Module 8) Advance Python Programming

1. Printing on Screen

Theory:

Introduction to the print() function in Python.

The print() function in Python displays specified messages or variables to the screen or other standard output devices.

Formatting outputs using f-strings and format().

Python uses f-strings (e.g., f"{var}") and the .format() method to embed expressions and variables directly into string literals for readable, dynamic output.

Lab:

Write a Python program to print a formatted string using print() and f-string.

```
name = "python"
print(f"hello, {name}!")
```

Practical Example:

1. Write a Python program to print “Hello, World!” on the screen.

```
print("hello, world!")
```

2. Reading Data from Keyboard

Theory:

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

The input() function pauses program execution to let the user type a string, which is then assigned to a variable for further use.

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

```
age = int(input("enter your age: "))
price = float(input("enter the price: "))
```

Lab:

Write a Python program to read a name and age from the user and print a formatted output.

```
name = input("enter your name: ")
age = int(input("enter your age: "))
print(f"hello {name}, you are {age} years old!")
```

Practical Example:

2) Write a Python program to read a string, an integer, and a float from the keyboard and display them.

```
text = input("enter a string: ")
number = int(input("enter an integer: "))
decimal = float(input("enter a float: "))
```

```
print(f"string: {text}, integer: {number}, float: {decimal}")
```

3. Opening and Closing Files

Theory:

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

Python manages files using the open() function, where the specified mode determines the interaction: 'r' is the default for reading and fails if the file is missing; 'w' opens a file for writing, either creating a new one or overwriting existing content; 'a' appends new data to the end of an existing file; 'r+' allows both reading and writing starting from the beginning; and 'w+' enables both reading and writing but truncates the file to zero length first.

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

The open() function serves as the primary gateway for file interaction in Python, requiring a filename and an optional mode to determine if you are reading, writing, or appending data. When a file is opened for writing with 'w', Python creates the file if it doesn't already exist, while the 'r' mode allows you to access and process the contents of an existing file. To ensure resources are managed correctly and files are properly closed after use, it is best practice to use the open() function within a with statement block.

Closing files using close().

Calling the .close() method on a file object manually releases system resources and ensures all data buffered in memory is fully written to the disk.It is essential to close files after use because leaving them open can lead to data corruption or prevent other programs from accessing the file.

Lab:

Write a Python program to open a file in write mode, write some text, and then close it.

```
file = open("example.txt", "w")
file.write("hello, this is a test.")
file.close()
```

Practical Example:

3) Write a Python program to create a file and write a string into it.

```
file = open("myfile.txt", "w")
file.write("this is a python string.")
file.close()
```

4. Reading and Writing Files

Theory:

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

Python provides the .read() method to extract a file's entire content as one string, while .readline() retrieves a single line and .readlines() collects all lines into a formatted list. These methods allow for precise control over data processing, whether you need to analyze the full text at once or iterate through it line by line.

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

The write() method takes a single string and inserts it into a file, whereas writelines() accepts a list of strings and writes them sequentially without adding newlines automatically. Utilizing these methods allows you to either append a continuous block of text or export structured data from a list into a permanent file.

Lab:

Write a Python program to read the contents of a file and print them on the console.

```
file = open("myfile.txt", "r")
content = file.read()
print(content)
file.close()
```

Write a Python program to write multiple strings into a file.

```
lines = ["first line\n", "second line\n", "third line\n"]
file = open("data.txt", "w")
file.writelines(lines)
file.close()
```

Practical Examples:

4) Write a Python program to create a file and print the string into the file.

```
file = open("output.txt", "w")
file.write("this is the string written into the file.")
file.close()
```

5) Write a Python program to read a file and print the data on the console.

```
file = open("data.txt", "r")
content = file.read()
print(content)
file.close()
```

6) Write a Python program to check the current position of the file cursor using tell().

```
file = open("data.txt", "r")
print(f"current cursor position: {file.tell()}")
content = file.read(5)
print(f"cursor position after reading 5 characters: {file.tell()}")
file.close()
```

5. Exception Handling

Theory:

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

Python uses the try block to monitor code for potential errors, while the except block catches and handles specific exceptions to prevent program crashes. The finally block ensures that a specified piece of code executes regardless of whether an error occurred, making it ideal for essential cleanup tasks like closing files.

Understanding multiple exceptions and custom exceptions.

Python allows you to handle various error types by using multiple except blocks or a single tuple to provide specific responses for different failures like ValueError or TypeError. You can also define custom exceptions by creating a new class that inherits from the built-in Exception class, allowing you to trigger specialized errors tailored to your program's unique logic.

Lab:

Write a Python program to handle exceptions in a simple calculator (division by zero, invalid input).

```
try:
    num1 = float(input("enter first number: "))
    num2 = float(input("enter second number: "))
    result = num1 / num2
    print(f"result: {result}")
except ZeroDivisionError:
    print("error: cannot divide by zero.")
except ValueError:
    print("error: please enter valid numeric values.")
```

Write a Python program to demonstrate handling multiple exceptions.

```
try:
    data = input("enter a number to divide 100: ")
    number = int(data)
    result = 100 / number
    print(f"result is {result}")
except ValueError:
    print("error: invalid input. please enter a numeric integer.")
except ZeroDivisionError:
    print("error: cannot divide by zero.")
except Exception as e:
    print(f"an unexpected error occurred: {e}")
```

Practical Examples:

7) Write a Python program to handle exceptions in a calculator.

```
try:
    num1 = float(input("enter first number: "))
    num2 = float(input("enter second number: "))
    operation = input("enter operation (+, -, *, /): ")

    if operation == "+":
        print(f"result: {num1 + num2}")
    elif operation == "-":
        print(f"result: {num1 - num2}")
    elif operation == "*":
        print(f"result: {num1 * num2}")
    elif operation == "/":
        print(f"result: {num1 / num2}")
    else:
        print("error: invalid operation.")
except ZeroDivisionError:
    print("error: division by zero is not allowed.")
except ValueError:
    print("error: please enter valid numeric values.")
except Exception as e:
    print(f"an unexpected error occurred: {e}")
```

8)Write a Python program to handle multiple exceptions (e.g., file not found, division by zero).

```
try:
    file_name = input("enter the filename to read a divisor from: ")
    with open(file_name, "r") as file:
        data = file.read().strip()
        divisor = float(data)
        result = 100 / divisor
        print(f"result: {result}")
except FileNotFoundError:
    print("error: the specified file was not found.")
except ZeroDivisionError:
    print("error: the file contains zero, and division by zero is undefined.")
except ValueError:
    print("error: the file does not contain a valid numeric value.")
```

```
except Exception as e:
    print(f"an unexpected error occurred: {e}")
```

9) Write a Python program to handle file exceptions and use the finally block for closing the file.

```
try:
    file = open("data.txt", "r")
    content = file.read()
    print(content)
except FileNotFoundError:
    print("error: the file does not exist.")
except Exception as e:
    print(f"an unexpected error occurred: {e}")
finally:
    if 'file' in locals() and not file.closed:
        file.close()
        print("file has been closed successfully.")
```

10) Write a Python program to print custom exceptions.

```
class MyCustomError(Exception):
    pass

try:
    raise MyCustomError("this is a custom exception message")
except MyCustomError as e:
    print(f"caught an error: {e}")
```

6. Class and Object (OOP Concepts)

Theory:

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

In Python, a class serves as a blueprint for creating objects, which are specific instances that encapsulate both data and functionality. These objects store their state in attributes (variables) and perform actions through methods (functions), allowing you to model real-world entities within your code.

Difference between local and global variables.

Local variables are defined inside a function and only accessible within it, while global variables are defined outside all functions and can be accessed throughout the entire program.

Lab:

Write a Python program to create a class and access its properties using an object.

```
class student:
    def __init__(self, name, age):
        self.name = name
        self.age = age

s1 = student("alice", 21)
print(f"name: {s1.name}")
print(f"age: {s1.age}")
```

Practical Examples:

11) Write a Python program to create a class and access the properties of the class using an object.

```
class car:
    def __init__(self, brand, year):
        self.brand = brand
        self.year = year

my_car = car("toyota", 2024)
print(f"brand: {my_car.brand}")
print(f"year: {my_car.year}")
```

12) Write a Python program to demonstrate the use of local and global variables in a class.

```
global_var = "i am a global variable"

class demo:
    class_var = "i am a class variable (shared by all objects)"

    def show_variables(self):
        local_var = "i am a local variable (only in this method)"
        print(global_var)
        print(self.class_var)
        print(local_var)

obj = demo()
obj.show_variables()
```

7. Inheritance

Theory:

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

Single Inheritance: A child class inherits the attributes and methods from a single parent class.
Multilevel Inheritance: A child class inherits from a parent class, which in turn inherits from another grandparent class.
Multiple Inheritance: A single child class inherits features from more than one base class simultaneously.
Hierarchical Inheritance: Multiple child classes inherit from a single common parent class.
Hybrid Inheritance: A complex combination of two or more types of inheritance within a single program.
Using the super() function to access properties of the parent class.

Lab:

Write Python programs to demonstrate different types of inheritance (single, multiple, multilevel, etc.).

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

```
class parent:
    def speak(self):
        print("parent is speaking.")

class child(parent):
    def play(self):
        print("child is playing.")

obj = child()
obj.speak()
obj.play()
```

In multilevel inheritance, a class inherits from a child class, forming a shared relationship across generations.

```
class grandparent:
    def greet(self):
        print("hello from grandparent.")

class parent(grandparent):
    def walk(self):
        print("parent is walking.")

class child(parent):
    def run(self):
        print("child is running.")

obj = child()
obj.greet()
obj.walk()
obj.run()
```

Multiple inheritance allows a child class to inherit features from more than one base class.

```
class father:
    def height(self):
        print("tall height inherited.")

class mother:
    def eyes(self):
        print("blue eyes inherited.")

class child(father, mother):
    def profile(self):
        print("child has mixed traits.")

obj = child()
obj.height()
obj.eyes()
```

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

```
class vehicle:
    def info(self):
        print("this is a vehicle.")

class car(vehicle):
    def drive(self):
        print("driving the car.")

class bike(vehicle):
    def ride(self):
        print("riding the bike.")

c = car()
b = bike()
c.info()
```

```
b.info()
```

Practical Examples:

13) Write a Python program to show single inheritance.

```
class parent:
    def greet(self):
        print("hello from the parent class.")

class child(parent):
    def display(self):
        print("hello from the child class.")

# creating an object of the child class
obj = child()
obj.greet()
obj.display()
```

14) Write a Python program to show multilevel inheritance.

```
class grandfather:
    def legacy(self):
        print("grandfather: i built the foundation.")

class father(grandfather):
    def career(self):
        print("father: i am managing the business.")

class son(father):
    def innovation(self):
        print("son: i am adding new technology.")

# creating an object of the youngest class
obj = son()
obj.legacy()
obj.career()
obj.innovation()
```

15) Write a Python program to show multiple inheritance.

```
class mother:
    def eyes(self):
        print("inherited blue eyes.")
```

```
class father:
    def height(self):
        print("inherited tall height.")

class child(mother, father):
    def personality(self):
        print("unique personality developed.")

# creating an object of the child class
obj = child()
obj.eyes()
obj.height()
obj.personality()
```

16) Write a Python program to show hierarchical inheritance.

```
class vehicle:
    def base_info(self):
        print("this is a generic vehicle.")

class car(vehicle):
    def drive(self):
        print("the car is driving on four wheels.")

class bike(vehicle):
    def ride(self):
        print("the bike is riding on two wheels.")

# creating objects for different child classes
c = car()
b = bike()

c.base_info()
c.drive()

b.base_info()
b.ride()
```

17) Write a Python program to show hybrid inheritance.

```
class school:
    def info(self):
        print("this is a school.")

class teacher(school):
    def teach(self):
        print("the teacher is teaching.")

class parent:
    def support(self):
        print("the parent provides support.")

class student(teacher, parent):
    def learn(self):
        print("the student is learning.")
obj = student()
obj.info()
obj.teach()
obj.support()
obj.learn()
```

18) Write a Python program to demonstrate the use of super() in inheritance.

```
class employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        print(f"employee {self.name} initialized.")

class manager(employee):
    def __init__(self, name, salary, department):
        # using super() to call the parent constructor
        super().__init__(name, salary)
        self.department = department
        print(f"manager of {self.department} department initialized.")

    def show_details(self):
        print(f"name: {self.name}, salary: {self.salary}, dept: {self.department}")
m1 = manager("john", 80000, "it")
m1.show_details()
```

8. Method Overloading and Overriding

Theory:

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

In Python, method overloading refers to defining multiple methods with the same name but different signatures; however, since Python only recognizes the last defined method, it is typically achieved using default arguments or variable-length arguments (*args, **kwargs). This allows a single method to perform different actions depending on the number or types of inputs provided during the call.

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

Method overriding allows a child class to provide a specific implementation of a method that is already defined in its parent class. When the method is called on an object of the child class, Python executes the version inside the child rather than the one in the parent.

Lab:

Write Python programs to demonstrate method overloading and method overriding.

Method Overloading

```
class calculator:
    def add(self, a, b, c=0):
        return a + b + c

calc = calculator()
print(f"sum of two: {calc.add(10, 20)}")
print(f"sum of three: {calc.add(10, 20, 30)}")
```

Method Overriding

```
class animal:
    def sound(self):
        print("animal makes a generic sound.")

class dog(animal):
    def sound(self):
        print("dog barks: woof woof!")

class cat(animal):
    def sound(self):
        print("cat meows: meow meow!")

# demonstrate overriding
```

```
my_dog = dog()
my_cat = cat()

my_dog.sound()
my_cat.sound()
```

Practical Examples:

19) Write a Python program to show method overloading.

```
class calculator:
    def add(self, a, b, c=None):
        if c is not None:
            return a + b + c
        else:
            return a + b
calc = calculator()
print(f"sum of two numbers: {calc.add(10, 20)}")
print(f"sum of three numbers: {calc.add(10, 20, 30)}")
```

20) Write a Python program to show method overriding.

```
class parent:
    def display(self):
        print("this is the parent class implementation.")

class child(parent):
    def display(self):
        print("this is the child class overridden implementation.")
p_obj = parent()
c_obj = child()

p_obj.display()
c_obj.display()
```

9. SQLite3 and PyMySQL (Database Connectors)

Theory:

Introduction to SQLite3 and PyMySQL for database connectivity.

SQLite3 is a built-in Python library that provides a lightweight, serverless, and file-based database ideal for local storage and prototyping. PyMySQL is an external library used to connect Python to a MySQL server, enabling complex, multi-user database interactions through a client-server architecture.

Creating and executing SQL queries from Python using these connectors.

To execute SQL queries, you first establish a connection and create a cursor object, which acts as the interface for sending commands to the database. You then use the cursor.execute() method to run your SQL statement and, for data-modifying queries, call connection.commit() to save the changes permanently.

Lab:

Write a Python program to connect to an SQLite3 database, create a table, insert data, and fetch data.

```
import sqlite3
# 1. connect to database (creates file if it doesn't exist)
conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# 2. create a table
cursor.execute('''create table if not exists users
              (id integer primary key, name text, age integer)''')

# 3. insert data
cursor.execute("insert into users (name, age) values ('alice', 25)")
cursor.execute("insert into users (name, age) values ('bob', 30)")

# 4. commit changes
conn.commit()

# 5. fetch and display data
cursor.execute("select * from users")
rows = cursor.fetchall()
for row in rows:
   print(row)
conn.close()
```

Practical Examples:

21) Write a Python program to create a database and a table using SQLite3.

```
import sqlite3
# 1. connect to database (creates file if it doesn't exist)
connection = sqlite3.connect('my_database.db')
cursor = connection.cursor()

# 2. create a table
cursor.execute('''create table if not exists products
              (id integer primary key, name text, price real)''')

# 3. commit and close
connection.commit()
connection.close()
print("database and table created successfully.")
```

22) Write a Python program to insert data into an SQLite3 database and fetch it.

```
import sqlite3
# 1. connect to database and create a cursor
connection = sqlite3.connect('example.db')
cursor = connection.cursor()
# 2. ensure the table exists
cursor.execute('create table if not exists items (id integer primary key, name text, quantity
integer)')
# 3. insert data into the table
cursor.execute("insert into items (name, quantity) values ('laptop', 10)")
cursor.execute("insert into items (name, quantity) values ('mouse', 50)")
# 4. commit the changes to the database
connection.commit()
# 5. fetch the data from the table
cursor.execute("select * from items")
rows = cursor.fetchall()
# 6. display the fetched records
for row in rows:
    print(row)
# 7. close the connection
connection.close()
```

10. Search and Match Functions
Theory:
Using re.search() and re.match() functions in Python's re module for pattern matching.
The re.match() function checks for a pattern only at the beginning of a string, while re.search() scans the entire string to find the first location where the pattern occurs.

Difference between search and match.
re.match() checks for a pattern only at the beginning of the string, whereas re.search() scans the entire string to find the first occurrence of the pattern.
Lab:
Write a Python program to search for a word in a string using re.search().
```
import re

text = "the quick brown fox jumps over the lazy dog"
pattern = "fox"

# searching for the pattern in the string
result = re.search(pattern, text)

if result:
   print(f"match found: {result.group()} at index {result.start()}")
else:
   print("no match found.")
```
Write a Python program to match a word in a string using re.match().
```
import re

text = "python is an amazing language"
pattern = "python"

# re.match checks only at the beginning of the string
result = re.match(pattern, text)

if result:
   print(f"match found at the start: {result.group()}")
else:
   print("no match found at the beginning.")
```

Practical Examples:

23) Write a Python program to search for a word in a string using re.search().

```
import re

text = "learning python is fun and powerful"
pattern = "python"

# re.search scans the entire string for the first match
result = re.search(pattern, text)

if result:
   print(f"pattern '{pattern}' found at index: {result.start()}")
else:
   print("pattern not found")
```

24) Write a Python program to match a word in a string using re.match().

```
import re

text = "apple is a fruit"
pattern = "apple"

# re.match only looks for the pattern at the very beginning of the string
result = re.match(pattern, text)

if result:
   print(f"match found: {result.group()}")
else:
   print("no match found at the start of the string.")
```