

PYTHON PROGRAMMING – STUDENT MATERIAL

MODULE 1: GETTING STARTED WITH PYTHON

What is Python?

Python is a **high-level, interpreted, and beginner-friendly** programming language created by **Guido van Rossum** in 1991. It focuses on readability and simplicity. Python was designed with the goal of making programming:

- Easy to learn
- Easy to read
- Easy to write

Unlike many other programming languages, Python uses **simple English-like syntax**, which makes it ideal for beginners as well as professionals.

Why is Python called a High-Level Language?

- It is close to human language
- You don't need to manage memory manually
- You can focus more on problem-solving than machine-level details

Key Features

- **Easy and readable syntax**

Python code is clean and easy to understand. Compared to other languages, Python uses fewer lines of code.

- **Interpreted language**

Python executes code **line by line**. No need to compile before running

- **Cross-platform**

Python programs can run on:

1. Windows
2. Linux
3. macOS

The same Python code works on all platforms without modification.

- **Open source** Python is **free to use** and source code is publicly available Anyone can contribute to its development

- Supports **multiple programming styles** (procedural, OOP, functional)

- **Dynamically typed**

No need to declare variable types Python decides the data type at runtime

- **Large standard library**

Python comes with built-in modules for:

- File handling
- Math operations
- Date and time
- Networking

This reduces the need for external libraries.

Applications of Python

- Web Development (Django, Flask)
- Data Science & Analytics

- Machine Learning & AI
- Automation & Scripting
- Game Development

Python File Extension

- .py

Python source files use the .py extension.
For example: hello.py

Comments

- Single-line comment: # comment
This is a single-line comment
- Multi-line comment: ''' comment '''
''' This is a multi-line comment '''

Installation & Setup

Steps to Install Python To begin programming in Python

- Go to the official website: python.org
- Download the latest version for your operating system.
- Launch the installer and ensure you check the "Add Python to PATH" option.
- Open your terminal or command prompt and type:

python --version - to verify the installation.

Setting Up VS Code for Python

To write Python code effectively, Visual Studio Code (VS Code) is a powerful and lightweight IDE:

- Download VS Code
<https://code.visualstudio.com>.
- Install the Python extension by Microsoft from the Extensions tab.

- Use Ctrl+Shift+P to open Command Palette and select the Python interpreter.
- Create a new file with a .py extension and begin writing code.
- Use the built-in terminal or the run button to execute your code.

Working with Virtual Environments

- Understanding Virtual Environments

A virtual environment is an isolated Python workspace that helps manage dependencies for different projects. It ensures that packages for one project do not interfere with another.

- To create a new environment:

python -m venv myenv

Activating the Environment Windows:

myenv\Scripts\activate

Sample Programs

- ❖ Program 1: Display a message: Welcome to Python

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

- ❖ Program 2: Addition of Two Numbers

```
a = 5
b = 3
sum = a + b
print("Sum:", sum)
```

MODULE 2:

PYTHON

FUNDAMENTALS

Keywords

Keywords are special reserved words in Python that have a specific meaning. They guide how Python code is written and understood. Python keywords are case-sensitive. Python has over 35 keywords in total, and the exact number may vary with different versions. Below is a list of the most commonly used keywords:

if else while import

except None is elif

def class True break

In for return try

False continue

Identifiers

Identifiers are the names given to variables, functions, classes, and other objects in Python.

Rules for Naming Identifiers

These are the rules Python follows for naming identifiers:

1. Can only contain letters (A–Z or a–z), digits (0–9), and underscores (_).
2. Must begin with a letter or an underscore.
3. Can't use keywords as names.
4. Python is case-sensitive: Sum and sum will be treated as two different identifiers.

Variables

Variables are containers for storing data values. There is no need to declare the type like in other

languages (e.g., int, float). Python is **dynamically typed**, meaning the data type of a variable is figured

out at runtime.

Memory Management

- Python handles memory automatically, but here's how it works behind the scenes **Memory Allocation**: When a value is assigned to a variable, Python stores it in memory. The variable name is just a label that points to the actual object (value) in memory. **Garbage Collection**: Python has a built-in Garbage Collector. It automatically cleans up memory by deleting objects that are no longer being used. There is no need to manually free up memory like in some other languages.

Input & Output

• input() Function

It is used to get information from the user. Always takes input as string, even if a number is typed. The input is stored in a variable.

```
name = input("Enter your name? ")
print("Hello", name)
```

To input a number:

```
num = int(input("Enter a number: "))
print("You entered:", num)
```

- **print() Function**

It is used to display text, results, or messages. It can show variables, numbers, strings, or even math expressions.

```
print("Python is fun!")
x = 10
print("Value of x:", x)
```

Data Types

Python data types define the type of value a variable can hold. Data types tell Python what kind of data is being used—like numbers, text, or a list of items—so it knows how to handle it.

- **Numeric Types** :Numeric data types help to work with numbers.

- a. int: Whole numbers (e.g., 5, -3,0)
- b. float: Decimal numbers (e.g., 1.23, -0.15)
- c. complex: Numbers with real and imaginary parts (e.g., 2 + 3j)

- **String (str)** A sequence of characters in single or double quotes (e.g., "hello", 'Python').

Used to store and manipulate text.

- **Boolean (bool)** Has two values: True and False. Used in decision-making and comparisons (e.g., 3 > 2 gives True).

- **None Type (None)** Represents no value or “nothing”. Used when a variable is empty or a

function returns nothing.

Type Conversion

Type conversion means changing a value from one data type to another. Python supports two types of type conversion:

- **Implicit Type Conversion**

(Automatic) Python automatically converts one data

type to another during an operation if needed. It happens without user involvement, usually when combining different numeric types.

```
x = 5 #int
y = 2.5 # float
z = x + y # float
```

- **Explicit Type Conversion (Type Casting)** This is when one type is manually

converted to another using built-in functions.

```
a = "10"
b = int(a) # Converts string to int
```

Type Checking

To check the type of a variable.

- **type()** Returns the actual type of a value or variable.

```
x = 5
print(type(x)) # <class 'int'>
```

- **isinstance()**

Checks if a value belongs to a specific type (returns True or False).

```
x = 3.5
print(isinstance(x, float)) # True

print(isinstance(x, int)) # False
```

- **id()** The id() function in Python returns the unique identity of an object, which is its memory address in the form of an integer. It helps to check if two variables refer to the same object in memory.

```
a = 5
b = a
print(id(a)) # e.g.,
140711248170384
print(id(b)) # Same as a, because b points to
the same object
```

MODULE 3:

OPERATORS & EXPRESSIONS

An operator is a symbol or keyword in Python used to perform operations on values or variables. It tells Python to do operations like add, compare, assign, or check values.

Types of Operators

- Arithmetic (+, -, *, /, %, //, **)
- Assignment (=, +=, -=)
- Comparison (==, !=, >, <)
- Logical (and, or, not)
- Bitwise (&, |, ^, ~, <<, >>)
- Membership (in, not in)
- Identity (is, is not)

1.Arithmetic operators

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

Operator	Meaning	Example (a = 10, b = 3)	Result
+	Addition	a + b	13
-	Subtraction	a - b	7
*	Multiplication	a * b	30
/	Division	a / b	3.33
//	Floor Division	a // b	3
%	Modulus	a % b	1
**	Exponent	a ** b	1000

```
x = 10
y = 4
sum = x + y
print("Sum:", sum)
diff = x - y
print("Difference:", diff)
mul = x * y
print("Product:", mul)
quo = x / y
print("Quotient:", quo)
floor_div = x // y
print("Floor Division:", floor_div)
mod = x % y
print("Modulus:", mod)
exp = x ** y
print("Exponentiation:", exp)
```

2.Assignment operator

Assignment operators are used to assign values to variables.

Operator	Meaning	Example
=	Assign value	x = 5
+=	Add and assign	x += 2 (x = x + 2)
-=	Subtract and assign	x -= 3 (x = x - 3)
*=	Multiply and assign	x *= 4 (x = x * 4)
/=	Division and assign	x /= 3 (x = x / 3)
//=	Floor Division and assign	x //= 2 (x = x // 2)
%=	Modulus and assign	x %= 2 (x = x % 2)
**=	Exponent and assign	x **= 3 (x = x ** 3)

Code:

```
x = 10
x += 5
print("After adding 5, x is:", x)
x -= 3
print("After subtracting 3, x is:", x)
x *= 2
print("After multiplying by 2, x is:", x)
x /= 4
print("After dividing by 4, x is:", x)
x //= 2
print("After floor dividing by 2, x is:", x)
x %= 2
print("After taking modulus with 2, x is:", x)
x **= 3
print("After exponentiation by 3, x is:", x)
```

3.Comparison Operator

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

Operator	Meaning	Example (a = 5, b = 3)
==	Equal to	a == b → False
!=	Not equal to	a != b → True
>	Greater than	a > b → True
<	Less than	a < b → False
>=	Greater than or equal to	a >= b → True
<=	Less than or equal to	a <= b → False

Code: check eligibility to vote

```
age = int(input("Enter your age: "))
if age >= 18:
    print("You are eligible to vote")
else:
    print("You are not eligible to vote")
```

4.Logical Operators

Logical operators are used to combine multiple conditions and return a Boolean result.

Operator	Meaning	Example (a = 5, b = 3)
and	True if both are true	a > 0 and b < 5 → True
or	True if at least one is true	a < 0 or b < 5 → True
not	Reverses the result	not(a > b) → False

AND operator

Returns True if both conditions are True

```
age = 20
marks = 75
if age >= 18 and marks >= 50:
    print("Eligible for admission")
else:
    print("Not eligible for admission")
```

OR operator

Returns True if at least one condition is True

```
username = input("Enter username: ")
if username == "admin" or username == "root":
    print("Access granted")
else:
    print("Access denied")
```

NOT operator

Reverses the condition

```
is_raining = False

if not is_raining:
    print("You can go outside")
else:
    print("Take an umbrella")
```

Combining and,or,not

```
age = 25
has_id = True
if age >= 18 and not has_id:
    print("Bring your ID proof")
elif age >= 18 and has_id:
    print("Entry allowed")
else:
    print("Entry not allowed")
```

5.Bitwise operator

Bitwise operators perform operations on individual bits of binary representations of integers.

Name	Symbol	Usage	What it does
Bitwise And	&	a & b	Returns 1 only if both the bits are 1
Bitwise Or		a b	Returns 1 if one of the bits is 1
Bitwise Not	~	~a	Returns the complement of a bit
Bitwise Xor	^	a ^ b	Returns 0 if both the bits are same else 1
Bitwise Left shift	<<	a << n	Shifts a towards left by n digits
Bitwise Right shift	>>	a >> n	Shifts a towards right by n digits

Code:

```
a = 5    # 0101
b = 3    # 0011

print(a & b)    # AND
print(a | b)    # OR
print(a ^ b)    # XOR
```

6.Membership Operators

Membership operators are used to check whether a value exists in a sequence (like a string, list, tuple, or set). There are two membership operators in Python - **in** and **not in**. **in** operator returns **True** if value is found. **not in** operator returns **True** if value is not found.

1. Using in Operator

```
fruits = ["apple", "banana", "mango"]
print("apple" in fruits) # True
print("grape" in fruits) # False
```

2. Using not in Operator

```
numbers = [1, 2, 3, 4, 5]
print(6 not in numbers) # True
print(3 not in numbers) # False
```

7.Identity Operators

Identity operators are used to check whether two variables refer to the same object in memory, not just equal values. There are two identity operators in Python - **is** and **is not**. **is** operator returns True if both variables point to the same object. **is not** operator returns True if they point to different objects.

1. Using is Operator

```
a = [1, 2, 3]
b = a
c = [1, 2, 3]
print(a is b) # True (same object)
print(a is c) # False (different object)
```

2. Using is not Operator

```
x = 10
y = 20
print(x is not y) # True
```

Difference Between is and == in Python

Operator	Meaning	Example	Result
==	Compares values	a == b	True
is	Checks memory reference	a is b	False

Difference between = and == in Python

Operator	Meaning	Example
==	Equality check	x==10
=	Assignment operator	x=10

Difference between in and is operator

is Operator (Identity)

- Checks whether two variables refer to the same object
- Compares memory location, not value
- Works with any object
- Returns True or False

in Operator (Membership)

- Checks **whether a value exists** in a sequence
- Used with **list, tuple, string, set, dictionary**
- Compares **values**, not memory location
- Returns **True or False**

MODULE 4:

CONTROL STRUCTURES

Control structures in Python are programming constructs that manage the flow of execution based on conditions or repetition. They allow decisions to be made (if, if-else), tasks to be

repeated (for, while), and the flow to be altered (break, continue, pass).

Types of Control structures

1. Conditional statement

2. Looping statement

3. Jumping statement

1. Conditional statements

Conditional statements are used to make decisions in Python based on whether a condition is True or False.

1. If statement

Executes a block of code only if the condition is true.

Syntax:

if condition:

block of code to execute

Code:

```
age = 18
if age >= 18:
    print("You are eligible to vote")
```

What is Indentation?

- **Indentation** is the **leading whitespace** (spaces or tabs) at the beginning of a line of code.

- In Python, **indentation defines blocks of code** instead of curly braces {} like in other languages.
- It tells Python **which statements belong to which control structure**.

Why is Indentation Important?

- Python uses indentation to **group statements**.
- Incorrect indentation will cause an **IndentationError**.

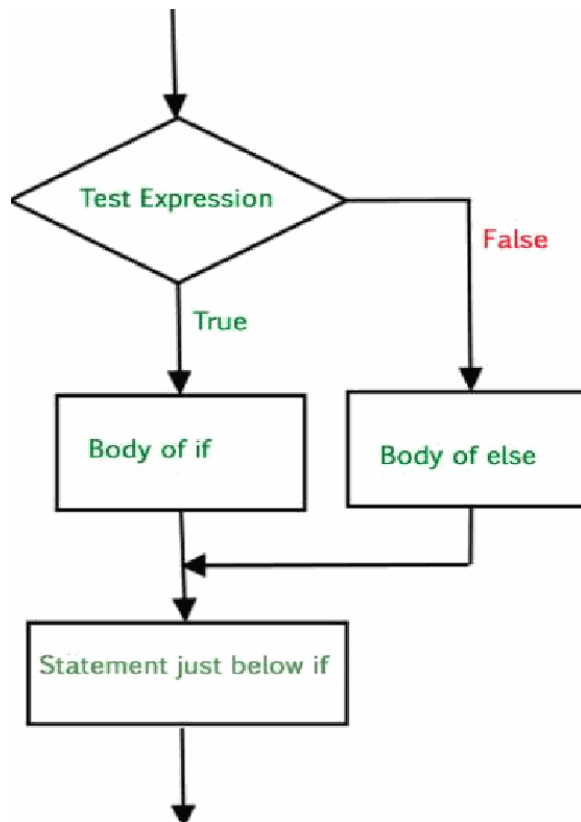
```
if True:
print("Hello") # ✗ IndentationError
```

Correct version:

```
if True:
    print("Hello") # ✔ 4 spaces
```

2. if-else statement

Checks a condition and runs one block if true, another if false. In Python, a **suite** is a group of one or more statements that form the body of a compound statement like if, for, while, def, or class. It is typically defined by indentation.



Code:

```
age = 16
if age >= 18:
    print("You are eligible to vote")
else:
    print("You are not eligible to vote")
```

3..if-else-elif statement

Used to check **multiple conditions**.

```
marks = 75
if marks >= 90:
    print("Grade A")
elif marks >= 75:
    print("Grade B")
elif marks >= 50:
    print("Grade C")
else:
    print("Fail")
```

4.Nested-if statement

An if statement inside another if statement.

```
num = int(input("Enter a number: "))
if num > 0:
    if num % 2 == 0:
        print("The number is positive and even.")
    else:
        print("The number is positive but odd.")
else:
    print("The number is not positive.")
```

2.Looping statements

Looping allows you to repeat a block of code multiple times. Python provides two main types of loops: **for** and **while**.

a) for loop

Used to iterate over a sequence (like list, string, or range). Syntax:

```
for variable in sequence:
    # block of code
```

Code:

```
fruits = ["apple", "banana", "mango"]
for fruit in fruits:
    print(fruit)
```

Output:

```
apple
banana
mango
```

range() in Python

range() is a built-in Python function that generates a sequence of numbers.

It is often used in for loops to control the number of iterations. Syntax:

Parameters of range

Parameter	Description	Example
start (optional)	Starting number of the sequence	range(1, 5) → starts at 1
stop	End number (exclusive)	range(5) → 0 to 4
step (optional)	Difference between consecutive numbers	range(1, 10, 2) → 1,3,5,7,9

Examples:

a) range(stop)

```
for i in range(5): # 0 to 4
    print(i)
```

Output:

```
0
1
2
3
4
```

b) range(start, stop)

```
for i in range(1, 6): # 1 to 5
    print(i)
```

Output:

```
1
2
3
4
5
```

c) range(start, stop, step)

```
for i in range(2, 11, 2): # 2 to 10, step 2
    print(i)
```

Output:

```
2
4
6
8
10
```

Nested for loop:

A nested for loop means a loop inside another loop. It's useful for working with patterns, grids, tables, etc. The outer loop runs first. For each iteration of the outer loop, the inner loop runs completely.

b) while Loop

Executes as long as the condition is True
Example: Prints numbers from 1 to 5

```
i = 1
while i <= 5:
    print(i)
    i += 1
```

Factorial Using while Loop

```
num = int(input("Enter a number: "))
fact = 1
while num > 0:
    fact *= num
    num -= 1
print("Factorial =", fact)
```

else with loops:

Python allows an else block to be used with both for and while loops. The else part runs only if the loop finishes normally (i.e., not interrupted by break)

```
for i in range(3):
    print(i)
else:
    print("Loop finished without break")
```

3. Jumping Statements

Jumping statements are used to change the normal flow of loops or control structures. Python provides three main jumping statements.

1. break
2. continue
3. Pass

1. break Statement

- Immediately terminates the loop
- Control moves outside the loop

```
for i in range(1, 10):
    if i == 5:
        break
    print(i)
```

Output:

```
1
2
3
4
```

2. continue

Purpose: Skips the current iteration of the loop and moves to the next iteration.

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

Output:

```
1  
2  
4  
5
```

When i is 3, the remaining code in that loop cycle is skipped.

3.pass

Purpose: Acts as a placeholder where a statement is required syntactically, but no action is needed.

```
for i in range(1, 4):  
    if i == 2:  
        pass  
    print(i)
```

MODULE 5:

FUNCTIONS IN PYTHON

A **function** is a block of reusable code that performs a specific task. Functions help make programs **modular, readable, and reusable**. Python uses indentation to define code blocks. All code inside a function must be consistently indented. Instead of writing the same code again and again, we place it inside a function and reuse it whenever needed.

Example (Without function):

```
print("Welcome")
print("Welcome")
print("Welcome")
```

Example (With function):

```
def welcome():
    print("Welcome")

welcome()
welcome()
welcome()
```

Why Do We Use Functions?

Functions are used because they:

- Reduce **code repetition**
- Improve **readability**
- Make programs **easy to debug**
- Help in **modular programming**

Types of Functions

- Built-in Functions: len(), max(), range(), etc.
- User-Defined Functions: Custom functions with def

- Lambda Functions: Anonymous functions using lambda

Built-in Functions in Python

Built-in functions are predefined functions that come with Python and are always available without importing any module. They help perform common tasks like type conversion, data handling, math operations, I/O, etc.

Functions provided by Python. Examples:

- print()
- len()
- type()
- max()
- min()

```
Example 1:
name = input("Enter your name: ")
print(name)
```

```
Example 2:
x = 10
print(type(x))
```

User-Defined Functions in Python

A user-defined function is a function that a programmer creates to perform a specific task. These help in:

- Breaking large problems into smaller, manageable parts
- Avoiding repetition (code reuse)

Enhancing clarity and maintainability

Defining a Function

A function definition is the part of the code where you declare and create a function using the

def keyword. It tells Python what the function will do when called.

```
def function_name(parameters):
    #function body
    return value #optional
```

- def → keyword to define a function
- function_name → name of the function
- parameters → values passed to the function (optional)
- return → sends result back (optional)

Example:

```
def greet():
    print("Hello Python")
```

Calling the function:

A function is executed by calling its name.

```
greet()
```

return statement

return statement is used to send a result from the function to the caller. Can return a value or

multiple values as tuple.

```
def stats(a, b):
    return a + b, a * b
s, p = stats(2, 4)
print("Sum:", s, "Product:", p)
```

Docstring in Functions

A docstring is a string used to document what a function does. It's placed right after the function definition line.

```
def square(n):
    """Returns the square of a number"""
    return n * n
help(square) # Shows the docstring
```

Lambda (Anonymous) Functions in Python

A lambda function is a small anonymous function (function without a name). It is used when a function is needed for a short period and for simple operations. It is nameless (often assigned to a variable), Inline (used where short functions are required), Limited to a single expression. The result of the expression is returned automatically (no return keyword needed).

```
lambda arguments : expression
```

a) Uses the keyword lambda b) Can have **any number of arguments** c) Contains **only one expression** d) Expression result is automatically returned

Example:

```
square = lambda x: x * x
print(square(5))
```

Types of Arguments in Python

In Python, function arguments define how values are passed to a function. Understanding argument types helps you write flexible and reusable code.

● **Positional Arguments** These are matched by position — the first argument goes to the first parameter, the second to the second, and so

on.

```
def greet(name, age):
    print(f"Name: {name}, Age: {age}")
greet("Achu", 25) # Positional order matters
```

● **Keyword Arguments** These arguments can be passed using parameter names (keywords). This allows you to specify arguments in any order.

```
def greet(name, age):
    print(f"Name: {name}, Age: {age}")
greet(age=25, name="Shivaay") # Same result as before
```

☉ **Default Arguments** Default values can be assigned to parameters. If no argument is passed, the default is used.

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

☉ **Variable-Length Arguments** **Variable length arguments** allow a function to accept **any number of arguments** instead of a fixed number.

They are used when we **do not know in advance** how many values will be passed to a function.

Types of Variable Length Arguments

Python supports **two types**:

1. ***args** → Non-keyword variable length arguments
2. ****kwargs** → Keyword variable length arguments

1. *args (Non-Keyword Arguments)

- ***args** allows passing multiple values
- Stored as a tuple
- Name args is conventional (can be any name)

```
def total(*numbers):
    sum = 0
    for n in numbers:
        sum += n
    return sum
print(total(10, 20, 30))
Output:60
```

2. **kwargs (Keyword Arguments)

- ****kwargs** allows passing key-value pairs
- Stored as a dictionary
- Name kwargs is conventional

```
def student_info(**data):
    for key, value in data.items():
        print(key, ":", value)
student_info(name="Ravi", age=18, marks=85)
```

Difference between *args and **kwargs

1. ***args** is used to pass a variable number of non-keyword (positional) arguments, while ****kwargs** is used to pass keyword arguments.
2. ***args** stores values as a tuple, whereas ****kwargs** stores values as a dictionary.
3. ***args** uses a single asterisk (*), while ****kwargs** uses a double asterisk (**).
4. In ***args**, values are accessed using index numbers, whereas in ****kwargs**, values are accessed using keys.
5. ***args** depends on the order of arguments, while ****kwargs** does not depend on order.

Difference between print and return

1. **print** is used to display output on the screen, whereas **return** is used to send a value back to the calling function.
2. **print** does not pass any value to the program, whereas **return** passes a value for further use.
3. **print** can be used anywhere in a program, whereas **return** is used only inside a function.

4. print does not end function execution, whereas return ends the function immediately.
5. The output of print cannot be stored in a variable, whereas the value returned by return can be stored in a variable.

Difference between Lambda Function and Normal Function

1. A **lambda function** is an **anonymous function** (has no name), whereas a **normal function** has a **defined name**.
2. A **lambda function** is written in **one line**, whereas a **normal function** can have **multiple lines** of code.
3. A **lambda function** contains **only one expression**, whereas a **normal function** can contain **multiple statements**.
4. A **lambda function** does **not use the return keyword**, whereas a **normal function** uses return to send a value back.
5. A **lambda function** is mainly used for **small, temporary tasks**, whereas a **normal function** is used for **complex and reusable logic**.

Variable Scope in Python

Scope refers to the region in a program where a variable is recognized and can be accessed. Python uses different scopes to manage variables, such as local, global, and enclosing.

1. Local and Global Variables

Local Variable Declared inside a function. Accessible only within that function.

```
def show():
    x = 10 # local variable
    print(x)
show()
print(x)# shows error since it is accessible
only inside function
```

2.Global Variable

Declared outside any function. Accessible inside and outside functions.

```
x = 10 # Global variable
def show():
    print("x inside function:", x)
show()
print("x outside function:", x)
```

Using global Keyword

Used to modify a global variable inside a function.

```
x = 5
def update():
    global x
    x = 10
update()
print(x)
```

3. nonlocal Keyword

Used inside nested functions to modify a variable from the enclosing (non-global) scope.

```
def outer():
    x = "outer"
    def inner():
        nonlocal x
        x = "inner"
    inner()
    print("x after inner():", x)
outer()
```

LEGB Rule

Python resolves variable names using the LEGB rule Python searches for a variable in the following order:

1. Local
2. Enclosing
3. Global

4. Built-in

```
Example:
x = "global"
def outer():
    x = "enclosing"
    def inner():
        x = "local"
        print("Inner x:", x)
    inner()
    print("Outer x:", x)
outer()
print("Global x:", x)
```

Recursion

Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem.

Two Important Parts of Recursion

Every recursive function must contain:

1. **Base condition** – stops the recursion
2. **Recursive call** – function calls itself

Without a base condition, recursion leads to **infinite recursion**. Syntax

```
def function_name():
    if condition: # base condition
        return value
    return function_name() # recursive call
```

Example: Factorial of a Number

```
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n - 1)
print(factorial(5))
```

When NOT to Use Recursion

- Very Large Input Sizes
- When Memory Efficiency Matters
- When You Need Better Performance

MODULE 6: DATA STRUCTURES IN DEPTH

A data structure is a method of organizing, storing, and managing data in a computer so that it can be used efficiently.

Why is a Data Structure Used?

Data structures are used because they:

1. Organize data in a structured manner
2. Allow fast access and modification of data
3. Improve program efficiency and performance
4. Help manage large amounts of data easily
5. Make programs easy to understand and maintain

Categorization of Data Structures

Data structures can be **categorized in different ways** based on how data is stored and organized.

1. Primitive Data Structures

These are **basic data types** that store simple values. Examples:

- Integer (int)
- Float (float)
- Character (char)
- Boolean (bool)

2. Non-Primitive Data Structures

These store **collections of data**. They are further divided into:

a) Linear Data Structures

Data elements are arranged in a **sequential order**. Examples:

- List
- Tuple
- String
- Array

What is a Sequence?

A **sequence** is an **ordered collection of elements** where each element has a fixed position.

Key Characteristics of Sequences

- Elements are stored in **order**
- Each element has an **index**
- Supports **indexing and slicing**
- Can be **iterated** using loops
- Can contain **duplicate values**

Built-in Sequence Types in Python

Python provides several built-in sequence types:

- String (str)
- List (list)
- Tuple (tuple)

☉ Multiline Strings Use triple quotes for multi-line strings. `message = "This is a multiline String."`

☉ **String Indexing** Strings are indexed (starting from 0).

```
word = "Python"
print(word[0]) # Output: P
print(word[-1]) # Output: n
```

String Slicing

- Extract a part (substring) of the string using slice notation

string[start:stop:step].

```
text = "Hello, World!"
print(text[0:5]) # Output: Hello
print(text[5:])
# Output: Hello
print(text[7:]) # Output: World!
print(text[:])
# Output: Hello, World!
print(text[-6:-1]) # Output: World
print(text[::-1]) # Output: !dlroW ,olleH
print(text[:2]) # Output: Hlo ol!
print(text[-1])
# Output: !
print(text[3:])
# Output: Hel
print(text[1:10:2]) # Output: el,Wr
```

Strings are Immutable

Once created, string contents cannot be changed; you must create a new string. Eg: word =

"Hello" # word[0] = 'J' Error

String Concatenation

Using + Operator The simplest way to join strings is using the + operator.

```
Example:
first = "Hello"
second = "World"
result = first + " " + second
print(result) # Output: Hello World
```

Using += Operator (Append to a string)

A new content can be added to an existing string using +=

```
Example:
message = "Good"
message += " Morning"
print(message) # Output: Good Morning
```

Using join() Method

Best for joining a list of strings with a separator.

```
words = ["Python", "is", "awesome"]
sentence = " ".join(words)
print(sentence) # Output: Python is awesome
words = ["Python", "is", "awesome"]
sentence = " ".join(words)
print(sentence) # Output: Python is awesome
```

Using f-strings (Python 3.6+)

The most modern and readable way to concatenate strings and variables.

```
Example:
name = "Veena"
age = 30
print(f"{name} is {age} years old") # Output:
Veena is 30 years old
```

Important: All parts must be strings Strings can't be concatenated with numbers directly. Use str() to convert:

```
age = 25
print("Age: " + str(age)) # ✓ Correct
# print("Age: " + age) # ✗ Error
TypeError
```

LISTS (list) – Linear Data Structure

What is a List? A list is an ordered, mutable collection of elements.

- Lists can store multiple values
- Lists can store different data types
- Lists allow duplicate values
- Lists support indexing, slicing, and iteration

```
numbers = [1, 2, 3, 4]
mixed = [10, "Python", 3.5, True]
```

Creating Lists

```
empty_list = []
names = ["Alice", "Bob", "Charlie"]
marks = [85, 90, 78, 92]
```

List Indexing Indexing starts from 0.

```
numbers = [10, 20, 30, 40]
print(numbers[0]) # 10
print(numbers[2]) # 30
print(numbers[-1]) # 40
```

List Slicing Used to extract part of a list.

```
numbers = [10, 20, 30, 40, 50]
print(numbers[1:4]) # [20, 30, 40]
print(numbers[:3]) # [10, 20, 30]
print(numbers[::2]) # [10, 30, 50]
print(numbers[::-1]) # [50, 40, 30, 20, 10]
```

Lists Are Mutable (Very Important) You can change list elements after creation.

```
numbers = [1, 2, 3]
numbers[0] = 10
print(numbers) # [10, 2, 3]
```

This is the biggest difference between lists and strings/tuples.

Adding Elements to a List

append() - Add at the End

```
nums = [1, 2]
nums.append(3)
print(nums) # [1, 2, 3]
```

insert() – Add at Specific Index

```
nums = [1, 3]
nums.insert(1, 2)
print(nums) # [1, 2, 3]
```

extend() – Add Multiple Elements

```
nums = [1, 2]
nums.extend([3, 4, 5])
print(nums) # [1, 2, 3, 4, 5]
```

Removing Elements from a List

remove() – Remove by Value

```
nums = [1, 2, 3, 2]
nums.remove(2)
print(nums) # [1, 3, 2]
```

Removes only the first occurrence.

pop() – Remove by Index

```
nums = [10, 20, 30]
nums.pop(1)
print(nums) # [10, 30]
nums.pop() # Removes last element
```

clear() – Remove All Elements

```
nums = [1, 2, 3]
nums.clear()
print(nums) # []
```

Finding Elements

index() – Get Index of Value

```
nums = [10, 20, 30]
print(nums.index(20)) # 1
```

count() – Count Occurrences

```
nums = [1, 2, 2, 3]
print(nums.count(2)) # 2
```

Sorting and Reversing

sort()

```
nums = [3, 1, 4, 2]
nums.sort()
print(nums) # [1, 2, 3, 4]
```

Descending order:

```
nums.sort(reverse=True)
```

reverse()

```
nums = [1, 2, 3]
nums.reverse()
print(nums) # [3, 2, 1]
```

Iterating Over a List

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

Membership Operators

```
nums = [1, 2, 3]
print(2 in nums) # True
print(5 not in nums) # True
```

List Concatenation and Repetition

```
a = [1, 2]
b = [3, 4]

print(a + b) # [1, 2, 3, 4]
print(a * 2) # [1, 2, 1, 2]
```

Nested Lists Lists can contain other lists.

```
matrix = [
    [1, 2],
    [3, 4]
]
print(matrix[0][1]) # 2
```

Common Use Cases of Lists

- Storing student marks
- Dynamic data collection
- Implementing stacks & queues
- Iterating over datasets

Summary – Lists

- Ordered and mutable
- Supports indexing & slicing
- Can store mixed data types
- Most commonly used data structure in Python

TUPLES (tuple) – Linear Data Structure

What is a Tuple?

A **tuple** is an **ordered, immutable collection of elements**.

- Tuples store **multiple values**
- Tuples allow **duplicate values**
- Tuples can store **different data types**
- **Once created, tuple values cannot be changed**

```
point = (10, 20)
colors = ("red", "green", "blue")
```

Creating Tuples

Normal Tuple

```
numbers = (1, 2, 3, 4)
```

Single-Element Tuple (Very Important)

A comma is **mandatory**.

```
t1 = (5) # ✗ Not a tuple
t2 = (5,) # ✓ Tuple
```

Tuple Without Parentheses (Tuple Packing)

```
data = 10, 20, 30
print(type(data)) # <class 'tuple'>
```

Tuple Indexing

Indexing works the same as lists.

```
colors = ("red", "green", "blue")
print(colors[0]) # red
print(colors[-1]) # blue
```

Tuple Slicing

```
numbers = (10, 20, 30, 40, 50)
print(numbers[1:4]) # (20, 30, 40)
print(numbers[:3]) # (10, 20, 30)
print(numbers[::-1]) # (50, 40, 30, 20, 10)
```

Tuples Are Immutable (Key Concept)

You **cannot modify** a tuple after creation.

```
colors = ("red", "green", "blue")
# colors[0] = "yellow" ✗ Error
```

Why Immutability Matters?

- Data safety
- Faster performance
- Prevents accidental changes

Tuple Methods

Tuples have **very few methods** because they are immutable.

count()

```
nums = (1, 2, 2, 3)
print(nums.count(2)) # 2
```

index()

```
nums = (10, 20, 30)
print(nums.index(20)) # 1
```

Tuple Iteration

```
colors = ("red", "green", "blue")
for color in colors:
    print(color)
```

Tuple Unpacking

Assign tuple values to variables.

```
point = (10, 20)
x, y = point
print(x) # 10
print(y) # 20
```

Tuple vs List (Important Comparison)

Feature	List	Tuple
Mutable	Yes	No
Syntax	[]	()
Performance	Slower	Faster
Methods	Many	Few
Safety	Less	More

When to Use Tuples?

- ✓ Fixed data (coordinates, RGB values)
- ✓ Data that should not change
- ✓ Function return values
- ✓ Dictionary keys (tuples are hashable)

Converting Between List and Tuple

```
nums_list = [1, 2, 3]
nums_tuple = tuple(nums_list)
nums_list2 = list(nums_tuple)
```

Nested Tuples

```
data = ((1, 2), (3, 4))
print(data[1][0]) # 3
```

Summary – Tuples

- Ordered and **immutable**
- Faster and safer than lists
- Limited methods
- Ideal for fixed, constant data

b) Non-Linear Data Structures

SET (set) – Collection Data Structure

What is a Set?

A **set** is an **unordered collection of unique elements**.

- Does **not allow duplicate values**
- Elements are **unordered**
- Elements are **mutable** (can add/remove)
- **Indexing and slicing are NOT supported**

```
numbers = {1, 2, 3, 4}
```

Key Characteristics of Sets

- No duplicates
- No fixed order
- Faster membership checking
- Useful for mathematical operations

```
nums = {1, 2, 2, 3}
print(nums) # {1, 2, 3}
```

Creating a Set

Using Curly Braces

```
colors = {"red", "green", "blue"}
```

Using set() Function

```
nums = set([1, 2, 3, 4])
```

Empty Set (Important)

```
empty = {} # ✗ This creates a dictionary  
empty_set = set() # ✓ Correct
```

No Indexing in Sets

```
nums = {10, 20, 30}  
# nums[0] ✗ TypeError
```

Use loops or membership checks instead.

Adding Elements to a Set

add() – Add Single Element

```
nums = {1, 2}  
nums.add(3)  
print(nums) # {1, 2, 3}
```

update() – Add Multiple Elements

```
nums = {1, 2}  
nums.update([3, 4, 5])  
print(nums) # {1, 2, 3, 4, 5}
```

Removing Elements from a Set

remove() – Raises Error if Not Found

```
nums = {1, 2, 3}  
nums.remove(2)
```

discard() – No Error if Not Found

```
nums.discard(5) # Safe
```

pop() – Removes Random Element

```
nums = {1, 2, 3}  
nums.pop()
```

clear() – Remove All Elements

```
nums.clear()
```

Set Operations (Very Important)

Union

```
A = {1, 2, 3}  
B = {3, 4, 5}  
print(A | B) # {1, 2, 3, 4, 5}  
print(A.union(B))
```

Intersection

```
print(A & B) # {3}  
print(A.intersection(B))
```

Difference

```
print(A - B) # {1, 2}  
print(A.difference(B))
```

Symmetric Difference

```
print(A ^ B) # {1, 2, 4, 5}  
print(A.symmetric_difference(B))
```

Membership Testing (Fast)

```
nums = {1, 2, 3}  
print(2 in nums) # True  
print(5 in nums) # False
```

Iterating Over a Set

```
for item in nums:  
    print(item)
```

Order is **not guaranteed**.

Common Use Cases of Sets

- Removing duplicates
- Membership checking
- Finding common elements
- Mathematical set operations

Summary – Set

- Unordered
- No duplicates
- Mutable
- No indexing
- Very fast lookup

FROZENSET (Brief Note)

What is a Frozenset?

A **frozenset** is an **immutable version of a set**.

- Cannot add or remove elements

- Unordered
- No duplicates
- Supports set operations

```
nums = frozenset([1, 2, 3])
```

Immutability of Frozenset

```
# nums.add(4) ✗ Error
# nums.remove(2) ✗ Error
```

Why Use Frozenset?

- ✓ When you need a **set that should not change**
- ✓ Can be used as **dictionary keys**
- ✓ Safer than normal sets

```
data = {
    frozenset([1, 2]): "Group A",
    frozenset([3, 4]): "Group B"
}
```

Set vs Frozenset

Feature	Set	Frozenset
Mutable	Yes	No
Unordered	Yes	Yes
Allows duplicates	No	No
Dictionary key	No	Yes

Final Summary (For Students)

- **Set** → Mutable, unique, unordered
- **Frozenset** → Immutable version of set
- Use sets for **dynamic unique data**
- Use frozenset for **fixed unique data**

DICTIONARY (dict) – Key–Value Data Structure

What is a Dictionary?

A **dictionary** is a **collection of key–value pairs**.

- Each value is accessed using a **unique key**
- Dictionaries are **unordered** (logically)
- **Keys must be unique**
- Values can be of any data type
- Keys must be **immutable** (string, number, tuple)

```
student = {
    "name": "Rahul",
    "age": 20,
    "course": "Python"
}
```

Key Characteristics of Dictionary

- Data is stored as key : value
- Fast data access using keys
- Mutable (values can be changed)
- No indexing or slicing like lists

Creating a Dictionary

Using Curly Braces

```
person = {
    "name": "Anita",
    "city": "Delhi",
    "age": 25
}
```

Using dict() Function

```
person = dict(name="Anita", city="Delhi", age=25)
```

Empty Dictionary

```
data = {}
```

Accessing (Retrieving) Dictionary Values

Using Key

```
student = {"name": "Rahul", "age": 20}
print(student["name"]) # Rahul
```

Accessing a non-existing key causes an error:

```
# print(student["marks"]) ✕ KeyError
```

Using get() Method (Recommended)

```
print(student.get("age")) # 20
print(student.get("marks")) # None

print(student.get("marks", 0)) # Default value
```

Updating Dictionary Values

Update Existing Key

```
student["age"] = 21
```

Add New Key–Value Pair

```
student["marks"] = 85
```

Using update() Method

```
student.update({"city": "Chennai", "grade": "A"})
```

Removing Elements from Dictionary

pop() – Remove by Key

```
student.pop("grade")
```

popitem() – Remove Last Inserted Item

```
student.popitem()
```

del Keyword

```
del student["city"]
```

clear() – Remove All Items

```
student.clear()
```

Dictionary Methods

keys()

```
student.keys()
```

values()

```
student.values()
```

items()

```
student.items()
```

Iterating Through a Dictionary

Iterate Over Keys

```
for key in student:
    print(key)
```

Iterate Over Values

```
for value in student.values():
    print(value)
```

Iterate Over Key–Value Pairs

```
for key, value in student.items():
    print(key, ":", value)
```

Checking Key Existence

```
print("name" in student) # True
print("marks" not in student) # True
```

Dictionary with Mixed Data Types

```
data = {
    1: "One",
    "two": 2,
    (3, 4): "Tuple key"
}
```

Nested Dictionaries

```
student = {
    "name": "Rahul",
    "marks": {
        "maths": 90,
        "science": 85
    }
}
print(student["marks"]["maths"]) # 90
```

Dictionary Use Cases

- Storing student records
- Configuration data
- JSON-like structures
- Fast lookup tables

Dictionary vs List (Quick Comparison)

Feature	List	Dictionary
Access method	Index	Key
Duplicate keys	Allowed	Not allowed
Data structure	Ordered values	Key–Value pairs
Lookup speed	Slower	Faster

Summary – Dictionary

- Stores data as **key–value pairs**
- Keys are **unique and immutable**
- Values are mutable
- Very fast for searching and updating data

Module 7 – Modules and Packages

Overview

As Python programs grow in size, writing all code in a single file becomes difficult to manage and maintain. This module introduces modules and packages, which are Python's way of organizing code into reusable and structured units. Students will learn how to create and use built-in modules, user-defined modules, and third-party packages, understand different import techniques, and organize real-world projects using packages.

7.1. What Is a Module?

A module is a single Python file with a .py extension that contains variables, functions, classes, or executable code. Modules allow developers to logically separate functionality and reuse code across multiple programs.

```
# utils.py
def add(a, b):
    return a + b
# main.py
import utils
print(utils.add(2, 3))
```

7.2. Import Statements

Python provides multiple ways to import modules depending on how much functionality is required. You can import the entire module, import specific names, or use aliases to simplify access and avoid naming conflicts.

```
import math as m
from datetime import datetime as dt
```

7.3. The Dunder (__) Convention

Names that begin and end with double underscores are called dunder names. They have special meanings in Python. The most commonly used is `__name__`, which helps distinguish whether a file is being executed directly or imported as a module.

```
if __name__ == "__main__":
    print("Running as a script")
```

7.4. Reloading a Module in Python

• What is Module Reloading?

During development, a Python module's code may change frequently. Normally, once a module is imported, Python does not automatically reload it, even if the source file is modified.

To apply changes without restarting the Python interpreter, Python provides the `importlib.reload()` function.

• Why Module Reloading is Needed?

1. Faster development and testing
2. Useful in interactive mode, IDLE, Jupyter Notebook, and REPL
3. Avoid restarting the interpreter after every code change

importlib Module

The `importlib` module provides tools related to the import system. The `reload()` function is part of this module.

Example

Step 1: Create a module `mymodule.py`

```
def greet():
    print("Hello from version 1")
```

Step 2: Import and use it

```
import mymodule
mymodule.greet()
```

Output:

```
Hello from version 1
```

Step 3: Modify mymodule.py

```
def greet():
    print("Hello from version 2")
```

Step 4: Reload the module

```
import importlib
importlib.reload(mymodule)
mymodule.greet()
```

Output:

```
Hello from version 2
```

7.5. Built-in Modules

Python includes a rich standard library of built-in modules that provide ready-made functionality such as mathematical operations, file handling, operating system interaction, and system-level access.

```
import sys
print(sys.version)
```

7.6. User-Defined Modules

User-defined modules are custom Python files created by developers to organize project-specific logic. They improve readability and allow teams to collaborate by separating responsibilities.

7.7. Third-Party Modules and pip

Third-party modules are developed by the Python community and distributed through the Python Package Index (PyPI). They can be installed using the pip tool and imported like any other module.

```
pip install requests
import requests
response =
requests.get('https://api.example.com')
```

7.8. sys.path

Python searches for modules in a list of directories defined in `sys.path`. Custom directories can be added at runtime if modules are stored outside standard locations.

```
import sys
sys.path.append(r'C:/my/custom/modules')
```

7.9. Packages and Sub-Packages

A package is a directory that groups related modules together. It must contain an `__init__.py` file to be recognized as a package. Packages help structure large applications into logical components.

```
myproject/
| __init__.py
| core.py
| utils.py
└─ subpkg/
| __init__.py
| helpers.py
```

7.10. Namespaces

Each module in Python has its own namespace, which maps names to objects. Namespaces prevent naming conflicts by keeping identifiers isolated within their modules.

7.11. Modular Programming and Code Reuse

Modular programming promotes separation of concerns, code reuse, and easier maintenance. Well-structured modules and packages make applications scalable and easier to test.

Sample Program – Using Modules and Packages

```
# mypackage/math_ops.py
def square(x):
    return x * x
def cube(x):
    return x ** 3
# main.py

from mypackage.math_ops import square, cube
print(square(4))
print(cube(2))
```

Assignments

1. Multiple Choice: Which statement correctly imports only the sqrt function from the math module? A) import math.sqrt B) from math import sqrt C) import sqrt from math D) import math as sqrt
2. Short Answer: Explain why the `__name__ == '__main__'` condition is important in Python programs.
3. Practical Task: Create a package named `utils` with two modules: `calc.py` (add, subtract) and `text.py` (reverse string). Write a script that imports and uses all functions.
4. Debugging Task: Fix the error in the following code:
`import mymodule`

`mymodule.reload()`

5. Conceptual Question: Mention two advantages of using third-party libraries instead of

writing code from scratch.

Conclusion

Modules and packages are fundamental to writing clean, maintainable, and scalable Python programs. By organizing code effectively and using Python's rich ecosystem of built-in and third-party modules, developers can build robust applications with minimal duplication and maximum clarity.

Module 8 – File Handling and Serialization

Overview

File handling allows Python programs to **store data permanently, read external files, and interact with the operating system**. Serialization enables Python objects to be converted into a format that can be saved to disk and reconstructed later.

This module covers:

- Text and binary file operations
- File modes and file pointers
- Safe file handling using context managers
- OS-level file and directory operations
- Serialization using **pickle** and **JSON**

Learning Objectives

- Understand the importance and real-world use cases of file handling
- Identify and use different file modes (r, w, a, b, x, +)
- Read, write, and append text and binary files
- Use `open()` and `close()` correctly
- Access file object properties (name, mode, closed)
- Navigate within files using `seek()` and `tell()`
- Use the `with` statement for safe file handling
- Handle file-related exceptions

- Perform file and directory operations using `os` and `pathlib`
- Understand serialization and its importance
- Serialize and deserialize data using `pickle` and `json`
- Follow best practices for clean and error-free file handling

8.1 What is File Handling?

File handling refers to **reading data from files** and **writing data to files** stored on disk.

Why File Handling is Important

- Store application data permanently
- Read configuration and log files
- Process large datasets
- Exchange data between programs
- Maintain user records

Real-World Examples

- Student records system
- Log files in servers
- Configuration files (.json, .ini)
- CSV files for reports

8.2 Opening and Closing Files

Python uses the `open()` function to work with files.

```
file = open("example.txt", "r")
```

```
file.close()
```

Syntax

```
open(filename, mode)
```

- filename → File name or path

- mode → How the file is accessed
- Always close files to free system resources.

8.3 File Modes (VERY IMPORTANT)

File modes define **how a file is opened**.

Text File Modes

'r' – Read Mode

- Opens file for reading
- File must exist

```
f = open("data.txt", "r")
print(f.read())
```

'w' – Write Mode

- Creates new file or overwrites existing file

```
f = open("data.txt", "w")
f.write("Hello World")
```

Existing data will be deleted.

'a' – Append Mode

- Adds data to the end of file

```
f = open("data.txt", "a")
f.write("\nNew Line")
```

'x' – Exclusive Creation

- Creates a new file
- Error if file already exists

```
f = open("newfile.txt", "x")
```

Binary File Modes

'b' – Binary Mode

Used with other modes (rb, wb, ab)

```
f = open("image.jpg", "rb")
```

Read & Write Combined Modes

'r+'

- Read and write
- File must exist

'w+'

- Write and read
- Overwrites file

'a+'

- Append and read

Text Mode ('t')

(Default mode)

```
open("file.txt", "rt")
```

File Mode Summary Table

Mode	Description
r	Read
w	Write (overwrite)
a	Append
x	Create new file
b	Binary
t	Text
+	Read + Write

8.4 Reading Files

read()

Reads entire file content.

```
f = open("data.txt", "r")
print(f.read())
```

readline()

Reads one line at a time.

```
print(f.readline())
```

readlines()

Reads all lines into a list.

```
lines = f.readlines()
```

8.5 Writing Files

write()

Writes a string.

```
f.write("Hello File")
```

writelines()

Writes multiple strings.

```
f.writelines(["Line1\n", "Line2\n"])
```

8.6 Context Manager (with Statement)

Automatically closes files.

```
with open("data.txt", "w") as f:
```

```
f.write("Safe Writing")
```

✓ Recommended

✓ Prevents resource leaks

8.7 File Object Properties

```
f = open("data.txt", "r")
```

```
print(f.name) # File name
```

```
print(f.mode) # File mode
```

```
print(f.closed) # File status
```

8.8 File Pointer: seek() and tell()

tell()

Returns current pointer position.

```
print(f.tell())
```

seek()

Moves pointer to a specific position.

```
f.seek(0)
```

8.9 Exception Handling in File I/O

try:

```
f = open("file.txt", "r")
```

```
print(f.read())
```

except FileNotFoundError:

```
print("File not found")
```

finally:

```
f.close()
```

8.10 OS Module – File & Directory Operations

```
import os
```

```
os.getcwd()
```

```
os.listdir()
```

```
os.mkdir("test")
```

```
os.remove("data.txt")
```

8.11 Binary Files

Used for images, audio, videos.

```
with open("image.jpg", "rb") as f:
```

```
data = f.read()
```

8.12 Serialization

Serialization converts Python objects into a format that can be saved and restored later.

Pickle Serialization

```
import pickle

data = {"name": "Alice", "score": 90}

with open("data.pkl", "wb") as f:
    pickle.dump(data, f)

with open("data.pkl", "rb") as f:
    data = pickle.load(f)
```

JSON Serialization

```
import json

with open("data.json", "w") as f:
    json.dump(data, f)

with open("data.json", "r") as f:
    data = json.load(f)
```

Pickle vs JSON

Feature	Pickle	JSON
Readable	No	Yes
Secure		
Python only	Yes	No
Complex objects	Yes	No

Best Practices

- Always use with
- Handle exceptions
- Choose correct file mode
- Avoid pickle for untrusted data
- Close files properly

Module 9 – Object-Oriented Programming (OOP)

9.1 Overview

Object-Oriented Programming (OOP) is a programming paradigm that organizes software design around data, or objects, rather than functions and logic. In Python, OOP allows developers to model real-world entities using classes and objects, making programs easier to understand, maintain, and extend. This module provides an in-depth understanding of OOP concepts that are essential for real-world software development.

9.2 Classes and Objects

9.2.1 Class Definition and Object Creation

A class is a blueprint that defines the structure and behavior of objects. An object is an instance of a class. Classes group data (attributes) and behavior (methods) together.

```
class Dog:
    def __init__(self, name):
        self.name = name
    def bark(self):
        print(f"{self.name} says woof!")
d = Dog("Rex")
d.bark()
```

9.2.2 The self Keyword

The self parameter represents the current object. It allows instance methods to access and modify the object's attributes. Every instance method must include self as its first parameter.

9.3 Attributes

9.3.1 Instance Attributes

Instance attributes are variables that belong to a specific object. Each instance maintains its own copy of these attributes.

9.3.2 Class Attributes

Class attributes are shared among all instances of a class. They are useful for maintaining common state.

```
class Counter:
    total = 0
    def __init__(self):
        self.value = 0
        Counter.total += 1
```

9.4 Access Modifiers and Name Mangling

Python does **not** strictly enforce access modifiers like public, private, or protected. Instead, Python uses **naming conventions** to indicate how a variable or method **should be used**, and relies on **developer discipline** rather than compiler enforcement.

These conventions help achieve **encapsulation**, which means **protecting internal data** and exposing only what is necessary.

```
class Example:
    public = 1
    _protected = 2
    __private = 3
```

9.4.1 Public Members

Public members are accessible from anywhere — inside or outside the class.

- **Naming rule:** No leading underscore
- **Access level:** Fully accessible
- **Usage:** Safe to use directly

```
class Student:
    name = "Alice" # public attribute
s = Student()
print(s.name) # Allowed
```

Public members are meant to be used freely by other parts of the program.

9.4.2 Protected Members (Single Underscore _)

Protected members are **intended for internal use** within the class and its subclasses.

- **Naming rule:** Single leading underscore (`_var`)
- **Access level:** Accessible, but **discouraged**
- **Usage:** For internal logic and inheritance

```
class Account:
    _balance = 1000 # protected attribute
class SavingsAccount(Account):
    def show_balance(self):
        print(self._balance)
acc = SavingsAccount()
acc.show_balance() # Allowed
print(acc._balance) # Allowed, but discouraged
```

9.4.3 Private Members (Double Underscore __)

Private members are **meant to be accessed only within the class**.

- **Naming rule:** Double leading underscore (`__var`)
- **Access level:** Restricted via **name mangling**

- **Purpose:** Prevent accidental access and override in subclasses

```
class Example:
    public = 1
    _protected = 2
    __private = 3
    def show(self):
        print(self.__private) # Allowed inside class
e = Example()
print(e.public) # 1
print(e._protected) # 2 (discouraged but works)
# print(e.__private) # AttributeError
```

9.4.4 Name Mangling (How Python Handles __private)

Python internally **renames** private variables using a process called **name mangling**.

`__private` → `_ClassName__private`

So internally, Python converts:

`__private` → `_Example__private`

This is why the following works:

`print(e._Example__private) # 3`

Name mangling:

- Prevents accidental access
- Avoids name clashes in subclasses
- Still allows access if absolutely required (advanced use)

9.4.5 Why Name Mangling Exists (Important for Exams)

Name mangling is **not about security**, but about:

1. **Avoiding accidental misuse**
2. **Preventing method overriding conflicts**
3. **Improving code maintainability**

Example with inheritance:

```
class Parent:
    def __init__(self):
        self.__data = "Parent Data"

class Child(Parent):
    def __init__(self):
        super().__init__()
        self.__data = "Child Data"
c = Child()
print(c._Parent__data) # Parent Data
print(c._Child__data) # Child Data
```

9.4.6 Summary Table (Student-Friendly)

Type	Naming	Enforced?	Accessible Outside Class	Intended Use
Public	var	No	Yes	General use
Protected	<code>_var</code>	No	⚠ Yes (discouraged)	Internal / subclass
Private	<code>__var</code>	⚠ Partially (name mangling)	Directly no	Strict internal use

9.5 Inheritance

9.5.1 Concept of Inheritance

Inheritance is an Object-Oriented Programming (OOP) concept where a **child class** acquires the **properties (variables) and behaviors (methods)** of a **parent class**. Inheritance helps to:

- Reduce code duplication
- Reuse existing logic
- Create a logical relationship between classes

- Improve maintainability and scalability

Example (real world):

Animal → *Dog* → *Puppy*

Basic Syntax

```
class Parent:
    pass
class Child(Parent):
    pass
```

Simple Example

```
class Animal:
    def speak(self):
        print("Animal makes a sound")
class Dog(Animal):
    def bark(self):
        print("Dog barks")
```

- Dog automatically gets access to `speak()`
- This is called **inheritance**

9.5.2 Types of Inheritance in Python

Python supports **all major inheritance types**.

1. Single Inheritance

One child class inherits from one parent class.

```
class Parent:
    pass
class Child(Parent):
    pass
```

- ✓ Most commonly used
- ✓ Easy to understand

2. Multilevel Inheritance

A class inherits from another child class.

```
class Animal:
    pass
class Dog(Animal):
    pass
class Puppy(Dog):
    pass
```

Hierarchy:

Animal → Dog → Puppy

3. Hierarchical Inheritance

Multiple child classes inherit from the same parent class.

```
class Animal:
    pass
class Dog(Animal):
    pass
class Cat(Animal):
    pass
```

- ✓ One parent
- ✓ Many children

4. Multiple Inheritance

A child class inherits from **more than one parent class**.

```
class Father:
    pass
class Mother:
    pass
class Child(Father, Mother):
    pass
```

Python allows multiple inheritance

⚠ Can cause confusion if not handled properly

5. Hybrid Inheritance

A combination of two or more inheritance types.

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

✓ Powerful

⚠ Requires understanding of **MRO**

9.5.3 super() Function

The super() function is used in a child class to call methods of the parent class.

Why use super()?

- Calls parent methods safely
- Supports multiple inheritance
- Avoids hardcoding parent class names

Example

```
class Animal:
    def speak(self):
        print("Animal sound")
class Dog(Animal):
    def speak(self):
        super().speak()
        print("Woof")
```

Output:

Animal sound Woof

9.5.4 Method Resolution Order (MRO)

MRO defines the **order in which Python searches for methods** when multiple inheritance is used. Python follows the **C3 Linearization Algorithm**.

Why MRO is important?

- Prevents ambiguity
- Decides which method gets executed
- Ensures consistent inheritance behavior

Example

```
class A:
    def show(self):
        print("A")
class B(A):
    def show(self):
        print("B")
class C(A):
    def show(self):
        print("C")
class D(B, C):
    pass
d = D()
d.show()
```

Output:

B

Checking MRO

```
print(D.mro())
```

Output:

[D, B, C, A, object]

Python searches methods **from left to right**, following this order.

9.6 Polymorphism and Duck Typing

Polymorphism

Polymorphism means “many forms”. In Python, the **same method name** can behave **differently for different objects**.

The behavior depends on the object, not the method name.

Example

```
class Dog:
    def speak(self):
        print("Woof")
class Cat:
    def speak(self):
        print("Meow")
animals = [Dog(), Cat()]
for a in animals:
    a.speak()
```

Output:

Woof Meow Same method speak(), different behavior.

Duck Typing

Python follows **duck typing**:

“If it looks like a duck and quacks like a duck, it is a duck.” Python **does not check object type**, it checks **method availability**.

```
class File:
    def read(self):
        print("Reading file")
class Socket:
    def read(self):
        print("Reading socket")
def fetch_data(source):
    source.read()
fetch_data(File())
fetch_data(Socket())
```

✓ No inheritance

✓ Only method behavior matters

9.7 Operator Overloading

Operator overloading allows us to define **custom behavior** for operators like +, -, *, ==. Python uses **special methods (dunder methods)**.

Common Operator Methods

- `__add__` → +
- `__sub__` → -
- `__str__` → `print()`

Example

```
class Point:
    def __init__(self, x):
        self.x = x
    def __add__(self, other):
        return self.x + other.x
p1 = Point(10)
p2 = Point(20)
print(p1 + p2)
```

Output:

30

9.8 Encapsulation with Properties

Encapsulation means:

- Hiding internal data
- Controlling access using methods

Python uses **properties** instead of traditional getters/setters.

Example

```
class Student:
    def __init__(self):
        self._marks = 0
    @property
    def marks(self):
        return self._marks
    @marks.setter
    def marks(self, value):
        if value < 0:
            print("Invalid marks")
        else:
            self._marks = value
s = Student()
s.marks = 80
print(s.marks)
```

- ✓ Data is protected
- ✓ Validation is enforced

9.9 Abstraction using abc Module

Abstraction hides implementation details and shows **only what is necessary**. Python provides abstraction using the **abc module**.

Abstract Class Rules

- Cannot be instantiated
- Forces child classes to implement methods

Example

```
from abc import ABC, abstractmethod
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass
class Square(Shape):
    def area(self):
        return 4 * 4
```

- ✓ Ensures method implementation
- ✓ Defines a contract for subclasses

9.10 Method Types

Python supports **three types of methods**.

1. Instance Method

- Works with object data
- Uses self

```
class Demo:
    def show(self):
        print("Instance method")
```

2. Class Method

- Works with class data
- Uses @classmethod and cls

```
class Demo:
    @classmethod
    def info(cls):
        print("Class method")
```

3. Static Method

- Independent of class and object
- Uses @staticmethod

```
class Demo:
    @staticmethod
    def utility():
        print("Static method")
```

9.11 Composition vs Inheritance

Inheritance → “is-a” relationship

```
class Car(Vehicle):
    pass
```

Composition → “has-a” relationship

```
class Engine:
    pass
class Car:
    def __init__(self):
        self.engine = Engine()
```

Why Composition is Preferred?

- Loosely coupled
 - Flexible design
 - Easier maintenance
- ✓ Modern applications prefer **composition over inheritance**

9.12 Object Lifecycle and Scope

Object Lifecycle

1. Object is created
2. Reference count increases
3. Object is used
4. Reference count becomes zero
5. Object is destroyed

Python uses:

- Reference counting
- Garbage collection

Example

```
class Test:
    pass
t = Test()
del t
```

Once no references exist, Python frees memory automatically.

9.13 Sample Program – OOP in Action

```
class BankAccount:
    def __init__(self, balance):
        self._balance = balance
    def deposit(self, amount):
        self._balance += amount
    def withdraw(self, amount):
        if amount > self._balance:
            raise ValueError("Insufficient balance")
        self._balance -= amount
```

9.14 Assignments

A. Conceptual Questions

1. Explain the four pillars of OOP with examples.
2. Differentiate between class attributes and instance attributes.

B. Coding Exercises

3. Create a Student class with attributes and methods for GPA calculation.
4. Implement inheritance for Employee and Manager classes.

C. Advanced Tasks

5. Implement operator overloading for a custom class.
6. Demonstrate abstraction using an abstract base class.

9.15 Conclusion

Object-Oriented Programming is a cornerstone of modern software development. By mastering OOP concepts in Python, developers gain the ability to write scalable, reusable, and maintainable applications suitable for real-world systems.

Module 10 – Exception Handling

10.1 Overview

Exception handling is the mechanism used by Python to detect, report, and manage errors that occur while a program is running. Instead of abruptly terminating a program when an error occurs, exception handling allows developers to respond gracefully, show meaningful messages to users, and safely release resources such as files or network connections. In this module, you will learn the difference between syntax errors and runtime exceptions, understand Python's exception hierarchy, and use try, except, else, and finally blocks to write robust and maintainable programs. You will also learn how to create and use custom exception classes and follow best practices for clean error handling.

10.2 Errors and Exceptions

10.2.1 Syntax Errors

A syntax error is detected by the Python parser before the program starts executing. These errors occur when Python's grammar rules are violated, such as missing colons, unmatched parentheses, or incorrect indentation. When a syntax error is found, the interpreter stops execution immediately and displays an error message along with the line number.

10.2.2 Runtime Errors (Exceptions)

Runtime errors, also called exceptions, occur while the program is executing. These errors arise due to invalid operations, such as dividing by zero or accessing a non-existent list index. When an exception occurs, Python creates an exception object and raises it. If the exception is not handled, the program terminates and prints a traceback.

10.3 The try-except-else-finally Structure

10.3.1 Basic Structure

Python uses structured blocks to handle exceptions in a controlled manner.

```
try:
    result = 10 / divisor
except ZeroDivisionError as e:
    print("Cannot divide by zero!", e)
except Exception as e:
    print("Unexpected error:", e)
else:
    print("Result is", result)
finally:
    print("Cleanup complete")
```

10.3.2 Purpose of Each Block

The try block contains code that may raise an exception. The except block handles specific exceptions and prevents program termination. The optional else block executes only when no exception occurs in the try block. The finally block always executes, regardless of whether an exception occurred, and is commonly used for cleanup tasks.

10.4 Raising Exceptions

10.4.1 Using the raise Keyword

Python allows programmers to raise exceptions explicitly using the raise keyword. This is useful when validating input or enforcing business rules.

```
if age < 0:
```

```
    raise ValueError("Age cannot be negative")
```

10.4.2 Re-raising Exceptions

Inside an except block, using raise without specifying an exception will re-raise the same exception. This is helpful when logging an error but still allowing the caller to handle it.

10.5 Built-in Exception Types

10.5.1 Commonly Used Exceptions

Python provides many built-in exception classes to represent common error conditions.

Exception	Typical Cause
ZeroDivisionError	Division by zero
ValueError	Invalid value for an operation
TypeError	Invalid type used in an operation
IndexError	Index out of range
NameError	Variable not defined
KeyError	Missing dictionary key
FileNotFoundError	File does not exist

IOError / OSError

Input/output related errors

10.6 Exception Hierarchy

10.6.1 Understanding the Hierarchy

All exceptions in Python inherit from BaseException. Most user-defined and commonly handled exceptions inherit from the Exception class. Understanding this hierarchy allows developers to catch groups of related exceptions efficiently.

BaseException $\hookleftarrow$ Exception

| $\hookleftarrow$ ArithmeticError

| | $\hookleftarrow$ ZeroDivisionError

| | $\hookleftarrow$ OverflowError

| $\hookleftarrow$ LookupError

| | $\hookleftarrow$ IndexError

| | $\hookleftarrow$ KeyError

| $\hookleftarrow$...

10.7 Handling Multiple and Nested Exceptions

10.7.1 Multiple Exceptions in a Single except

Multiple exceptions can be handled together by specifying them as a tuple.

```
try:
    val = int(user_input)
except (ValueError, TypeError) as e:
    print("Invalid input:", e)
```

10.7.2 Nested try–except Blocks

Nested exception handling is useful when different operations require separate handling logic.

```
try:
    f = open("data.txt", "r")
    try:
        data = f.read()
        number = int(data)
    except ValueError:
        print("File does not contain a valid integer")
    finally:
        f.close()
except FileNotFoundError:
    print("File not found")
```

10.8 Custom Exceptions

10.8.1 Creating Custom Exception Classes

Custom exceptions are created by subclassing `Exception` or one of its subclasses. They make programs easier to read and maintain by clearly expressing specific error conditions.

```
class InsufficientFundsError(Exception):
    """Raised when a withdrawal exceeds the account balance."""
    pass
class BankAccount:
    def __init__(self, balance=0):
        self.balance = balance
    def withdraw(self, amount):
        if amount > self.balance:
            raise InsufficientFundsError(
                f"Attempted to withdraw {amount}, but only {self.balance} available"
            )
        self.balance -= amount
```

10.9 Best Practices for Exception Handling

10.9.1 Recommended Guidelines

Always catch specific exceptions instead of using a bare `except`. Avoid silently ignoring exceptions; log or display meaningful messages. Use `finally` or context managers to ensure resources are released properly. Prefer `with` statements for file handling. Document raised exceptions in function docstrings and design a clear custom exception hierarchy for larger projects.

10.10 Sample Program – Exception Handling Demonstration

10.10.1 Complete Example

```
# exception_demo.py
def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        print("Error: division by zero")
        raise
def get_int(prompt):
    while True:
        try:
```

```

        return int(input(prompt))
    except ValueError:
        print("Please enter a valid
integer.")
def main():
    try:
        x = get_int("Enter numerator: ")
        y = get_int("Enter denominator: ")
        result = divide(x, y)
    except ZeroDivisionError:
        print("Cannot compute result due to
zero denominator.")
    else:
        print(f"Result: {result}")
    finally:
        print("Program finished - any cleanup
would go here.")
if __name__ == "__main__":
    main()

```

10.11 Assignments

10.11.1 Objective Questions

1. Which block executes only when the try block completes without raising an exception? A) except B) else C) finally D) raise

10.11.2 Short Answer Questions

2. Explain why catching the base class Exception in a broad except clause can be problematic.

10.11.3 Programming Exercises

3. Write a function read_json_file(path) that opens a JSON file safely using a with statement, catches FileNotFoundError and returns None, catches json.JSONDecodeError and prints an error message, and returns the parsed JSON object on success.

10.11.4 Debugging Exercise

4. The following code is intended to handle both ValueError and TypeError, but it does not work as expected. Identify and correct the issue.

try:

number = int("abc")

except ValueError, TypeError as e:

print(e)

10.11.5 Conceptual Question

5. In what situations would inheriting from BaseException instead of Exception be appropriate? Explain briefly.

10.12 Conclusion

Exception handling is essential for building reliable and user-friendly Python applications. While syntax errors are detected before execution, runtime exceptions occur during program execution and must be handled carefully to prevent crashes. By using structured exception handling, understanding the exception hierarchy, and following best practices, developers can write programs that validate input, manage resources safely, and provide clear feedback when errors occur.

Module 11 – Functional Programming

11.1 Overview

Functional Programming (FP) is a programming paradigm that emphasizes the use of functions as the primary building blocks of a program. In Python, functional programming is supported alongside procedural and object-oriented styles, allowing developers to choose the most suitable approach for a given problem. By treating functions as first-class objects and encouraging immutability, FP leads to code that is easier to test, debug, reuse, and reason about. In this module, you will explore core functional programming concepts, higher-order functions, lambda expressions, comprehensions, iterators, generators, decorators, and advanced utilities from the `functools` module.

11.2 Functional Programming Concepts

11.2.1 Pure Functions

A pure function is a function that always returns the same output for the same input and does not produce any side effects. Side effects include modifying global variables, changing input arguments, or performing input/output operations. Pure functions are predictable and easy to test, making them a core concept in functional programming.

11.2.2 Immutability

Immutability refers to data that cannot be changed after it is created. In Python, data types such as strings and tuples are immutable by default. Functional programming encourages treating other data structures as immutable as well, meaning that instead of modifying existing data, new data is created. This reduces bugs caused by unintended data changes.

11.2.3 First-Class Functions

In Python, functions are first-class citizens. This means they can be assigned to variables, passed as arguments to other functions, and returned from functions. This capability enables powerful programming patterns such as callbacks, decorators, and higher-order functions.

11.3 Higher-Order Functions

11.3.1 Definition

A higher-order function is a function that either takes one or more functions as arguments or returns a function as its result. Python provides several built-in higher-order functions that are commonly used in functional programming.

11.3.2 Common Higher-Order Functions

The `map()` function applies a given function to each item of an iterable and returns a new iterator with the transformed values. The `filter()` function selects elements from an iterable for which a given function returns `True`. The `functools.reduce()` function cumulatively applies a function to the items of an iterable to reduce it to a single value. The `sorted()` function can also act as a higher-order function when a key function is provided.

11.4 Lambda Functions

11.4.1 Anonymous Functions

Lambda functions are small, anonymous functions defined using the `lambda` keyword. They can take any number of arguments but contain only a single expression. Lambda functions are commonly used for short, simple operations where defining a full function would be unnecessary.

```
add = lambda x, y: x + y
print(add(2, 3))
```

11.4.2 Use Cases and Limitations

Lambda functions improve readability when used sparingly, such as within `map()` or `filter()`. However, they should not replace regular functions for complex logic, as they can reduce code clarity.

11.5 The `functools` Module

11.5.1 `lru_cache()`

The `lru_cache()` decorator provides memoization by caching the results of function calls. This is particularly useful for expensive or recursive computations, as it avoids repeated calculations for the same inputs.

11.5.2 `partial()`

The `partial()` function allows you to fix a certain number of arguments of a function and generate a new function with fewer parameters. This is useful when you want to reuse a general function with specific default values.

```
from functools import partial
def power(base, exp):
    return base ** exp
square = partial(power, exp=2)
print(square(4))
```

11.6 Comprehensions

11.6.1 Types of Comprehensions

Comprehensions provide a concise and readable way to create collections. List comprehensions generate lists, set comprehensions generate sets, dictionary comprehensions generate dictionaries, and generator expressions generate generators. Nested comprehensions allow processing of multi-dimensional data structures.

11.6.2 Benefits of Comprehensions

Comprehensions are often more readable and efficient than traditional loops, especially for data transformation tasks. They align well with functional programming principles.

11.7 Iterables and Iterators

11.7.1 Iterables

An iterable is any object capable of returning its elements one at a time. Examples include lists, tuples, strings, sets, and dictionaries. An object is iterable if it implements the `__iter__()` method or the `__getitem__()` method.

11.7.2 Iterators

An iterator is an object that represents a stream of data. It keeps track of the current state and produces the next value using the `__next__()` method. When no more data is available, it raises a `StopIteration` exception.

```
nums = [1, 2]

it = iter(nums)
print(next(it))
```

11.8 Generators

11.8.1 Generator Functions

Generators are a simple way to create iterators using functions and the `yield` keyword. Unlike regular functions, generator functions return values one at a time and pause execution between yields.

11.8.2 Lazy Evaluation

Generators use lazy evaluation, meaning values are generated only when requested. This makes them memory-efficient and ideal for working with large datasets or infinite sequences.

```
def count_up_to(n):
    count = 1
    while count <= n:
        yield count
        count += 1
```

11.9 Decorators

11.9.1 Concept of Decorators

Decorators are higher-order functions that modify or extend the behavior of other functions without changing their source code. They are commonly used for logging, access control, timing, and caching.

11.9.2 Decorator Syntax

The `@decorator_name` syntax provides a clean and readable way to apply decorators to functions.

```
def my_decorator(func):
    def wrapper():
        print("Before")
        func()
        print("After")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")
```

11.10 Sample Program – Functional Features Demonstration

11.10.1 Complete Example

```
import functools
import time

# Decorator to measure execution time
def timer(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"Execution time: {end - start} seconds")
        return result
    return wrapper
```

```

        result = func(*args, **kwargs)
        print(f"{func.__name__} took
{time.time() - start:.4f}s")
        return result
    return wrapper
# Generator for Fibonacci numbers
def fib_gen(n):
    a, b = 0, 1
    for _ in range(n):
        yield a
        a, b = b, a + b
# Pure function for reduce
def multiply(x, y):
    return x * y
# Recursive Fibonacci with caching
@functools.lru_cache(maxsize=None)
def fib_recursive(n):
    if n < 2:
        return n
    return fib_recursive(n-1) +
fib_recursive(n-2)
@timer
def process_data():
    data = range(20)
    evens = list(filter(lambda x: x % 2 == 0,
data))
    squared = [x**2 for x in evens]
    product = functools.reduce(multiply,
squared if squared else [0])
    print(f"Product of squared evens:
{product}")
    print("First 5 Fibonacci numbers:",
list(fib_gen(5)))
process_data()

```

11.11 Assignments

11.11.1 Objective Question

Which of the following is not a characteristic of a pure function? A) It depends on global variables B) It returns the same output for the same input C) It has no side effects D) It does not modify its arguments

11.11.2 Short Answer Question

Explain the difference between an iterable and an iterator with examples.

11.11.3 Programming Exercise

Write a generator function `squares(n)` that yields the square of numbers from 1 to `n`. Use a for loop to display the values.

11.11.4 Debugging Exercise

The following code uses a generator expression twice but produces an unexpected result. Explain the reason and correct the code.

```
nums = (x for x in range(10))
```

```
print(sum(nums))
```

```
print(sum(nums))
```

11.11.5 Conceptual Question

Explain how `functools.lru_cache` improves the performance of recursive functions such as Fibonacci.

11.12 Conclusion

Functional programming enhances Python's flexibility by encouraging clean, modular, and expressive code. Through concepts such as pure functions, immutability, higher-order functions, and lazy evaluation, developers can write efficient and maintainable programs. While Python is not a purely functional language, combining functional programming techniques with procedural and object-oriented approaches allows developers to fully leverage Python's versatility for real-world applications.

Module 12 – Python and MySQL Database Connectivity

12.1 Overview

Database connectivity is a critical skill for building real-world applications, as most software systems need to store, retrieve, and manage data efficiently. In this module, you will learn how to integrate Python with **MySQL**, one of the most widely used relational database management systems. The module covers establishing database connections, executing SQL queries securely, managing transactions, handling exceptions, and implementing complete **CRUD** (Create, Read, Update, Delete) operations. By the end of this module, you will be able to build Python programs that interact reliably and safely with a MySQL database.

12.2 The `mysql.connector` Module

12.2.1 Purpose of the Connector

To communicate with a MySQL database from Python, a connector library is required. The `mysql.connector` module is an official MySQL driver that allows Python programs to connect to MySQL servers and execute SQL commands.

12.2.2 Installation and Import

The connector can be installed using `pip`.

```
pip install mysql-connector-python
```

```
import mysql.connector
```

12.3 Connecting to a MySQL Database

12.3.1 Establishing a Connection

A connection represents an active session between the Python program and the MySQL server. The `connect()` method is used to establish this connection by providing server and database credentials.

```
import mysql.connector
conn = mysql.connector.connect(
    host="localhost",
    user="root",
    password="password",
    database="school_db"
)
```

12.3.2 Connection Best Practices

Always keep database credentials secure and avoid hardcoding sensitive information in production applications. Close the connection once database operations are complete to free resources.

12.4 Cursor Object

12.4.1 Role of the Cursor

The cursor object is used to execute SQL statements and fetch results from the database. It acts as a pointer that navigates through query results.

```
cursor = conn.cursor()
```

12.4.2 Types of Cursor Operations

Cursors support operations such as executing queries, fetching one record, fetching all records, and retrieving metadata such as the number of affected rows.

12.5 Executing SQL Queries

12.5.1 Creating Tables

SQL commands can be executed directly from Python using the cursor's `execute()` method.

```
cursor.execute(
    "CREATE TABLE IF NOT EXISTS students ("
    "id INT AUTO_INCREMENT PRIMARY KEY,"
    "name VARCHAR(255),"
    "age INT"
    ")")
```

12.5.2 DDL vs DML Queries

Data Definition Language (DDL) queries such as `CREATE` and `ALTER` define the database structure, while Data Manipulation Language (DML) queries such as `INSERT`, `UPDATE`, and `DELETE` modify data stored in tables.

12.6 Parameterized Queries and Security

12.6.1 SQL Injection Risk

Building SQL queries by concatenating strings can lead to SQL injection vulnerabilities, where malicious input alters the intended SQL command.

12.6.2 Using Placeholders

Parameterized queries use placeholders (`%s`) to safely pass values to SQL statements. This ensures that user input is treated as data, not executable SQL code.

```
sql = "INSERT INTO students (name, age) VALUES (%s, %s)"
values = ("John", 20)
cursor.execute(sql, values)
```

12.7 Transactions Management

12.7.1 Committing Transactions

For operations that modify data, changes are not saved permanently until `commit()` is called. This allows multiple related operations to be treated as a single transaction.

12.7.2 Rolling Back Transactions

If an error occurs during a transaction, `rollback()` can be used to undo all changes made since the last commit, preserving data consistency.

```
try:
    cursor.execute(sql, values)
    conn.commit()
except mysql.connector.Error:
    conn.rollback()
```

12.8 Fetching Query Results

12.8.1 `fetchone()` and `fetchall()`

The `fetchone()` method retrieves a single row from the result set, while `fetchall()` retrieves all rows as a list of tuples.

```
cursor.execute("SELECT * FROM students")
rows = cursor.fetchall()
for row in rows:
    print(row)
```

12.8.2 Iterating Over Results

Fetches results can be processed using loops, allowing data to be displayed, transformed, or used in further logic.

12.9 Handling JSON Data in MySQL

12.9.1 JSON Data Type

MySQL supports a native JSON data type, which allows structured data to be stored within a column. This is useful for semi-structured or flexible data models.

12.9.2 Storing JSON from Python

Python dictionaries can be converted into JSON strings using the `json` module before being inserted into the database.

```
import json
metadata = {"hobbies": ["reading", "coding"]}
json_str = json.dumps(metadata)
cursor.execute(
    "INSERT INTO users (metadata) VALUES (%s)",
    (json_str,)
)
```

12.10 Exception Handling in Database Operations

12.10.1 Importance of Error Handling

Database operations can fail due to connection issues, invalid queries, or constraint violations. Using exception handling ensures that errors are managed gracefully.

12.10.2 Safe Cleanup of Resources

Connections and cursors should always be closed, preferably using a `finally` block.

```
try:
    conn = mysql.connector.connect(...)
except mysql.connector.Error as err:
    print(f"Error: {err}")
finally:
    if conn.is_connected():
        conn.close()
```

12.11 CRUD Operations

12.11.1 Create

Inserting new records into a table using `INSERT` statements.

12.11.2 Read

Retrieving existing records using `SELECT` queries.

12.11.3 Update

Modifying existing records using `UPDATE` statements.

12.11.4 Delete

Removing records from a table using `DELETE` statements.

12.12 Sample Program – MySQL CRUD Demonstration

```
import mysql.connector
import json
def get_connection():
    return mysql.connector.connect(
        host="localhost",
        user="root",
        password="password",
        database="test_db"
    )
def create_table():
    try:
        conn = get_connection()
        cursor = conn.cursor()
        cursor.execute(
            "CREATE TABLE IF NOT EXISTS
inventory (\"
            \"id INT AUTO_INCREMENT PRIMARY
KEY,\"
            \"item_name VARCHAR(100),\"
            \"quantity INT,\"
            \"details JSON\"
            \")\"
        )
        print("Table created successfully.")
    except mysql.connector.Error as err:
        print(f"Error: {err}")
    finally:
        if conn.is_connected():
            cursor.close()
            conn.close()
def insert_item(name, qty, details_dict):
    try:
        conn = get_connection()
        cursor = conn.cursor()
        sql = "INSERT INTO inventory
(item_name, quantity, details) VALUES (%s, %s,
%s)"
        cursor.execute(sql, (name, qty,
json.dumps(details_dict)))
        conn.commit()
        print(f"Inserted item ID:
{cursor.lastrowid}")
    except Exception as e:
        conn.rollback()
        print(f"Insert failed: {e}")
    finally:
        conn.close()
```

12.13 Assignments

1. Which method is used to permanently save changes made by an INSERT or UPDATE operation?
A) cursor.save() B) conn.commit() C) cursor.execute() D) conn.close()
2. Explain why parameterized queries are safer than using string formatting for SQL commands.
3. Write a Python script that connects to a MySQL database, retrieves all users where age > 18, and prints their names. Include proper exception handling.
4. The following code inserts a record, but the data does not appear in the database. Identify and fix the issue.

conn = mysql.connector.connect(...)

cursor = conn.cursor()

cursor.execute("INSERT INTO log (msg)
VALUES ('test')")

conn.close()
5. What is the purpose of calling conn.rollback() inside an except block?

12.14 Conclusion

Python and MySQL integration enables developers to build powerful, data-driven applications. By using secure parameterized queries, proper transaction management, and structured exception handling, Python programs can interact with databases reliably and efficiently. Mastering these concepts is essential for backend development, automation tasks, and full-stack application development.