**Module 8) Advance Python Programming**

**1. Printing on Screen**

**Theory:**

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

The print() function is used to display output on the screen.

### Basic Syntax:

### print("Hello, World!")

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

**Lab:**



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

### ****Using f-strings:****

name = "Arjun"

age = 22

print(f"My name is {name} and I am {age} years old.")

## Summary

| **Method** | **Syntax Example** | **Python Version** |
| --- | --- | --- |
| print() | print("Hello") | All versions |
| f-strings | f"My name is {name}" | 3.6+ |
| .format() | "My name is {}".format(name) | 2.7 / 3.x |

**Practical Example:**

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

# Simple program to print a message

print("Hello, World!")

1. Reading Data from Keyboard

name = input("Enter your name: ")

print(f"Hello, {name}!")

**Theory:**

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

## ****Using the**** input() ****Function in Python****

### What is input()?

The input() function is used in Python to take **input from the user** through the keyboard.  
It pauses the program and waits for the user to type something and press Enter.

### Syntax:

variable\_name = input("Prompt message")

"Prompt message" is optional and is shown to the user as a message.

The input() function **always returns a string** (text), even if the user enters numbers.

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

| **Input Type** | **Example Code** | **Output Type** |
| --- | --- | --- |
| String | name = input("Enter name: ") | str |
| Integer | age = int(input("Enter age: ")) | int |
| Float | price = float(input("Enter price: ")) | float |
| List | values = input().split() | list[str] |
| Boolean | flag = input().lower() == "yes" | bool |

**Lab:**

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

# Read input from the user

name = input("Enter your name: ")

age = int(input("Enter your age: "))

# Print formatted output

print(f"Hello, {name}! You are {age} years old.")

**Practical Example:**

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

# Reading inputs from the user

string\_value = input("Enter a string: ")

int\_value = int(input("Enter an integer: "))

float\_value = float(input("Enter a float: "))

# Displaying the values

print(f"\nYou entered:\nString: {string\_value}\nInteger: {int\_value}\nFloat: {float\_value}")

**3. Opening and Closing Files**

**Theory:**

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

| **Mode** | **Description** | **File must exist?** | **File pointer position** |
| --- | --- | --- | --- |
| 'r' | Read mode — opens file for reading (default) | Yes | Start of the file |
| 'w' | Write mode — opens file for writing (creates or overwrites) | No | Start of the file |
| 'a' | Append mode — opens file for appending (adds at the end) | No | End of the file |
| 'r+' | Read and write mode — reading and writing allowed | Yes | Start of the file |
| 'w+' | Write and read mode — overwrites or creates new file | No | Start of the file |

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

### ****Opening Files using**** open()

The open() function is used to open a file and returns a **file object**.  
Syntax:

file\_object = open(filename, mode)

**Closing files using close().**

### ****Closing Files using**** close()

After you're done working with a file, **always close it** to free system resources:

file1.close()

**Lab:**

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

# Open the file in write mode

file = open("sample.txt", "w")

# Write text to the file

file.write("Hello, this is a sample text.\nWelcome to file handling in Python!")

# Close the file

file.close()

print("Text written to 'sample.txt' successfully.")

**Practical Example:**

1. **Write a Python program to create a file and write a string into it.**

# Open a file in write mode ('w') — creates file if it doesn't exist

file = open("myfile.txt", "w")

# Write a string into the file

file.write("This is a newly created file.\nWriting a string into it.")

# Close the file

file.close()

print("File 'myfile.txt' created and string written successfully.")

1. **. Reading and Writing Files**

# Writing to a file

with open("myfile.txt", "w") as file:

file.write("Hello! This is a test file.\nLet's read and write files in Python.")

# Reading from the same file

with open("myfile.txt", "r") as file:

content = file.read()

print("File Content:")

print(content)

**Theory:**

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

## ****Reading from a File****

Assuming you have a file called example.txt with some text content.

### 1. read()

Reads the **entire content** of the file as a single string.

with open("example.txt", "r") as file:

content = file.read()

print(content)

### 2. readline()

Reads **one line at a time** (including the newline character \n).

with open("example.txt", "r") as file:

line1 = file.readline()

print(line1)

line2 = file.readline()

print(line2)

### 3. readlines()

Reads **all lines** and returns a **list of strings**, each string is a line.

with open("example.txt", "r") as file:

lines = file.readlines()

print(lines) # prints list of lines

for line in lines:

print(line.strip()) # print lines without extra newline

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

## ****Writing to a File****

### 1. write()

Writes a **single string** to the file.

with open("output.txt", "w") as file:

file.write("Hello, this is a single line.\n")

file.write("Writing another line.")

### 2. writelines()

Writes a **list of strings** to the file (does not add newlines automatically).

lines = ["First line.\n", "Second line.\n", "Third line.\n"]

with open("output.txt", "w") as file:

file.writelines(lines)

**Lab:**

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

# Open the file in read mode and print its contents

with open("sample.txt", "r") as file:

content = file.read()

print("File contents:\n")

print(content)

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

# List of strings to write

lines = [

"First line of text.\n",

"Second line of text.\n",

"Third line of text.\n"

]

# Open the file in write mode and write multiple lines

with open("output.txt", "w") as file:

file.writelines(lines)

print("Multiple lines written to 'output.txt' successfully.")

**Practical Examples:**

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

# Open (or create) a file in write mode

with open("file4.txt", "w") as file:

# Write a string into the file

file.write("This is a sample string written into the file.")

print("String written to 'file4.txt' successfully.")

1. **Write a Python program to read a file and print the data on the console.**

# Open the file in read mode

with open("file4.txt", "r") as file:

content = file.read()

print("File content:")

print(content)

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

with open("file4.txt", "r") as file:

# Read first 10 characters

text = file.read(10)

print("First 10 characters:", text)

# Get current file cursor position

position = file.tell()

print("Current file cursor position:", position)

**5. Exception Handling**

**Theory:**

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

## ****Introduction to Exceptions****

**Exception**: An error that occurs during program execution that disrupts the normal flow.

Examples: Division by zero, file not found, invalid input, etc.

If not handled, exceptions cause the program to crash.

## ****Handling Exceptions:**** try****,**** except****, and**** finally

### Basic structure:

try:

# Code that might raise an exception

risky\_operation()except SomeException:

# Code to handle the exception

handle\_error()finally:

# Code that runs no matter what (optional)

cleanup()

try **block**: Code that may cause an error.

except **block**: Code to handle specific errors if they occur.

finally **block**: (Optional) Code that runs always, even if an error occurs or not (e.g., closing files).

**Understanding multiple exceptions and custom exceptions.**

## ****Handling Multiple Exceptions****

You can catch multiple exceptions in one except block by grouping them in parentheses:

try:

# risky codeexcept (TypeError, ValueError):

# handle both TypeError and ValueError here

Or use separate except blocks for different exception types (as above).

## ⚙️ ****Custom Exceptions****

You can create your own exception types by **subclassing** the built-in Exception class.

**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: {num1} / {num2} = {result}")

except ZeroDivisionError:

print("Error: Cannot divide by zero!")

except ValueError:

print("Error: Invalid input! Please enter numeric values.")

finally:

print("Calculation attempt completed.**")**

**Write a Python program to demonstrate handling multiple exceptions.**

try:

# Convert input to int

num = int(input("Enter an integer: "))

# Division operation that may raise ZeroDivisionError

result = 10 / num

# Accessing an invalid index in a list (IndexError)

my\_list = [1, 2, 3]

print(my\_list[num])

except (ZeroDivisionError, ValueError, IndexError) as e:

print(f"An error occurred: {e}")

**Practical Examples:**

1. **Write a Python program to handle exceptions in a calculator.**

def calculator():

try:

num1 = float(input("Enter first number: "))

num2 = float(input("Enter second number: "))

op = input("Enter operation (+, -, \*, /): ")

if op == '+':

result = num1 + num2

elif op == '-':

result = num1 - num2

elif op == '\*':

result = num1 \* num2

elif op == '/':

result = num1 / num2

else:

print("Invalid operation!")

return

print(f"Result: {num1} {op} {num2} = {result}")

except ZeroDivisionError:

print("Error: Cannot divide by zero!")

except ValueError:

print("Error: Invalid input! Please enter numeric values.")

finally:

print("Calculation attempt finished.")

calculator()

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

try:

filename = input("Enter filename to read: ")

with open(filename, "r") as file:

content = file.read()

print(content)

num = int(input("Enter a number to divide 100 by: "))

result = 100 / num

print(f"Result: {result}")

except FileNotFoundError:

print("Error: The file was not found.")

except ZeroDivisionError:

print("Error: Division by zero is not allowed.")

except ValueError:

print("Error: Invalid number entered.")

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

try:

file = open("testfile.txt", "r")

content = file.read()

print(content)

except FileNotFoundError:

print("Error: File not found.")

finally:

# Ensure the file is closed even if an exception occurs

try:

file.close()

print("File closed successfully.")

except NameError:

print("File was never opened, so no need to close.")

1. **Write a Python program to print custom exceptions.**

class NegativeNumberError(Exception):

def \_\_init\_\_(self, message="Negative numbers are not allowed"):

self.message = message

super().\_\_init\_\_(self.message)

def check\_positive(num):

if num < 0:

raise NegativeNumberError

else:

print(f"{num} is positive.")

try:

number = int(input("Enter a positive number: "))

check\_positive(number)

except NegativeNumberError as e:

print(f"Custom Exception Caught: {e}")

except ValueError:

print("Invalid input! Please enter an integer.")

**6. Class and Object (OOP Concepts)**

**Theory:**

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

## ****Classes and Objects****

### ****Class:****

A **class** is a blueprint or template for creating objects.

It defines **attributes** (data) and **methods** (functions) that the objects created from the class will have.

Syntax example:

class Car:

pass

### ****Object:****

An **object** is an instance of a class.

It represents a specific entity created using the class blueprint.

Example:

my\_car = Car()

## ****Attributes and Methods****

### ****Attributes:****

Attributes are variables that belong to an object.

They store the state or properties of the object.

Example:

class Car:

def \_\_init\_\_(self, brand, color):

self.brand = brand # attribute

self.color = color # attribute

### ****Methods:****

Methods are functions defined inside a class.

They define the behavior/actions of an object.

Example:

class Car:

def start\_engine(self):

print(f"The {self.brand} engine started.")

**Difference between local and global variables.**

### ****Local Variable:****

Defined **inside a function or method**.

Accessible **only within** that function/method.

Created when the function runs and destroyed when it finishes.

### ****Global Variable:****

Defined **outside any function or class**.

Accessible **anywhere in the program** (unless shadowed by a local variable).

Use the global keyword inside functions if you want to modify a global variable.

**Lab:**

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

# Define a class named Car

class Car:

def \_\_init\_\_(self, brand, color, year):

self.brand = brand # attribute

self.color = color # attribute

self.year = year # attribute

# Create an object of the Car class

my\_car = Car("Toyota", "Red", 2020)

# Access and print the object's attributes

print("Car Brand:", my\_car.brand)

print("Car Color:", my\_car.color)

print("Car Year:", my\_car.year)

**Practical Examples:**

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

class Person:

def \_\_init\_\_(self, name, age):

self.name = name # attribute

self.age = age # attribute

# Create an object of Person

person1 = Person("Arjun", 23)

# Access properties using the object

print("Name:", person1.name)

print("Age:", person1.age)

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

count = 0 # Global variable

class Counter:

def \_\_init\_\_(self):

self.local\_count = 0 # Instance attribute (local to object)

def increment(self):

global count

count += 1 # Modifies global variable

self.local\_count += 1 # Modifies instance attribute

def display(self):

print("Global count:", count)

print("Local count:", self.local\_count)

# Create two objects

c1 = Counter()

c2 = Counter()

c1.increment()

c1.increment()

c1.display()

c2.increment()

c2.display()

print("Access global count directly:", count)

7. Inheritance

**Theory:**

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

### ****Types of Inheritance:****

**Single Inheritance:**

A child class inherits from a single parent class.

The child class can use or override the parent’s attributes and methods.

**Multilevel Inheritance:**

Involves a chain of inheritance where a class inherits from a child class, which in turn inherits from another class.

Example: Class C inherits from Class B, which inherits from Class A.

**Multiple Inheritance:**

A class inherits from more than one parent class.

This allows combining functionalities from multiple classes.

**Hierarchical Inheritance:**

Multiple child classes inherit from a single parent class.

Useful when different classes share common features of the parent.

**Hybrid Inheritance:**

Combination of two or more types of inheritance.

It’s a mix of single, multiple, and multilevel inheritance.

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

### ****The**** super() ****Function:****

super() is a built-in Python function that returns a temporary object of the superclass, allowing you to call its methods.

Commonly used in child classes to:

Call the parent class’s constructor (\_\_init\_\_) to initialize inherited attributes.

Invoke or extend parent class methods when overridden in the child class.

Using super() helps avoid explicitly naming the parent class, making code easier to maintain and supports multiple inheritance better.

**Lab:**

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

### ****1) Single Inheritance****

python

CopyEdit

class Animal:

def sound(self):

print("Animals make different sounds")

class Dog(Animal):

def bark(self):

print("Dog says Woof!")

# Create object

d = Dog()

d.sound()

d.bark()

### ****2) Multilevel Inheritance****

class Animal:

def sound(self):

print("Animals make sounds")

class Dog(Animal):

def bark(self):

print("Dog barks")

class Puppy(Dog):

def weep(self):

print("Puppy weeps")

# Create object

p = Puppy()

p.sound()

p.bark()

p.weep()

### ****3) Multiple Inheritance****

python

CopyEdit

class Father:

def skills(self):

print("Father: Programming")

class Mother:

def skills(self):

print("Mother: Designing")

class Child(Father, Mother):

def own\_skills(self):

print("Child: Gaming")

# Create object

c = Child()

c.skills() # Shows Father's method due to MRO

c.own\_skills()

### ****4) Hierarchical Inheritance****

class Animal:

def sound(self):

print("Animals make sounds")

class Dog(Animal):

def bark(self):

print("Dog barks")

class Cat(Animal):

def meow(self):

print("Cat meows")

# Create objects

d = Dog()

c = Cat()

d.sound()

d.bark()

c.sound()

c.meow()

### ****5) Hybrid Inheritance****

class Vehicle:

def vehicle\_info(self):

print("This is a vehicle")

class Car(Vehicle):

def car\_info(self):

print("This is a car")

class Electric:

def electric\_info(self):

print("Runs on electricity")

class Tesla(Car, Electric): # Hybrid: Car inherits Vehicle, Tesla inherits Car and Electric

def tesla\_info(self):

print("This is a Tesla car")

# Create object

t = Tesla()

t.vehicle\_info()

t.car\_info()

t.electric\_info()

t.tesla\_info()

**8. Method Overloading and Overriding**

**Theory:**

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

## ****1. Method Overloading:****

Method Overloading means defining **multiple methods with the same name** but **different parameters**.

Python **does not support method overloading natively** like Java or C++.

Instead, you can use **default arguments**, \*args, or keyword arguments (\*\*kwargs) to mimic overloading.

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

## ****2. Method Overriding:****

### Method Overriding means ****redefining a method**** from the parent class ****in the child class****.

The child class method **overrides** the parent’s implementation.

This allows **custom behavior** for inherited methods.

**Lab:**

**Write Python programs to demonstrate method overloading and method overriding.**

class Calculator:

def multiply(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

elif a is not None:

return a

else:

return 0

# Create object

calc = Calculator()

# Test cases

print("Multiply with 1 argument:", calc.multiply(5)) # Output: 5

print("Multiply with 2 arguments:", calc.multiply(5, 10)) # Output: 50

print("Multiply with 3 arguments:", calc.multiply(5, 10, 2)) # Output: 100

print("Multiply with no arguments:", calc.multiply()) # Output: 0

**Practical Examples:**

1. **Write a Python program to show method overloading.**

class Calculator:

def multiply(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

elif a is not None:

return a

else:

return 0

# Create object

calc = Calculator()

# Test cases

print("Multiply with 1 argument:", calc.multiply(5)) # Output: 5

print("Multiply with 2 arguments:", calc.multiply(5, 10)) # Output: 50

print("Multiply with 3 arguments:", calc.multiply(5, 10, 2)) # Output: 100

print("Multiply with no arguments:", calc.multiply()) # Output: 0

**20)Write a Python program to show method overriding.**

**class Animal:**

def speak(self):

print("The animal makes a sound")

class Dog(Animal):

def speak(self): # Overriding the parent method

print("The dog barks")

# Create objects

animal = Animal()

dog = Dog()

# Call methods

animal.speak() # Output: The animal makes a sound

dog.speak() # Output: The dog barks

**9. SQLite3 and PyMySQL (Database Connectors)**

**Theory:**

Introduction to SQLite3 and PyMySQL for database connectivity.

Creating and executing SQL queries from Python using these connectors.

**Lab:**

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

**Practical Examples:**

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

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

**10. Search and Match Functions**

**Theory:**

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

matching.

Difference between search and match.

**Lab:**

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

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

**Practical Examples:**

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

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