Module-3
Advance Python Programming
(theory)

-Harsh Chauhan

1. Printing on Screen

1)Introduction to the print() function in Python.

- print() with files
  - In file handling, print() can send data directly to a file using the file parameter.
  - By default, print() writes to the console (screen), but with file=, it writes to the given file object.

- Basic syntax for file handling
  - Open a file first, for example: f = open("data.txt", "w").
  - Use print("Hello", file=f) to write the text "Hello" inside that file.

- Modes used with print() and files
  - Common modes: "w" for write (overwrites file), "a" for append (adds at the end).
  - After using print() with a file, always close the file using f.close() to save the data properly

- Why use print() in file handling?
  - It is an easy way to log messages or save program output for later use.
  - Syntax is simple because you can print multiple values just like normal print, but they go into a file

2) Formatting outputs using f-strings and format().

- Using f-strings in file handling
  - f-strings are written as f"some text {variable}" and allow variables directly inside {}.
  - In file handling, you combine f-strings with print(..., file=f) or f.write(...) to save formatted text.
  - Example:

```
        with open("data.txt", "w") as f:
        name = "Raj"
        marks = 89
        print(f"Student: {name}, Marks: {marks}",
file=f)
```
→ neatly stores the line in the file.

- Using format() in file handling
  - format() uses placeholders {} inside a string and then fills them using .format(...).
  - You can use it with print and the file parameter to write formatted output.
  - Example:

```
with open("report.txt", "w") as f:
  for i in range(1, 6):
    print("Number: {}, Square: {}".format(i, i*i),
file=f)
```
→ each line in file is clean and aligned logically.

- Why use f-strings and format() in files?
    - They help create structured reports like tables, bills, or logs (e.g., "ID: 01 | Name: Raj | Marks: 89").
    - Both give control over number formats, like f"{price:.2f}" or "Price: {:.2f}".format(price) to save fixed two-decimal values in files.

2. Reading Data from Keyboard

1) Using the input() function to read user input from the keyboard.

- Basic input() usage
  - `name = input("Enter name: ")` gets string input from user and stores in variable.
  - Always returns string type, so convert if needed: `age = int(input("Enter age: "))`.

- With file handling
  - Get input from the user, then write to the file using print or write().
  - Example:
    ```
    with open("student.txt", "w") as f:
      name = input("Enter student name: ")
      print(f"Name: {name}", file=f)
    ```
    User types name, it gets saved neatly in file.

- Common pattern in assignments
  - Read multiple inputs (name, marks) and save formatted records to file.
  - Use loop: `for i in range(3): data = input(); print(data, file=f)`.

- Key points
    - input() for interactive programs that create/update files based on user data.
    - Combine with f-strings/format() for clean file output from keyboard input.

2) Converting user input into different data types (e.g., int, float, etc.)

- Common conversions
  - int(input()) → converts string to integer (e.g., "25" → 25).
  - float(input()) → converts to decimal (e.g., "25.5" → 25.5).
  - bool(input()) → converts to True/False (empty string is False).

- In file handling context
  - Get numbers from the user, convert, then write to file.
  - Example:

```
marks = int(input("Enter marks: "))
with open("scores.txt", "a") as f:
 print(f"Marks: {marks}", file=f)
```

- Error handling tip
  - Use try-except for safe conversion: try: num = int(input()); except ValueError: print("Invalid number").
  - Prevents crashes if user enters letters instead of numbers

3. Opening and Closing Files

1) Opening files in different modes ('r', 'w', 'a', 'r+', 'w+').

- Read Mode ('r')
  - Opens file for reading only; file pointer starts at beginning.
  - File must exist, or it raises FileNotFoundError.

- Write Mode ('w')
  - Opens for writing; overwrites file if it exists, creates new if not.
  - File pointer starts at beginning; good for new content.

- Append Mode ('a')
  - Opens for writing; adds data at end without overwriting.
  - Creates file if it doesn't exist; pointer at end.

- Read-Write Mode ('r+')
  - Allows both reading and writing; file must exist.
  - Pointer starts at beginning; use seek() to move position.

- Read-Write Mode ('w+')
  - Read and write; overwrites files or creates new ones.
  - Pointer at beginning; truncates existing content

2) Using the open() function to create and access files.

- Basic Syntax
    - file_object = open("filename.txt", "mode") opens/creates file in specified mode.
    - Always use with open() for automatic closing: safer and cleaner code.

- Creating Files
    - Modes 'w', 'a', 'w+' automatically create a file if it doesn't exist.
    - Example: with open("newfile.txt", "w") as f: f.write("Hello") → creates newfile.txt.

- Accessing Existing Files
    - Use 'r' mode to read existing files safely.
    - Example: with open("data.txt", "r") as f: content = f.read() → reads the whole file.

3) Closing files using close().

- Why close files?
  - Saves changes to disk (due to buffering) and prevents data loss.
  - Frees memory; too many open files can crash programs

- Basic syntax

  -
    ```
    f = open("file.txt", "w")
    f.write("data")
    f.close()
    ```
    → always call after operations.

4. Reading and Writing Files

1) Reading from a file using read(), readline(), readlines().

- read() method
  - Reads entire file content as one string (or specified bytes if given).
  - f.read() → gets the whole file; f.read(10) → first 10 characters.

- readline() method
  - Reads one line at a time (includes newline character \n).
  - f.readline() → first line; call again for next line.

- readlines() method
  - Reads all lines into a list (each line as a separate string).
  - lines = f.readlines() → ['line1\n', 'line2\n']

2) Writing to a file using write() and writelines().

- write() method
  - Writes a single string to file (no automatic newline).
  - `f.write("Hello\n")` → adds "Hello" + newline to file.

- writelines() method
  - Writes multiple strings or list of strings at once.
  - `f.writelines(["Line1\n", "Line2\n"])` → adds both lines together.

- Key differences
  - write() for one string; writelines() for list/iterable of strings.
  - Neither adds newlines automatically (add `\n` manually).

5. Exception Handling

1) Introduction to exceptions and how to handle them using try, except, and finally.

Exceptions are errors that occur during program execution, like division by zero or file not found.

- try-except structure
  - try: Put risky code here that might cause errors.
  - except: Handles the error gracefully instead of crashing.
  - Example:

```
try:
    num = int(input("Enter number: "))
    print(10/num)
except ValueError:
    print("Enter valid number")
except ZeroDivisionError:
    print("Cannot divide by zero")
```

- finally block
  - finally: Always runs, even if error occurs or not (good for cleanup).
  - Used to close files or release resources safely.
  - Example:

```
try:
    f = open("data.txt")
    # read file
except FileNotFoundError:
    print("File missing")
finally:
    f.close()  # Always closes
```

File handling example

```
try:
    with open("scores.txt", "r") as f:  # auto-close, but still use
try
        data = f.read()
except FileNotFoundError:
    print("Create file first")
finally:
    print("Operation complete")
```

2) Understanding multiple exceptions and custom exceptions.

In Python, multiple exceptions let you handle different error types, and custom exceptions let you create your own error types for specific situations.

- Handling multiple exceptions
    - Use separate except blocks when each exception needs a different message or action.

```
try:
    a = int(input("Enter number: "))
    b = int(input("Enter divisor: "))
    print(a / b)
except ValueError:
    print("Please enter numbers only")
except ZeroDivisionError:
    print("Cannot divide by zero")
```

    - Use a single except with a tuple when handling many exceptions in the same way.

```
try:
    num = int(input("Enter number: "))
    print(10 / num)
except (ValueError, ZeroDivisionError) as e:
    print("Error:", e)
```

- Creating custom exceptions
    - Custom exceptions are classes that usually inherit from Exception.
    -

```
class NegativeMarksError(Exception):
    pass
```

- Raise them using raise when a special condition occurs.

```
marks = int(input("Enter marks: "))
if marks < 0:
    raise NegativeMarksError("Marks cannot be
negative")
```

- Handling custom exceptions
    - Catch custom exceptions like builtin ones using exceptions.

```
try:
    marks = int(input("Enter marks: "))
    if marks < 0:
        raise NegativeMarksError("Marks cannot be
    negative")
except NegativeMarksError as e:
        print("Custom error:", e)
```

6. Class and Object (OOP Concepts)

1) Understanding the concepts of classes, objects, attributes, and methods in Python.

Classes define blueprints for objects, while objects are instances with data (attributes) and functions (methods).

- Classes
  - Blueprint/template for creating objects with shared structure.
  - Defined using class ClassName:.

- Objects
  - Actual instances created from class using obj = ClassName().
  - Each object has its own copy of instance attributes.

- Attributes
  - Class attributes: Shared by all objects (defined directly in class).
  - Instance attributes: Unique to each object (set in __init__).
  - Access: obj.attribute or Class.attribute.

- Methods
  - Functions inside class that work on object data (use self).
  - __init__(self) is constructor to initialize attributes

2) Difference between local and global variables.

Local variables exist only inside a function/block, while global variables are defined outside and can be used in many functions.

- Local variables
  - Defined inside a function or block and can be accessed only there.
  - Created when the function starts and destroyed when the function ends.

- Global variables
  - Defined outside all functions, usually at the top of the program
  - Accessible from any function in the file (can be read everywhere)

- Effect of changes
  - Changing a local variable affects only that function, not the rest of the program.
  - Changing a global variable (using global keyword inside a function) affects its value everywhere.

7. Inheritance

1) Single, Multilevel, Multiple, Hierarchical, and Hybrid inheritance in Python.

Python supports five main types of inheritance: single, multilevel, multiple, hierarchical, and hybrid.

- Single Inheritance
  - One child class inherits from one parent class.
  - Example: class Dog(Animal): → Dog gets Animal's methods.

- Multilevel Inheritance
  - Chain-like: Grandchild → Child → Parent.
  - Example: class Bulldog(Dog): class Dog(Animal): → Bulldog accesses Animal too.

- Multiple Inheritance
  - One child inherits from multiple parents.
  - Example: class Duck(Flyer, Swimmer): → Duck can fly() and swim().

- Hierarchical Inheritance
  - One parent, multiple children.
  - Example: class Dog(Animal): class Cat(Animal): → Both inherit speak().

- Hybrid Inheritance
  - Mix of above types (multiple + hierarchical usually).
  - Example: Multiple children from multilevel chain

2) Using the super() function to access properties of the parent class.

super() calls parent class methods/constructors from child classes without naming the parent directly.

- Basic usage
  - super().__init__(args) calls parent's constructor to initialize inherited attributes.
  - super().method() calls parent's method from overridden child method.

- Why use super()?
  - Cleaner code; works automatically in multiple inheritance (follows MRO).
  - Avoids hardcoding parent class name, which breaks if inheritance changes

8. Method Overloading and Overriding

1) Method overloading: defining multiple methods with the same name but different parameters.

- Method overloading in theory
  - Same method name, different number or type of parameters in the same class.
  - Example idea: add(int a, int b) and add(int a, int b, int c) are two overloaded methods.

- Python's reality
  - Python does not support true method overloading: if you define same method name twice, the last one overwrites the earlier one.
  - Because Python is dynamically typed, there is no compile-time selection based on parameter types.

- How to simulate overloading in Python
  - Use default arguments in a single method:

```
class Calc:
    def add(self, a=None, b=None, c=None):
        if a is not None and b is not None and c is not None:
            return a + b + c
        elif a is not None and b is not None:
            return a + b
```

- One method handles “2‑argument” and “3‑argument” cases.
- Use variable‑length arguments *args/**kwargs and write logic inside based on count/usage

Definition line: “Method overloading is using the same method name with different parameter options; in Python it is simulated using default/variable‑length arguments, not supported natively like in Java/C++.”

2) Method overriding: redefining a parent class method in the child class.

Method overriding redefines a parent class method in the child class with the same name and parameters.

- How it works
  - Child class provides its own implementation for inherited method.
  - Python calls the child's version when object is of child type (runtime polymorphism).

Basic example

```
class Animal:
    def speak(self):
        print("Animal makes sound")

class Dog(Animal):
    def speak(self):    # Overrides parent's speak()
        print("Dog barks: Woof!")

dog = Dog()
dog.speak()             # Output: Dog barks: Woof!
```

Parent's speak() is replaced by child's version.

- Calling parent method (using super())
    - Use super().method() inside overridden method to call parent's version too.

```
class Dog(Animal):
    def speak(self):
        super().speak()        # Calls Animal's speak()
        print("Dog barks: Woof!")
# Output: Animal makes sound\nDog barks: Woof!
```

9. SQLite3 and PyMySQL (Database Connectors)

1) Introduction to SQLite3 and PyMySQL for database connectivity.

SQLite3 and PyMySQL are Python libraries for connecting to databases: SQLite3 for lightweight local files, PyMySQL for remote MySQL servers.

- SQLite3 Introduction
  - Built-in module (no installation needed); uses single `.db` file as database.
  - Perfect for small apps, assignments, learning database basics.
  - Import: `import sqlite3`; connect: `conn = sqlite3.connect('school.db')`.

- PyMySQL Introduction
  - Third-party library for MySQL servers (install: `pip install PyMySQL`).
  - Connects to remote MySQL databases (like hosted on servers).
  - Import: `import pymysql`; connect: `conn = pymysql.connect(host='localhost', user='root', password='', db='school')`.

2) Creating and executing SQL queries from Python using these connectors.

Both SQLite3 and PyMySQL use cursor objects to execute SQL queries like CREATE, INSERT, SELECT, UPDATE, DELETE.

- Basic pattern (same for both)

```
conn = connect_database()
cursor = conn.cursor()
cursor.execute("SQL_QUERY")
conn.commit()  # For INSERT/UPDATE/DELETE
data = cursor.fetchall()  # For SELECT
conn.close()
```

- SQLite3 examples

```
import sqlite3
conn = sqlite3.connect('school.db')
cursor = conn.cursor()

# Create table
cursor.execute("CREATE TABLE IF NOT EXISTS
students (id INTEGER PRIMARY KEY, name TEXT,
marks INTEGER)")

# Insert data
```

```
cursor.execute("INSERT INTO students (name, marks)
VALUES ('Raj', 89)")

# Select data
cursor.execute("SELECT * FROM students WHERE
marks > 50")
students = cursor.fetchall()  # Returns list of tuples
print(students)  # [('Raj', 89), ...]

conn.commit()
conn.close()
```

- PyMySQL examples

```
import pymysql
conn = pymysql.connect(host='localhost', user='root',
password='', db='school')
cursor = conn.cursor()

# Create table
cursor.execute("CREATE TABLE IF NOT EXISTS
students (id INT AUTO_INCREMENT PRIMARY
KEY, name VARCHAR(50), marks INT)")

# Insert multiple rows
cursor.executemany("INSERT INTO students (name,
marks) VALUES (%s, %s)", [('Raj', 89), ('Priya', 92)])
```

```
# Select with parameters
cursor.execute("SELECT name FROM students WHERE
marks > %s", (80,))
names = cursor.fetchall()
print(names)

conn.commit()
conn.close()
```

- Key differences
  - SQLite3: ? placeholders; PyMySQL: %s placeholders.
  - SQLite3: INTEGER PRIMARY KEY; PyMySQL: INT AUTO_INCREMENT PRIMARY KEY

10. Search and Match Functions

1) Using re.search() and re.match() functions in Python’s re module for pattern matching.

re.search() and re.match() find patterns in strings using regular expressions from Python's re module.

- Import and basic setup
  - import re to use regex functions.
  - Patterns use special characters: \d (digit), \w (word), . (any char), + (1+ times).

- re.match()
  - Matches pattern only at the beginning of string.
  - Returns match object if starts with pattern, else None.

```
match = re.match(r'\d+', "123abc")  # Matches → <match object>
match = re.match(r'\d+', "abc123")  # No match → None
```

- re.search()
  - Searches for pattern anywhere in string.
  - Returns first match found, regardless of position.

```
search = re.search(r'\d+', "abc123def")  # Matches →
<match object> (finds 123)
search = re.search(r'\d+', "abc")        # No match → None
```

- Getting matched text

```
if match := re.search(r'\d+', "phone: 9876543210"):
    print(match.group())  # 9876543210
    print(match.start())  # Position where match starts
```

2) Difference between search and match.

re.match() checks for pattern only at the start of the string.
re.search() looks for the pattern anywhere in the string.

Feature	**re.match()**	**re.search()**
Search position	Beginning only	Anywhere
Example: `re.match(r'\d+', "abc123")`	None	Finds "123"
Example: `re.match(r'\d+', "123abc")`	Finds "123"	Finds "123"

- Practical Examples

```
import re

text = "Phone: 9876543210"

# match() - must start with digits
print(re.match(r'\d+', text))  # None (starts with "Phone")

# search() - finds digits anywhere
```

```
print(re.search(r'\d+', text).group())  # 9876543210
```

- When to use each
  - match(): Validate phone numbers, commands, file extensions (must start correctly).
  - search(): Extract data from text, logs, HTML (find anywhere).