text difficult to comprehend. We wish to know your difficulties and suggestions, in order to improve the course. Therefore, we request you to fill up and send us the following questionnaire, which pertains to this block. If you find the space provided insufficient, kindly use a separate sheet.

Please mail to aisectcontent@aisect.org/ for online submission please scan QR code:

Questionnaire

1. Name : _____

2. Enrolment No. :

--	--	--	--	--	--	--	--	--	--	--

3. How many hours did you need for studying the units? :

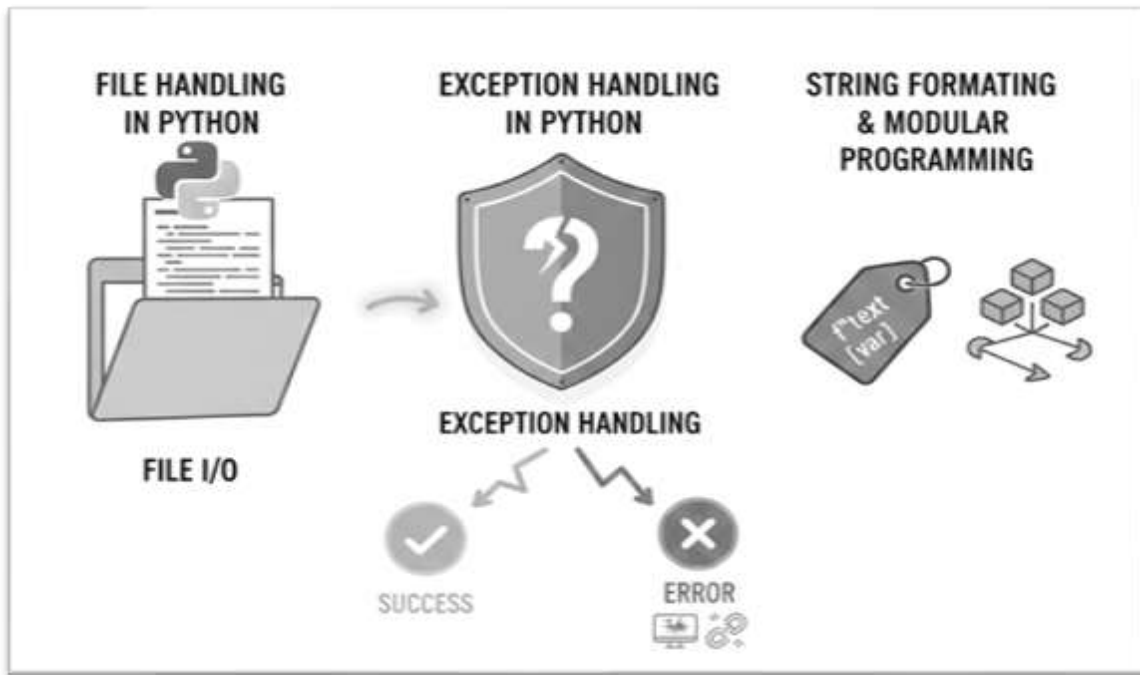
--

4. Please give your reactions to the following items based on your reading of the block:

Items	Excellent	Very Good	Good	Poor	Give specific Examples (If poor)
Presentation Quality					
Language and Style					
Illustrations					
Used (diagrams, tables, etc.)					
Conceptual Clarity					
Check Your Progress Questions					
Suggested Readings - OER					

5. Any other comments:

BLOCK V	FILE AND EXCEPTION HANDLING	151-196
Unit 13	File Handling in Python	153
Unit 14	Exception Handling in Python	169
Unit 15	String Formatting & Modular Programming	179



This block teaches students how Python interacts with external resources and handles unexpected errors in a safe and controlled manner. Students learn how to open, read, write, and manage files, which is essential when building applications that store or process data outside the program. The block also explains exception handling, which allows programs to run smoothly even when problems occur, such as missing files, invalid input, or runtime errors. Additionally, students learn modern techniques for string formatting and the principles of modular programming to create cleaner and better-organized applications. By the end of this block, learners will be able to build Python programs that handle real-world data, manage errors gracefully, and follow professional coding practices.

UNIT 13

FILE HANDLING IN PYTHON

NOTE

UNIT STRUCTURE

- 13.1 Introduction
- 13.2 Objective
- 13.3 User defined Modules and Packages in Python
- 13.4 Files
- 13.5 File manipulations (Read, Write, Create, Delete)
- 13.6 File and Directory related methods
- 13.7 Summary
- 13.8 Glossary
- 13.9 Answers to check your progress
- 13.10 Suggested readings
- 13.11 Reference
- 13.12 Exercise

13.1 INTRODUCTION

Python provides powerful features for organizing and managing code and files efficiently. User-defined modules and packages allow programmers to create reusable code and structure projects effectively. Files in Python enable data storage and retrieval, supporting operations like reading, writing, creating, and deleting files. File manipulations allow handling different types of files, while file and directory-related methods help navigate, modify, and organize the file system. These features make Python an excellent choice for managing data and structuring large applications.

13.2 OBJECTIVE.

After completing this unit, you will be capable of:

- To understand how to create and use **user-defined modules and packages** for better code organization.
- To learn **file handling** operations like reading, writing, creating, and deleting files in Python.
- To explore various **file manipulation methods** for efficient data storage and processing.
- To use **file and directory-related methods** to manage the file system programmatically.
- To implement these concepts to develop structured and data-driven Python applications.

NOTE**13.3 USER DEFINED MODULES AND PACKAGES IN PYTHON**

In Python, a module refers to a single file containing code—such as variables, classes, or functions—that can be imported and reused in other Python programs. A package, on the other hand, is a folder containing related modules, recognized by the special **__init__.py** file. Modules and packages play a crucial role in promoting code reuse and organizing Python projects with a modular structure.

1. User-Defined Modules

A module in Python is simply a file with a .py extension. By defining variables, functions, or classes in the module, you can import and utilize them in other scripts.

Creating and Using a Module:

1. Create a Python file called `math_operations.py` containing this code:

```
# math_operations.py
def add(a, b):
    return a + b
```

```
def subtract(a, b):
    return a - b
```

2. In another script (`main.py`), import and use the module:

```
# main.py
import math_operations

print("Addition:", math_operations.add(5, 3)) # Output: 8
print("Subtraction:", math_operations.subtract(5, 3)) # Output: 2
```

Step-by-Step Breakdown:

- The `math_operations.py` file contains two functions: `add()` and `subtract()`.
- The import statement in `main.py` loads the module, making its functions accessible.
- You can call functions like `math_operations.add(5, 3)` directly from the module.

2. User-Defined Packages

A package is a folder that groups related modules and contains an `__init__.py` file to mark it as a package. Packages help break down large projects into smaller, more manageable sections.

Creating a Package:

1. Organize the following folder structure:
`my_package/`

NOTE**init.py**

arithmetic.py

geometry.py

- `__init__.py`: This file can be empty or include code to initialize the package.
- `arithmetic.py`:

```
def multiply(a, b):
    return a * b
```

- `geometry.py`:

```
def area_circle(radius):
    return 3.14 * radius * radius
```

2. Use the package in your program:

main.py

from my_package.arithmetic import multiply

from my_package.geometry import area_circle

print("Multiplication:", multiply(4, 5)) # Output: 20

print("Area of Circle:", area_circle(7)) # Output: 153.86

Step-by-Step Explanation:

- The `my_package` directory serves as a package containing the modules `arithmetic` and `geometry`.
- The syntax `from my_package.module_name import function_name` is used to import specific functions.
- Functions like `multiply()` and `area_circle()` can then be directly used in the main script.

Benefits of Using Modules and Packages

1. **Modularity**: Divides the codebase into smaller, manageable units.
2. **Reusability**: permits functions and classes to be reused across many projects.
3. **Scalability**: Helps manage larger projects by organizing related components.
4. **Namespace Management**: Reduces naming conflicts by isolating code within module-level namespaces.

Advanced Features

1. **Aliasing Modules**:

import math_operations as mo

print("Addition:", mo.add(10, 5)) # Output: 15

2. **Importing Specific Functions**:

from math_operations import add

NOTE

```
print("Addition:", add(7, 3)) # Output: 10
```

3. Wildcard Imports:

```
from math_operations import *
print("Subtraction:", subtract(9, 2)) # Output: 7
```

Note: It is not recommended to use wildcard imports (*) in large projects because they can cause name conflicts and make the code more difficult to read.

13.5 FILES

Definition

A file is a specific place on a disk where data is kept. Files are a means of storing and retrieving data in Python. To efficiently carry out tasks like reading, writing, appending, and file management, Python comes with built-in functions.

1. File Modes

You must indicate the mode in which the file will be opened when working with files in Python:

- **'r'**: “Read mode (default).”
- **'w'**: “Write mode (creates a new file or replaces existing content).”
- **'a'**: “Append mode (adds content to the file's end).”
- **'r+'**: “Read and write mode.”
- **'b'**: “Binary mode.”

2. File Operations**Opening a File**

To open a file in ‘Python’, utilize the ‘open()’ function:

```
file = open('example.txt', 'r')
# Opens the file in read mode
```

This function returns a file object, which can be utilized for further operations.

Reading a File

You can use methods like “read(), readline(), or readlines()” to access file content.

Example:

Suppose *example.txt* contains:

Hello, World!

Welcome to Python file handling.”

```
file = open('example.txt', 'r')
content = file.read()
print(content) # Output: Entire file content
file.close()
“
```

Step-by-Step:

1. In read mode, open example.txt.
2. To retrieve the complete text as a string, use read().
3. Use the file to close the file. To release system resources, use close().

Writing to a File

To write data, use write() or writelines().

Example:

```
file = open('output.txt', 'w') # Opens file in write mode
file.write("This is a new file.\n")
file.write("File handling in Python is easy.")
file.close()
```

Step-by-Step:

1. Open output.txt in write mode, which, if it doesn't already exist, generates a new file.
2. To add text to the file, use write().
3. To save modifications, close the file.

Appending to a File

The 'a' mode is used to append new content without altering existing data.

Example:

```
file = open('output.txt', 'a')
file.write("\nThis line is appended.")
file.close()
```

Explanation: The new line is added to the file's end without overwriting existing text.

Working with Binary Files

Binary files, such as images and videos, store data in binary format.

Example:

```
with open('image.png', 'rb') as file:
    content = file.read()
    print("Binary Content Read Successfully")
```

3. File Handling Using with

File management is made easier by the with statement, which immediately shuts files following operations.

Example:

```
with open('example.txt', 'r') as file:
    print(file.read()) # No need to explicitly close the file
```

NOTE

NOTE**4. Checking File Properties**

File objects provide attributes like name, mode, and closed for examining file properties.

Example:

```
file = open('example.txt', 'r')

print("File Name:", file.name) # Output: example.txt
print("Mode:", file.mode)      # Output: r
file.close()

print("Is Closed:", file.closed) # Output: True
```

5. Handling File Errors

Python allows you to handle file-related errors using try and except.

Example:

```
try:
    file = open('non_existent.txt', 'r')
except FileNotFoundError:
    print("File not found!")
```

13.4 FILE MANIPULATIONS (READ, WRITE, CREATE, DELETE)

File Manipulation

File manipulation involves performing operations such as reading, writing, creating, and deleting files. Python simplifies these tasks by providing the open() function along with modules like os and shutil for efficient file handling.

1. Reading Files

Reading a file lets you access and process its stored data. Python offers methods like read(), readline(), and readlines() to cater to different use cases.

Example: Reading the entire file content

Suppose example.txt contains:

Hello, Python!

Welcome to file manipulation.

with open('example.txt', 'r') as file:

```
    content = file.read()
    print(content)
```

Step-by-Step Explanation:

1. The file is opened in read mode ('r') using the open() function.

2. The full content is retrieved as a string using the `read()` method.
3. The file will be automatically closed once the operation is complete because of the statement.

2. Writing to Files

Writing to a file either changes the existing content or creates a new file if one doesn't already exist. The `write()` and `writelines()` APIs are commonly used for this.

Example: Writing content to a file

```
with open('output.txt', 'w') as file:
    file.write("This is a new file.\n")
    file.write("File manipulation in Python is powerful.")
```

Step-by-Step Explanation:

1. Begin by opening the `output.txt` file in write mode ('w').
2. The given strings are written to the file using the `write()` function.
3. If the file doesn't exist, it will be created automatically.

3. Creating Files

Files are often created implicitly when opened in write ('w') or append ('a') mode. Alternatively, The `open()` function with mode 'x' or the `OS` module can be used to specifically create a file.

Example: Explicitly creating a file

```
file = open('newfile.txt', 'x') # 'x' ensures the file is newly created
file.close()
```

Explanation:

- A new file is created in the 'x' mode. A `FileExistsError` is raised if the file already exists.

4. Appending to Files

An existing file can have data appended to the end without its current content being overwritten.

Example: Adding content to an 'existing file'

```
with open('output.txt', 'a') as file:
    file.write("\nThis is an appended line.")
```

Explanation:

- The 'a' mode ensures that new content is appended to the file rather than replacing the existing data.

5. Deleting Files

Python's `os` module provides the `remove()` function to delete files.

NOTE

NOTE**Example: Deleting a file**

```
import os

if os.path.exists('newfile.txt'):
    os.remove('newfile.txt')
    print("File deleted successfully.")
else:
    print("File not found.")
```

Step-by-Step Explanation:

1. The existence of the file is confirmed using the `os.path.exists()` function.
2. The `os.remove()` function removes the file if it exists.

6. File Operations with the os Module

The `os` module supports various file system interactions, including renaming and deleting files, as well as checking file existence.

Renaming a File

```
import os

os.rename('output.txt', 'renamed_output.txt')
```

Explanation:

- The `rename()` function changes the file name from `output.txt` to `renamed_output.txt`.

7. Handling Errors During File Operations

Errors like missing files or permission problems can occasionally arise from file operations. Try-except clauses in Python enable elegant error management.

Example: Managing file-related exceptions

```
try:
    with open('nonexistent.txt', 'r') as file:
        print(file.read())
except FileNotFoundError:
    print("Error: File not found.")
```

13.6 FILE AND DIRECTORY RELATED METHODS

Python offers various functions and methods in the `'os'` and `'os.path'` modules for interacting with files and directories. These functions enable tasks such as checking file existence, renaming files, creating

directories, and navigating between directories. Familiarity with these methods is essential for managing files effectively in Python.

1. Working with Files:

Checking File Existence

Python's `os.path` module includes functions to verify whether a file exists, returning `True` or `False` based on the file's availability.

Example: Verifying if a file exists

```
import os
```

```
file_name = 'example.txt'
if os.path.exists(file_name):
    print(f'{file_name} exists.')
else:
    print(f'{file_name} does not exist.')
```

Explanation:

1. To determine whether the file `example.txt` is present in the current working directory, the `os.path.exists()` function is used.
2. It prints a confirmation message if the file is located; if not, it shows that the file is not there.

Renaming Files

By entering the old and new filenames as inputs, you can use the `os.rename()` method to rename files.

Example: Renaming a file

```
import os
old_name = 'old_file.txt'
new_name = 'new_file.txt'
# Rename the file
os.rename(old_name, new_name)
print(f'File renamed from {old_name} to {new_name}.')
```

Explanation:

1. The `os.rename()` function renames “`old_file.txt` to `new_file.txt`.”
2. The file must exist in the directory; otherwise, a `FileNotFoundError` will occur.

Removing Files

To ‘delete’ a file, the ‘`os.remove()` method’ is used. This action permanently removes the file, so it should be used carefully.

NOTE

NOTE

Example: Deleting a file

```
import os

file_name = 'file_to_delete.txt'

if os.path.exists(file_name):
    os.remove(file_name)
    print(f'{file_name} has been deleted.')
else:
    print(f'{file_name} not found.')
```

Explanation:

1. `os.path.exists()` checks if the file exists.
2. If found, `os.remove()` deletes the file.
3. If the file does not exist, it outputs a message indicating that.

2. Working with Directories:**Creating Directories**

While `os.makedirs()` enables the creation of nested directories, `os.mkdir()` is used to create a 'new directory'.

Example: Creating a directory

```
import os

directory_name = 'new_directory'

# Create directory - if it doesn't exist
if not os.path.exists(directory_name):
    os.mkdir(directory_name)
    print(f'Directory '{directory_name}' created successfully.')
else:
    print(f'Directory '{directory_name}' already exists.')
```

Explanation:

1. The `os.mkdir()` function creates a directory named `new_directory`.
2. The check `if not 'os.path.exists()'` ensures that the directory isn't created if it already exists.

Changing Directories

The `os.chdir()` function is used to change the current working directory.

NOTE

Example: Changing the working directory

```
import os
```

```
new_path = '/home/user/my_folder'
os.chdir(new_path)
print(f'Current working directory changed to: {os.getcwd()}')
```

Explanation:

1. The `os.chdir()` function switches to the directory specified by `new_path`.
2. The `os.getcwd()` function confirms and prints the current working directory.

‘Listing Directory’ Contents

To facilitate ‘directory’ navigation and inspection, the `os.listdir()` method produces a list of ‘files and directories’ in the designated directory.

Example: displaying the files in a directory

```
import os
```

```
directory_name = '/home/user/documents'

# List all files and directories in the specified directory
files_and_dirs = os.listdir(directory_name)
print(f'Contents of '{directory_name}': {files_and_dirs}')
```

Explanation:

1. `os.listdir()` returns a ‘list of names’ (not paths) for all files and directories in the given ‘directory’.

Removing Directories

The ‘`os.rmdir()`’ function deletes an empty directory, while `os.removedirs()` can be used to delete an entire directory structure.

Example: Removing a directory

```
import os
```

```
directory_name = 'empty_folder'

# Remove empty directory
if os.path.exists(directory_name) and os.path.isdir(directory_name):
```

NOTE

```

os.rmdir(directory_name)
print(f'Directory '{directory_name}' has been removed.")
else:
    print(f'Directory '{directory_name}' does not exist or is not empty.")

```

Explanation:

1. os.rmdir() removes the empty directory empty_folder.
2. The check ensures that the directory exists and is empty before removal.

3. Working with Pathnames:**Joining Pathnames**

The 'os.path.join()' method is used to combine multiple path components, ensuring platform-independent path creation.

Example: Joining path components

```

import os

path1 = '/home/user'
path2 = 'documents'
full_path = os.path.join(path1, path2)

print(f'Full path: {full_path}')

```

Explanation:

1. os.path.join() merges path1 and path2 into one full path, handling different separators depending on the operating system.

**Activity 13.1****1. Objective**

- To learn how to create and use user-defined modules and packages.
- To understand file manipulation operations such as reading, writing, creating, and deleting files.

2. Instructions

- Create a Python module (mymodule.py) that contains a function to return the square of a number.
- Write a Python script to import the module and use the function.
- Create a text file (data.txt) and write some sample text into it.
- Read the content of data.txt and display it on the screen.
- Try deleting the file using Python and check if it is removed successfully.

Materials Needed:

- A computer with Python installed
 - A text editor or IDE (e.g., VS Code, PyCharm, Jupyter Notebook)
 - Basic understanding of Python syntax
-
-
-
-

NOTE**☑ Check Your Progress 13.1**

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

1. What is the purpose of a Python module, and how is it created?

.....

.....

.....

.....

2. How can you delete a file and a directory in Python?

.....

.....

.....

.....

13.7 SUMMARY

Python supports **user-defined modules and packages**, enabling code reuse and project organization. **File handling** allows interaction with files using operations like reading, writing, creating, and deleting. **File manipulation methods** help manage data efficiently, while **file and directory-related methods** assist in organizing and modifying directories programmatically. Understanding these concepts enhances Python's capability to work with external data and build scalable applications.

13.8 GLOSSARY

- **User-Defined Module** – A Python file (.py) containing functions and classes that can be imported (import mymodule).
- **Package** – A collection of Python modules organized in directories with an `__init__.py` file.
- **File Handling** – The process of reading, writing, creating, and deleting files in Python.
- **Read File (read())** – Reads the content of a file (file.read()).

NOTE

- **Write File (write())** – Writes data to a file (file.write("Hello, World!")).
- **Create File (open() with 'w' or 'x')** – Creates a new file if it doesn't exist (open("file.txt", "w")).
- **Delete File (os.remove())** – Removes a file from the system (os.remove("file.txt")).
- **Directory Methods (os & shutil)** – Functions for creating, deleting, and modifying directories (os.mkdir(), shutil.rmtree()).
- **Path Handling (os.path)** – Methods for handling file paths (os.path.exists("file.txt")).

13.9 ANSWERS TO CHECK YOUR PROGRESS

1. What is the purpose of a Python module, and how is it created?

Answer: A Python module is a reusable file that contains Python code such as functions, classes, or variables. It is created by saving a Python script with a .py extension. A module can be imported into another script using the import statement. For example:

```
# mymodule.py
def greet(name):
    return f'Hello, {name}!'

# main.py
import mymodule
print(mymodule.greet("Alice"))
```

2. How can you delete a file and a directory in Python?

Answer:

To delete a file, use the os.remove() function, and to delete a directory, use os.rmdir() or shutil.rmtree().

```
import os
import shutil

# Deleting a file
if os.path.exists("sample.txt"):
    os.remove("sample.txt")

# Deleting an empty directory
if os.path.exists("my_folder"):
    os.rmdir("my_folder")
```

```
# Deleting a non-empty directory
if os.path.exists("my_folder"):
    shutil.rmtree("my_folder")
```

These functions help in managing files and directories effectively in Python.

13.10 SUGGESTED READINGS

Sr. No.	Title	URL	QR Code
1	Open Educational Resource	http://books.aisectionline.com/UnitUrls?unitId=1288	

NOTE

13.11 REFERENCE

- *Python Crash Course* by Eric Matthes
- Official Python Documentation:
<https://docs.python.org/3/tutorial/>

13.12 EXERCISE

Part 1

A. Multiple Choice Questions (MCQs)

- Which keyword is used to create a module in Python?**
 - define
 - module
 - import
 - No keyword is needed (just create a .py file)
- What method is used to remove a file in Python?**
 - delete()
 - remove()
 - erase()
 - drop()
- Which function is used to write data into a file?**
 - file.append()
 - file.write()
 - file.insert()
 - file.save()
- Which module in Python is commonly used to work with file and directory paths?**
 - sys
 - os
 - file
 - pathlib

NOTE**5. What is the purpose of the `__init__.py` file in a package?**

- a) It is required to initialize the package.
- b) It automatically imports all modules.
- c) It improves the performance of the package.
- d) It prevents the package from being used.

Answers with Explanations:

1. **d) No keyword is needed (just create a .py file)** – A Python module is simply a .py file; no special keyword is required to create it.
2. **b) remove()** - The remove() function (from the os module) is used to delete a file from the system.
3. **b) file.write()** - The write() method is used to write data into a file opened in write or append mode.
4. **d) pathlib** - The pathlib module provides an object-oriented way to work with file and directory paths.
5. **a) It is required to initialize the package** - The `__init__.py` file marks a directory as a Python package and can run initialization code.

B. Short Answer Type Questions

1. Explain the difference between a module and a package in Python.
2. How can you read a file line by line in Python?
3. What are the common directory-related functions in the os module?

Part 2**C. Long Answer Type Questions (200 words)**

1. Explain the concept of user-defined modules and packages in Python. How do they improve code organization and reusability? Provide an example.
2. Describe the different file manipulation methods in Python, including reading, writing, creating, and deleting files. Explain their significance with examples.

Part 3**D. Long Answer Type Questions (300 words)**

1. Explain file handling operations in Python. Discuss opening, reading, writing, and closing files.
2. Describe file modes (`'r'`, `'w'`, `'a'`, `'b'`, etc.). Explain how different modes behave during file operations.
3. Discuss the importance of file handling in data processing applications. Provide real-life examples.

UNIT 14

EXCEPTION HANDLING IN PYTHON

NOTE

UNIT STRUCTURE

- 14.1 Introduction
- 14.2 Objective
- 14.3 Python Exception Handling (Finally)
- 14.4 Raise an Exception
- 14.5 Summary
- 14.6 Glossary
- 14.7 Answers to check your progress
- 14.8 Suggested readings
- 14.9 Reference
- 14.10 Exercise

14.1 INTRODUCTION

Exception handling in Python is a mechanism that allows programs to gracefully handle runtime errors, preventing unexpected crashes. Python provides the try-except block to catch and handle exceptions. The **finally** block is used to execute code regardless of whether an exception occurs, ensuring necessary cleanup actions like closing files or releasing resources. The **raise** statement allows programmers to manually trigger exceptions when specific conditions are met. Exception handling is crucial for writing robust and error-resilient programs.

14.2 OBJECTIVE.

After completing this unit, you will be capable of:

- To understand the importance of **exception handling** in Python.
- To learn how to use try-except to handle runtime errors.
- To explore the **finally** block for executing cleanup actions.
- To study the **raise** statement to manually generate exceptions.
- To implement exception handling for more stable and reliable Python applications.

14.3 PYTHON EXCEPTION HANDLING (FINALLY)

In Python, exception handling is a method for addressing runtime failures that prevents the application from halting suddenly. For efficient error handling, Python has the try, except, else, and finally blocks. The finally block ensures that specific code runs whether or not

NOTE

an exception occurs, while the `raise` keyword is used to manually generate exceptions.

1. Basic Structure of “Exception Handling”

In Python, the `try` and `except` blocks are used for handling exceptions that arise during program execution. If an error occurs within the `try` block, it is identified and managed by the corresponding `except` block.

Example: Handling a “Division by Zero Error”

```
try:
    x = 10 / 0
except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")
```

Step-by-Step Explanation:

1. The `try` block attempts to execute the division operation, which triggers a division by zero error (`10 / 0`).
2. The `except` block gains control after a `ZeroDivisionError` is raised as a result of the improper operation.
3. After catching the exception, the `except` block creates the following output: "Error: Division by zero is not allowed."

2. The finally Block

The `finally` block is used to run code that must execute whether or not an exception was raised. File closure and resource release are two common uses for it. The `finally` block always comes after the `try` and `except` blocks.

Example: Using finally to Close a File

```
file = open("example.txt", "r")
data = file.read()
print(data)
except FileNotFoundError:
    print("Error: The file does not exist.")
finally:
    print("This will always execute.")
    file.close() # Ensures the file is closed
```

Step-by-Step Explanation:

1. An attempt is made by the `try` block to open and read from the file "example.txt."
2. A `FileNotFoundError` is raised and the error message is printed in the `except` block if the file is not found.
3. The `finally` block is executed to guarantee that the file is closed following the action, regardless of whether an exception arises. This block is always the last to run.

14.4 RAISE AN EXCEPTION

NOTE

1. The raise Keyword

In Python, exceptions are explicitly triggered using the raise keyword. This is particularly useful for enforcing conditions or creating custom error messages within the program.

Example: Raising a Custom Exception

```
def check_age(age):
    if age < 18:
        raise ValueError("Age must be 18 or older.")
    else:
        print("You are eligible.")
```

```
try:
    check_age(16)
except ValueError as ve:
    print(f"Error: {ve}")
```

Step-by-Step Explanation:

1. The check_age() function checks whether the age is below 18.
2. If the age is less than 18, a ValueError is raised with the message "Age must be 18 or older."
3. The try-except block catches the exception and prints the error message, "Error: Age must be 18 or older."

2. Raising Built-In Exceptions

TypeError, IndexError, KeyError, and AttributeError are among the built-in exceptions in Python that can be raised with the raise keyword under certain circumstances.

Example: Raising a TypeError for Incorrect Data Type

```
def add_numbers(a, b):
    if not isinstance(a, int) or not isinstance(b, int):
        raise TypeError("Both inputs must be integers.")
    return a + b
```

```
try:
    result = add_numbers(5, "10")
except TypeError as te:
    print(f"Error: {te}")
```

Step-by-Step Explanation:

1. The add_numbers() function checks whether both a and b are integers.
2. If either a or b is not an integer, a TypeError is raised with the message "Both inputs must be integers."
3. The exception is caught in the try-except block, and the error message "Error: Both inputs must be integers." is printed.

NOTE**3. Handling Multiple Exceptions**

Several except blocks can be used to handle several exceptions inside a single try block. This makes it possible to handle various error kinds in certain ways.

Example: Handling Multiple Exceptions

```
try:
    x = int(input("Enter a number: "))
    result = 10 / x
except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")
except ValueError:
    print("Error: Invalid input. Please enter a valid number.")
```

Step-by-Step Explanation:

1. The try block attempts to get user input and perform a division operation.
2. If the user inputs 0, a ZeroDivisionError occurs, which is handled by the first except block.
3. If the user inputs a non-numeric value, a ValueError occurs, and the second except block handles the error by asking for a valid number.

4. The else Block

The else block doesn't run until the try block has raised no exceptions. When you want to accomplish something only if the code in the try block executes correctly, it's useful.

Example: Using else to Confirm Successful Execution

```
try:
    x = 10 / 2
except ZeroDivisionError:
    print("Error: Division by zero.")
else:
    print("Division successful!")
```

Step-by-Step Explanation:

1. The try block performs the division 10 / 2.
2. Since no error occurs, the else block is executed, and "Division successful!" is printed.
3. If a ZeroDivisionError had occurred, the program would print an error message and skip the else block.

**Activity 14.1****1. Understanding try-except and finally Blocks**

- Write a Python program that asks the user to input two numbers and divides them.
- Implement exception handling to manage division by zero errors.

NOTE

- Use a finally block to print a message that the program has completed execution.

2. Using raise to Handle Custom Exceptions

- Write a Python program that asks the user for their age.
- If the age entered is negative, raise a ValueError with a custom message.
- Handle this exception using a try-except block.

Materials Needed:

- A computer with Python installed
- Python IDE or a text editor (e.g., VS Code, PyCharm, Jupyter Notebook)
- Sample input values for testing the programs

.....

.....

.....

.....

☑ Check Your Progress 14.1

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

1. What is the purpose of the finally block in Python's exception handling?

.....

.....

.....

.....

2. How does the raise statement work in Python exception handling?

.....

.....

.....

.....

14.5 SUMMARY

Python's exception handling mechanism prevents program crashes by catching and handling errors using try-except blocks. The **finally** block ensures that specific code runs regardless of exceptions, making it useful for cleanup tasks. The **raise** statement allows programmers to generate custom exceptions when required. Mastering these concepts helps in writing efficient, error-resistant Python programs that can handle unexpected issues smoothly.

NOTE

14.6 GLOSSARY

- **Exception Handling** – A technique to catch and manage runtime errors in Python.
- **Try-Except Block** – A structure that catches exceptions to prevent program crashes.
- **Finally Block (finally)** – A block that executes regardless of whether an exception occurs.
- **Raise (raise)** – A statement used to manually trigger exceptions (raise ValueError("Invalid Input")).
- **Error Handling** – The process of managing and responding to runtime errors.
- **Custom Exceptions** – User-defined exceptions triggered using the raise statement.
- **Runtime Error** – An error that occurs during program execution, such as division by zero.
- **Cleanup Operations** – Tasks like closing files or releasing resources, typically handled in the finally block.

14.7 ANSWERS TO CHECK YOUR PROGRESS

1. What is the purpose of the finally block in Python's exception handling?

Answer: The finally block in Python is used to execute a set of statements regardless of whether an exception occurs or not. It is typically used for **cleanup operations** such as closing files, releasing resources, or displaying final messages. Even if an exception is raised and caught in the try-except block, the finally block will still execute, ensuring that essential tasks are completed.

2. How does the raise statement work in Python exception handling?

Answer: The raise statement in Python is used to **manually trigger an exception** when a specific condition is met. It allows programmers to define and enforce constraints in the program. For example, if a function expects a positive number but receives a negative input, it can raise a ValueError using:

```
def check_age(age):  
    if age < 0:  
        raise ValueError("Age cannot be negative")  
    return f"Valid age: {age}"
```

```
try:  
    print(check_age(-5))  
except ValueError as e:  
    print(f"Error: {e}")
```

This ensures that incorrect values are handled properly rather than causing unexpected behavior in the program.

14.8 SUGGESTED READINGS

Sr. No.	Title	URL	QR Code
1	Open Educational Resource	http://books.aisectionline.com/UnitUrls?unitId=1289	

NOTE

14.9 REFERENCE

- Luciano Ramalho, *Fluent Python*, O'Reilly Media
- Al Sweigart, *Automate the Boring Stuff with Python*, No Starch Press

14.10 EXERCISE

Part 1

A. Multiple Choice Questions (MCQs)

- Which of the following blocks in Python is always executed, whether an exception occurs or not?**
 - try
 - except
 - finally
 - raise
- Which keyword is used to manually trigger an exception in Python?**
 - error
 - exception
 - raise
 - throw
- What happens if an exception occurs inside a try block but is not handled by an except block?**
 - The program continues execution normally
 - The program crashes with an error message
 - The exception is ignored
 - The finally block prevents the program from stopping
- What is the purpose of the finally block in exception handling?**
 - To define custom exceptions
 - To execute code only when an exception occurs
 - To execute important code regardless of exceptions
 - To stop the program execution

NOTE**5. Which of the following is a correct way to raise a ValueError in Python?**

- a) raise ValueError("Invalid input")
- b) exception ValueError("Invalid input")
- c) error ValueError("Invalid input")
- d) throw ValueError("Invalid input")

Answers with Explanations:

1. **c) finally** - The finally block always executes, whether an exception occurs or not.
2. **c) raise** - The raise keyword is used to manually trigger an exception in Python.
3. **b) The program crashes with an error message** - If an exception is not handled by any except block, the program terminates and shows an error.
4. **c) To execute important code regardless of exceptions** - The finally block is used for cleanup code that must run in all situations.
5. **a) raise ValueError("Invalid input")** - This is the correct syntax to raise a built-in ValueError with a custom message.

B. Short Answer Type Questions

1. Explain the difference between try-except and finally blocks in Python exception handling.
2. How does the raise statement help in handling user-defined exceptions?
3. What are some common exceptions in Python, and how can they be handled effectively?

Part 2**C. Long Answer Type Questions (200 words)**

1. Explain the structure of Python's exception handling using try-except, finally, and raise. Provide examples demonstrating different scenarios.
2. Discuss the importance of handling exceptions in Python. How does the use of finally and raise improve the robustness of a program? Provide a real-world example.

Part 3**D. Long Answer Type Questions (300 words)**

1. Explain the need for exception handling. Discuss how errors affect program execution.
2. Describe `try`, `except`, `else`, and `finally` blocks with detailed examples.
3. Discuss common Python exceptions and how custom exceptions can be created.

UNIT 15

STRING FORMATTING & MODULAR PROGRAMMING

NOTE

UNIT STRUCTURE

- 15.1 Introduction
- 15.2 Objective
- 15.3 String Formatting: String Format
- 15.4 Multiple Values
- 15.5 Index Numbers
- 15.6 Named Indexes
- 15.7 Summary
- 15.8 Glossary
- 15.9 Answers to check your progress
- 15.10 Suggested readings
- 15.11 Reference
- 15.12 Exercise
- 15.13 Feedback form

15.1 INTRODUCTION

String formatting in Python allows developers to insert variables and values into strings dynamically, making output more readable and flexible. Python provides different ways to format strings, including the **format() method**, which enables inserting **multiple values** into a string. It supports **index numbers** to control the order of values and **named indexes** for better readability. Mastering string formatting techniques helps in creating well-structured and user-friendly output in Python applications.

15.2 OBJECTIVE.

After completing this unit, you will be capable of:

- To understand different methods of **string formatting** in Python.
- To learn how to insert **multiple values** into a formatted string.
- To explore **index numbers** for controlling the order of inserted values.
- To use **named indexes** for improved readability in formatted strings.
- To apply string formatting techniques in real-world applications like reports and messages.

NOTE**15.3. STRING FORMATTING: STRING FORMAT**

String formatting in Python is a technique for embedding expressions within string literals to create more readable, dynamic, and flexible outputs. Python provides various methods to format strings, allowing the inclusion of variables, values, and expressions within the string. One of the most widely used methods is the `str.format()` function.

1. Basic String Formatting with `str.format()`

The `str.format()` method in Python lets you add values to a string. It is a flexible tool that can manage intricate formatting requirements, such as managing alignment, position, and number formatting.

Example: Basic Usage of `str.format()`

```
name = "Alice"
age = 30
formatted_string = "My name is {} and I am {} years old.".format(name,
age)
print(formatted_string)
```

Explanation:

1. "Alice" and "30," respectively, are used to initialize the two variables, name and age.
2. The data entered into the `format()` function will replace the placeholders `{}` in the string "My name is {} and I am {} years old."
3. The `format()` method takes name and age as inputs and inserts them into the string.
4. The output will be: "My name is Alice and I am 30 years old."

2. Using Positional and Keyword Arguments in `str.format()`

The `format()` method also supports positional and keyword arguments, allowing more control over the placement of values within the string.

Example: Positional and Keyword Arguments

```
formatted_string = "Name: {}, Age: {}".format("Bob", 25)
print(formatted_string)

formatted_string_keyword = "Name: {name}, Age: {age}.".format(name="Bob", age=25)
print(formatted_string_keyword)
```

Explanation:

1. In the first example, positional arguments are utilized, where "Bob" and 25 are entered into the placeholders in the order they occur.
2. In the second example, keyword 'arguments' are used, where the placeholders `{name}` and `{age}` are replaced by the corresponding values passed as `name="Bob"` and `age=25`.

3. Both outputs will be "Name: Bob, Age: 25", but the method of passing the values differs.

3. Reordering Arguments in str.format()

You can control the order in which arguments are inserted into the string by specifying indices inside the curly braces.

Example: Reordering Arguments

```
formatted_string = "Name: {1}, Age: {0}".format(30, "Alice")
print(formatted_string)
```

Explanation:

1. The format() method receives two arguments: 30 and "Alice".
2. The placeholders {1} and {0} correspond to the second and first arguments, respectively, so the output will be "Name: Alice, Age: 30" despite the order being reversed in the string.

4. Formatting Numbers in Strings

The str.format() method can also be used to format numbers, allowing you to control decimal places, insert commas for thousands, or format as percentages.

Example: Number Formatting

```
price = 1234.56789
formatted_price = "The price is {:.2f}".format(price)
print(formatted_price)
large_number = 1000000
formatted_number = "Formatted number: {:,}".format(large_number)
print(formatted_number)
```

Explanation:

1. In example, the floating-point number 1234.56789 is rounded to two decimal places using "{:.2f}", resulting in "The price is 1234.57".
2. In the second example, the number 1000000 is formatted with commas for thousands using "{:,}", resulting in "Formatted number: 1,000,000".

5. Padding and Alignment with str.format()

You can align text within a fixed-width field using padding and alignment specifiers, such as < for left, > for right, and ^ for center alignment.

Example: Aligning and Padding Strings

```
formatted_string = "{:<10} | {:^10} | {:>10}".format("Left", "Center", "Right")
print(formatted_string)
```

NOTE

NOTE**Explanation:**

1. The placeholders "{:<10}", "{:^10}", and "{:>10}" specify the alignment for left, center, and right within a field of 10 characters.
2. The output will be:

mathematica

Copy code

Left | Center | Right

6. Using 'F-Strings' (Python 3.6+)

'F-strings', which were first introduced in 'Python 3.6', provide a more succinct and understandable way to insert expressions inside strings. You can directly insert variables and expressions by prefixing the string with `f` and enclosing them in curly braces `{}`.

Example: Using F-Strings

```
name = "Charlie"
age = 35
formatted_string = f"My name is {name} and I am {age} years old."
print(formatted_string)
```

Explanation:

1. The string is prefixed with `f`, indicating it is an f-string.
2. The variables `name` and `age` are inserted directly into the string within curly braces.
3. The output will be: "My name is Charlie and I am 35 years old."

F-strings also support inline expressions:

```
result = f"The square of 5 is {5 ** 2}."
print(result)
```

This will output: "The square of 5 is 25."

15.4 MULTIPLE VALUES

Managing multiple values in Python refers to keeping numerous elements in a single variable or data structure. You may effectively store and manage numerous linked items with the language's "built-in data types", which include "lists, tuples, sets, and dictionaries". Additionally, Python gives you flexibility and strength when working with several data points by allowing you to unpack and pass multiple values at once.

1. Storing Multiple Values in a List

Multiple values can be stored by separating them with commas inside square brackets `[]`. A list is a sorted and modifiable collection that can contain elements of different data kinds.

NOTE**Example:** Storing Multiple Values in a List

```
fruits = ["apple", "banana", "cherry", "date"]
print(fruits)
```

Explanation:

1. We create a list fruits with four strings: "apple", "banana", "cherry", and "date".
2. The output is the list: ['apple', 'banana', 'cherry', 'date'].

Lists can hold items of different types, such as integers, strings, or even other lists. For instance:

```
mixed_list = [1, "hello", 3.14, [5, 6, 7]]
print(mixed_list)
```

This will output: [1, 'hello', 3.14, [5, 6, 7]], demonstrating the versatility of lists in storing diverse data types.

2. Using Tuples to Store Multiple Values

Similar to a list, a tuple is immutable, which means that once it is generated, it cannot be altered. When you need to store values that should never change, tuples are perfect.

Example: Storing Multiple Values in a Tuple

```
coordinates = (10, 20, 30)
print(coordinates)
```

Explanation:

1. The tuple coordinates contains three integers: 10, 20, and 30.
2. The output is: (10, 20, 30).

Since tuples are immutable, they ensure data integrity by preventing modification after creation.

3. Storing Unique Values with a Set

Only unique things are stored in an unordered collection called a set. It is helpful for validating membership or getting rid of repetitious data because it automatically eliminates duplicates.

Example: Storing Multiple Values in a Set

```
numbers = {1, 2, 3, 4, 4, 5}
print(numbers)
```

Explanation:

1. We create a set numbers with five integers, including the duplicate value 4.
2. The output is: {1, 2, 3, 4, 5}, showing that duplicates are automatically discarded.

4. Storing Multiple Key-Value Pairs with a Dictionary

A dictionary stores data as key-value pairs. It allows you to map each value to a unique key, making it useful for situations that require pairing data, such as linking items to their prices.

Example: Storing Key-Value Pairs in a Dictionary

```
student_scores = {"Alice": 85, "Bob": 92, "Charlie": 78}
print(student_scores)
```

NOTE**Explanation:**

1. We construct a dictionary called `student_scores`, in which the names of the students serve as the keys and their scores as the values.
2. The output is: `{'Alice': 85, 'Bob': 92, 'Charlie': 78}`.

You can access values using their corresponding keys, like so:

```
print(student_scores["Alice"]) # Output: 85
```

5. Unpacking ‘Multiple Values’

Multiple values from collections, like lists or tuples, can be unpacked into individual variables in a single Python command. When you wish to allocate each component of a collection to a different variable, this is helpful.

Example: Unpacking a List into Variables

```
coordinates = [10, 20, 30]
```

```
x, y, z = coordinates
```

```
print(x, y, z)
```

Explanation:

1. The list `coordinates` contains three values: 10, 20, and 30.
2. The statement `x, y, z = coordinates` unpack these values into the variables `x`, `y`, and `z`.
3. The output is: 10 20 30.

6. Passing Multiple Values to Functions

Python allows passing multiple values to functions using various methods, such as using arguments, `*args` “(for arbitrary numbers of positional arguments)”, and `**kwargs` (for keyword arguments).”

Example: Using `*args` to Pass Multiple Values to a Function

```
def print_scores(*scores):
```

```
    for score in scores:
```

```
        print(score)
```

```
print_scores(85, 90, 78, 92)
```

Explanation:

1. The function `print_scores` accepts multiple arguments with `*scores`.
2. We pass four values (85, 90, 78, 92) to the function.
3. Inside the function, it prints each score one by one.
4. The output will be:

Copy code

```
85
```

```
90
```

```
78
```

```
92
```

7. Multiple Values with List Comprehensions

List comprehensions allow you to quickly generate new lists by performing operations on each element of an existing iterable. You can also create a list from multiple values.

Example: Creating a List of Squares Using List Comprehension

```
numbers = [1, 2, 3, 4, 5]
squares = [n**2 for n in numbers]
print(squares)
```

Explanation:

1. We define the list numbers containing five integers.
2. The list comprehension `[n**2 for n in numbers]` iterates over numbers, calculating the square of each element.
3. The resulting list `[1, 4, 9, 16, 25]` is stored in squares and printed.

NOTE

15.5 INDEX NUMBERS

In Python, index numbers refer to the position of an element within a sequence type data structure like a list, tuple, or string. The elements in these structures are ordered, and each element has a unique position known as its index. The index in Python is zero-based, meaning that the first element in the sequence has an index of 0, the second element has an index of 1, and so on.

Understanding index numbers is crucial for accessing and manipulating data in sequences. Negative indices are also supported in Python, which allow access to elements starting from the end of the sequence (from -1).

1. Indexing in Lists

Lists in Python are ordered collections, and you can access any item using its index number. Indexing is one of the most common operations used with lists.

Example: Accessing Elements in a List using Index Numbers

```
fruits = ["apple", "banana", "cherry", "date", "elderberry"]
print(fruits[0]) # Access the first element
print(fruits[2]) # Access the third element
print(fruits[-1]) # Access the last element
```

Step-by-Step Explanation:

1. Five string items make up the defined list of fruits.
2. "Apple" is the first element in the list that is accessed by the `fruits[0]`. Apple will be the final product.
3. The third item in the list, "cherry," is accessed by the `fruits[2]`. Cherry will be the final product.
4. "Elderberry" is the final item in the list, accessed via the `fruits[-1]` function. Elderberry will be the result.

NOTE

o Negative Indexing: The last element is denoted by -1, the second-last by -2, and so forth when employing negative indices.

2. Indexing in Tuples

Tuples are similar to lists, but they are immutable (cannot be changed). Indexing in tuples works the same way as in lists, allowing access to specific elements based on their index.

Example: Accessing Elements in a Tuple using Index Numbers

```
coordinates = (10, 20, 30, 40)
print(coordinates[1]) # Access the second element
print(coordinates[-2]) # Access the second last element
```

Step-by-Step Explanation:

1. A tuple coordinates is defined, containing four integers.
2. The second element of the tuple, 20, is accessed by the coordinates[1]. Twenty will be the result.
3. The coordinates[-2] accesses the second last element, which is 30. The output will be 30.

3. Indexing in Strings

Strings in Python are also ordered sequences, and indexing is used to access individual characters in a string. Each character in the string is assigned an index number, starting from 0.

Example: Accessing Characters in a String using Index Numbers

```
message = "Hello, World!"
print(message[0]) # Access the first character
print(message[7]) # Access the eighth character
print(message[-1]) # Access the last character
```

Step-by-Step Explanation:

1. A string message is defined with value "Hello, World!".
2. message[0] accesses first char of the string, which is "H". The output will be H.
3. The message[7] accesses the eighth character, which is "W". The output will be W.
4. The message[-1] accesses the last character of the string, which is "!". The output will be !.

4. Slicing with Index Numbers

Slicing is another powerful technique in Python that allows you to access a sub-sequence (a portion of the sequence) using a start index, stop index, and an optional step. It is commonly used with lists, tuples, and strings.

Example: Slicing a List using Index Numbers

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
print(numbers[2:5]) # Slice from index 2 to index 4
print(numbers[:4]) # Slice from the beginning to index 3
print(numbers[6:]) # Slice from index 6 to the end
print(numbers[::2]) # Slice with a step of 2
```

NOTE**Step-by-Step Explanation:**

1. A list `numbers` is defined with values from 0 to 9.
2. The slice `numbers[2:5]` selects elements starting from index 2 (inclusive) to index 5 (exclusive). The result will be `[2, 3, 4]`.
3. The slice `numbers[:4]` selects elements from the beginning up to index 4 (exclusive). The result will be `[0, 1, 2, 3]`.
4. From index 6 until the end of the list, elements are chosen using the slice `numbers[6:]`. `[6, 7, 8, 9]` will be the outcome.
5. The slice `numbers[::2]` choose elements from the full list using a step of 2, or every second element. The result will be `[0, 2, 4, 6, 8]`.

5. Modifying Values Using Index Numbers

Strings and tuples are immutable, whereas lists are mutable. While you can use index numbers to change items in a list, you are unable to directly change elements in strings or tuples.

Example: Modifying Elements in a List using “Index Numbers”

```
colors = ["red", "green", "blue", "yellow"]
colors[1] = "purple" # Change the second element
print(colors)
```

Step-by-Step Explanation:

1. A list `colors` is defined with four color names.
2. The `colors[1] = "purple"` statement changes the second element (index 1) from "green" to "purple".
3. The output will be: `['red', 'purple', 'blue', 'yellow']`.

6. Indexing with Iteration

Index numbers are frequently used with loops for iterating over sequences. This enables you to manipulate each element according to its location.

Example: Iterating through a List with Index Numbers

```
fruits = ["apple", "banana", "cherry"]
for i in range(len(fruits)):
    print(f"Index {i}: {fruits[i]}")
```

Step-by-Step Explanation:

1. The list `fruits` is defined with three items.
2. The `for i in range(len(fruits))` loop iterates over the indices of the list, where `i` will take values 0, 1, and 2.
3. Inside the loop, `fruits[i]` accesses the element at index `i` in the list.
4. The output will be:

```
Index 0: apple
Index 1: banana
Index 2: cherry
```

NOTE**15.6 NAMED INDEXES****Definition:**

Named indexes, also referred to as *named arguments* or *named indices*, are a feature in Python that allow you to access values in a sequence or a dictionary using a meaningful, descriptive name instead of using a numeric index or key. Named indexes are especially useful in functions, data structures like dictionaries, and when handling data with Python's more advanced features, such as *namedtuples* and *keyword arguments*.

Named indexes provide clarity in code, improving readability and reducing the chance of errors by referencing data through self-explanatory names rather than relying on numerical indices or arbitrary keys. Python provides several ways to use named indexes effectively, including **namedtuples**, **dictionaries**, and **functions with keyword arguments**.

1. Named Indexes with 'Dictionaries'

In Python, a dictionary is a group of key-value pairs. By acting as named indexes, the keys let you access values by using their names rather than index numbers.

Example: Using Named Indexes with Dictionaries

```
student = {  
    "name": "John",  
    "age": 22,  
    "course": "Computer Science",  
    "grade": "A"  
}  
  
print(student["name"])    # Access value using named index (key)  
print(student["age"])    # Access value using named index (key)
```

Step-by-Step Explanation:

1. A dictionary named `student` is created, where the keys "name", "age", "course", and "grade" are used as named indexes.
2. The statement `student["name"]` accesses the value associated with the 'key' "name", which is "John". The output will be "John".
3. Similarly, `student["age"]` accesses the value "22", and the output will be 22.
 - This demonstrates how named indexes (keys) are used to refer to the associated values in a dictionary.

2. Named Indexes with Namedtuples

In Python, `namedtuple` is a class from the `collections` module that allows you to define a simple class with named fields. `Namedtuples` are like regular tuples, but the elements can be accessed using names

(indexes) instead of numeric positions. This is particularly useful for creating data structures with fields that have meaningful names.

NOTE

Example: Using Named Indexes with Namedtuples

```
from collections import namedtuple

# Define a namedtuple class for a 'Car'
Car = namedtuple("Car", ["make", "model", "year"])

# Create an instance of the Car namedtuple
car1 = Car(make="Toyota", model="Camry", year=2021)

# Access fields using named indexes
print(car1.make) # Access using named index 'make'
print(car1.year) # Access using named index 'year'
```

Step-by-Step Explanation:

1. The namedtuple function is used to create a class Car with fields "make", "model", and "year".
2. A Car object, car1, is instantiated with values "Toyota", "Camry", and 2021 for the respective fields.
3. The car1.make accesses the value of the field "make", which is "Toyota". The output will be Toyota.
4. Similarly, car1.year accesses the "year" field, and the output will be 2021.
 - o This demonstrates the use of named indexes to access elements of a namedtuple.

3. Named Indexes with Functions (Keyword Arguments)

Python functions allow you to pass parameters as *keyword arguments* (named arguments). This enables you to call a 'function' by specifying the name of the parameters instead of just passing them in order. This makes the function calls more readable and allows you to skip arguments if they have default values.

Example: Using Named Indexes with Function Parameters

```
def create_student(name, age, course="Mathematics"):
    print(f"Name: {name}, Age: {age}, Course: {course}")

# Call function using named indexes (keyword arguments)
create_student(name="Alice", age=20)
create_student(name="Bob", age=22, course="Physics")
```

Step-by-Step Explanation:

1. The function create_student is defined with parameters name, age, and course (with a default value of "Mathematics").
2. The function is called twice:
 - o The first call uses named indexes name="Alice" and age=20. Since the course is not specified, the default value "Mathematics" is used.

NOTE

- The second call explicitly specifies all parameters: name="Bob", age=22, and course="Physics".

3. The output will be:

Name: Alice, Age: 20, Course: Mathematics

Name: Bob, Age: 22, Course: Physics

- This shows how named indexes (keyword arguments) make it easier to pass arguments to functions in a readable way.

4. Named Indexes with Data Classes (Python 3.7+)

Starting with Python 3.7, data classes were introduced to provide a more concise way of creating classes with built-in features like comparisons, default values, and field definitions. Data classes use named indexes to access the attributes of objects, making the code more readable.

Example: Using Named Indexes with Data Classes

```
from dataclasses import dataclass
```

```
@dataclass
class Student:
    name: str
    age: int
    course: str
```

```
# Create a Student object
```

```
student1 = Student(name="Tom", age=21, course="Engineering")
```

```
# Access attributes using named indexes
```

```
print(student1.name) # Access using named index 'name'
```

```
print(student1.course) # Access using named index 'course'
```

Step-by-Step Explanation:

1. A data class Student is defined using the @dataclass decorator, with the fields name, age, and course.
2. A Student object, student1, is created with the values "Tom", 21, and "Engineering".
3. The fields are accessed using named indexes:
 - student1.name accesses the "name" attribute, and the output will be Tom.
 - student1.course accesses the "course" attribute, and the output will be Engineering.



Activity 15.1

1. Understanding String Formatting Methods

Write a Python program that asks for the user's name and age, then formats and displays the output using the `format()` method and f-strings.

2. Using Index Numbers and Named Indexes

Modify the program to:

- Print the name and age using index numbers in `format()`.
- Print the name and age using named indexes in `format()`.
- Print the name and age using an f-string.

Materials Needed:

- A computer with Python installed
- A text editor or Python IDE (e.g., VS Code, PyCharm, Jupyter Notebook)

.....

NOTE

☑ Check Your Progress 15.1

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

1. What is the difference between index numbers and named indexes in string formatting?

.....

2. Why are f-strings preferred over the `format()` method in modern Python?

.....

15.7 SUMMARY

Python provides powerful string formatting options using the `format()` method, allowing dynamic insertion of values into strings. It supports **multiple values** to format complex outputs efficiently. **Index numbers** help control the position of values in formatted strings, while **named indexes** enhance readability by explicitly referring to variables. These

FUNDAMENTALS OF PYTHON

NOTE

formatting techniques improve the clarity and structure of Python-generated outputs, making data representation more effective.

15.8 GLOSSARY

- **String Formatting** – A method of inserting variables and values into a string dynamically.
- **format() Method** – A built-in function used for string formatting ("Hello {}".format(name)).
- **Multiple Values** – Inserting more than one value in a formatted string ("{} and {}".format("Alice", "Bob")).
- **Index Numbers** – Numeric positions used in the format() method ("{1} {0}".format("first", "second")).
- **Named Indexes** – Using variable names for placeholders ("My name is {name}".format(name="Alice")).
- **F-Strings (f'')** – A modern alternative for string formatting (f'Hello {name}').
- **Placeholder** – A {} symbol in a string that gets replaced with a value using format().

15.9 ANSWERS TO CHECK YOUR PROGRESS

1. What is the difference between index numbers and named indexes in string formatting?

Answer: Index numbers use numerical positions to insert values in a formatted string ("{1} {0}".format("Apple", "Orange") → "Orange Apple"), while named indexes use variable names for better readability ("{fruit}".format(fruit="Apple") → "Apple").

2. Why are f-strings preferred over the format() method in modern Python?

Answer: F-strings (f'') are preferred because they are **faster, more readable, and require less code**. Instead of using format(), variables can be directly inserted into the string (f'Hello {name}'), making the syntax cleaner.

15.10 SUGGESTED READINGS

Sr. No.	Title	URL	QR Code
1	Open Educational Resource	http://books.aisectionline.com/UnitUrls?unitId=1290	

15.11 REFERENCE

- **"Automate the Boring Stuff with Python"** by Al Sweigart
- **"Learning Python"** by Mark Lutz

15.12 EXERCISE**Part 1****A. Multiple Choice Questions (MCQs)**

- Which method is used for string formatting in Python?**
 - format()
 - stringify()
 - insert()
 - replace()
- What will be the output of "{1} {0}".format("Apple", "Orange")?**
 - Apple Orange
 - Orange Apple
 - {1} {0}
 - Error
- Which of the following is a correct way to use named indexes in format()?**
 - "{fruit}".format(Apple)
 - "{0}".format(fruit="Apple")
 - "{fruit}".format(fruit="Apple")
 - "{}".format(fruit="Apple")
- What is the benefit of using f-strings (f"") over format()?**
 - They are faster and more readable
 - They do not require placeholders {}
 - They work only with numbers
 - They are an outdated method
- What is the output of f"My name is {name}" if name = "John"?**
 - My name is {name}
 - My name is John
 - {name} is my name
 - Error

NOTE

NOTE**Answers with Explanations:**

1. **a) format()** – format() is the built-in method used for string formatting in Python.
2. **b) Orange Apple** - {1} refers to the second argument and {0} refers to the first argument.
3. **c) "{fruit}".format(fruit="Apple")** - Named indexes require matching keyword arguments in format().
4. **a) They are faster and more readable** - f-strings are evaluated at runtime and are easier to read.
5. **b) My name is John** - f-strings directly substitute the value of the variable inside {}.

B. Short Answer Type Questions

1. Explain the difference between index numbers and named indexes in string formatting.
2. What are the advantages of using f-strings (f'') in Python?
3. How can you format a string to display multiple values in a specific order?

Part 2**C. Long Answer Type Questions (200 words)**

1. Describe the different ways of formatting strings in Python with examples. Compare and contrast format(), f-strings, and the older % formatting method.
2. Explain how index numbers and named indexes work in string formatting. Provide examples of when to use each method and discuss their advantages and disadvantages.

Part 3**D. Long Answer Type Questions (300 words)**

1. Explain different string formatting techniques (% formatting, format(), f-strings). Compare their usage.
2. Discuss the importance of modular programming. Explain how breaking programs into modules improves readability and maintenance.
3. Describe how to create reusable modules and packages in Python. Discuss the role of namespaces.



DR. C.V. RAMAN UNIVERSITY

Chhattisgarh, Bilaspur AN AICTE GROUP UNIVERSITY

Approved by: PCI | AICTE | NCTE | BCI | Member of: AUC | Joint Committee: DGC | DEB | AICTE | Recognized by: BGC | A NAAC Accredited University



DR. C. V. RAMAN UNIVERSITY (C.G.)

SLM FEEDBACK FORM

PROGRAMME : _____

COURSE : _____

BLOCK : _____

UNIT : _____

Dear Student,

While studying the units of this block, you may have found certain portions of the text difficult to comprehend. We wish to know your difficulties and suggestions, in order to improve the course. Therefore, we request you to fill up and send us the following questionnaire, which pertains to this block. If you find the space provided insufficient, kindly use a separate sheet.

Please mail to aisectcontent@aisect.org/ for online submission please scan QR code:

Questionnaire

1. Name : _____

2. Enrolment No. :

--	--	--	--	--	--	--	--	--	--	--

3. How many hours did you need for studying the units? :

--

4. Please give your reactions to the following items based on your reading of the block:

Items	Excellent	Very Good	Good	Poor	Give specific Examples (If poor)
Presentation Quality					
Language and Style					
Illustrations					
Used (diagrams, tables, etc.)					
Conceptual Clarity					
Check Your Progress Questions					
Suggested Readings - OER					

5. Any other comments:
