Assignment-1

❖Printing on Screen

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

➢ The print() function in Python is a built-in function used to display information to the standard output (usually the screen). It serves as a primary means of communication between a program and the user.

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

f-Strings (Formatted String Literals)

f-strings were introduced in Python 3.6 and provide a concise and efficient way to format strings. They allow expressions and variables to be written directly inside string literals using curly braces {}. At runtime, Python evaluates these expressions and inserts their values into the string. f-strings improve code readability and reduce complexity compared to older formatting techniques.

format() Method

The format() method formats strings by replacing placeholders within a string with specified values. Placeholders are defined using curly braces {} and can be filled using positional or named arguments. This method separates the string template from the data, which can be useful for more complex or reusable formatting scenarios.

❖Reading Data from Keyboard

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

The input() function in Python is used to take input from the user through the keyboard. When executed, it waits for the user to enter data. The entered value is always returned as a string. It can display a message to guide the user. The input() function makes programs interactive. It is widely used in beginner-level Python programs.

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

In Python, user input is taken using the input() function, which always returns a string value. To convert this input into other data types, type casting functions are used. The int() function converts input into an integer, while float() converts it into a decimal value. The str() function is used to convert data into string format. Type conversion is necessary for performing mathematical operations. It helps ensure correct and error-free program execution.

Example:

```
a = input("Enter a number: ")
b = int(a)
c = float(a)
```

output:

```
Integer value: 10

Float value: 10.0
```

❖ Opening and Closing Files

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

Python files can be opened in different modes depending on the operation required. The 'r' mode is used for reading files. The 'w' mode writes data and creates a new file if it does not exist. The 'a' mode appends data to an existing file. The 'r+' and 'w+' modes allow both reading and writing. Choosing the correct mode is important for file safety.

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

The open() function is used to create or access files in Python. It takes two main arguments: file name and mode. If the file does not exist, it is created in write mode. The function returns a file object used to perform operations. Proper mode selection ensures correct file handling.

Example:

```
file = open("data.txt", "w")
```

3. Closing files using close().

The close() function is used to close an opened file. It releases system resources associated with the file. Closing a file ensures data is properly saved. It also prevents data corruption. It is considered good programming practice.

Example:

```
file.close()
```

❖ Reading and writing Files

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

Python provides different methods to read file data. The read() method reads the entire file content. The readline() method reads one line at a time. The readlines() method reads all lines and returns them as a list. These methods help manage file data efficiently.

Example:

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

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

The write() method is used to write a single string to a file. The writelines() method writes multiple strings at once. These methods work only in write or append modes. Writing data allows permanent storage. Proper file handling is important to avoid data loss.

Example:

```
file = open("data.txt", "w")

file.write("Hello World")
```

❖ Exception Handling

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

Exceptions are runtime errors that disrupt program execution. Python uses try, except, and finally blocks to handle exceptions. The try block contains risky code. The except block handles errors. The finally block always executes. Exception handling improves program reliability.

Example:

```
try:
   x = int(input())
except ValueError:
   print("Invalid input")
finally:
   print("Done")
```

2. Understanding multiple exceptions and custom exceptions

Multiple exceptions allow handling different errors separately. Python supports multiple except blocks. Custom exceptions are user-defined error classes. They improve clarity and control over errors. Custom exceptions are created using classes.

Example:

```
class MyError(Exception):
   pass

try:
   x = int(input("Enter number: "))
   if x < 0:
      raise MyError
except ValueError:
   print("Value Error")
except MyError:
   print("Custom Error: Negative number")
```

Output (input = -5):

```
Custom Error: Negative number
```

❖ Class and Object(OOP Concepts)

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

A class is a blueprint used to create objects. An object is an instance of a class. Attributes are variables that store data inside a class. Methods are functions defined inside a class that describe object behavior. Classes help organize code logically. This concept is part of object-oriented programming.

Example:

```
class Student:
   name = "Niya"
   def show(self):
      print(self.name)

s = Student()
s.show()
```

output:

```
Niya
```

2. Difference between local and global variables.

Local variables are declared inside a function and can be accessed only within that function. Global variables are declared outside functions and can be accessed anywhere in the program. Local variables provide better data security. Global variables remain in memory throughout program execution. Using global variables excessively is not recommended. Proper scope management improves code quality.

❖ Inheritance

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

Inheritance is a key concept of object-oriented programming that allows one class to use the properties and methods of another class. Single inheritance involves one parent and one child class. Multilevel inheritance forms a chain where a class is derived from another derived class. Multiple inheritance allows a class to inherit from more than one parent class. Hierarchical inheritance occurs when multiple child classes inherit from the same parent class. Hybrid inheritance is a combination of more than one type of inheritance. Inheritance helps in code reusability and better program structure.

1. Single Inheritance

```
class Parent:
   def show(self):
      print("This is Parent class")

class Child(Parent):
   def display(self):
      print("This is Child class")

obj = Child()
obj.show()
obj.display()
```

output:

```
This is Parent class
This is Child class
```

2. Multilevel Inheritance

```
class A:
   def showA(self):
      print("Class A")

class B(A):
   def showB(self):
      print("Class B")

class C(B):
   def showC(self):
      print("Class C")

obj = C()
obj.showA()
obj.showB()
obj.showC()
```

output:

```
Class A
Class B
Class C
```

3. Multiple Inheritance

```
class A:
    def showA(self):
        print("Class A")

class B:
    def showB(self):
        print("Class B")

class C(A, B):
    def showC(self):
        print("Class C")

obj = C()
obj.showA()
obj.showB()
obj.showC()
```

output:

```
Class A
Class B
Class C
```

4. Hierarchical Inheritance

```
class Parent:
   def show(self):
      print("Parent class")

class Child1(Parent):
   def display1(self):
      print("Child class 1")

class Child2(Parent):
   def display2(self):
      print("Child class 2")

obj1 = Child1()
obj2 = Child2()
obj1.show()
obj1.display1()
obj2.show()
obj2.display2()
```

output:

```
Parent class
Child class 1
Parent class
Child class 2
```

5. Hybrid Inheritance

```
class A:
   def showA(self):
      print("Class A")

class B(A):
   def showB(self):
      print("Class B")

class C(A):
   def showC(self):
      print("Class C")

class D(B, C):
   def showD(self):
      print("Class D")

obj = D()
obj.showA()
obj.showB()
obj.showC()
obj.showD()
```

output:

```
Class A
Class B
Class C
Class D
```

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

The super() function is used to access methods and attributes of a parent class.
It helps avoid explicitly naming the parent class.
This makes the code cleaner and easier to maintain.
It is mainly used in inheritance scenarios.
super() allows reuse of parent class functionality in child classes.
It supports method overriding efficiently.
Example:

```
class Parent:
   def show(self):
      print("This is parent class")

class Child(Parent):
   def show(self):
      super().show()
      print("This is child class")

obj = Child()
obj.show()
```

output:

```
This is parent class
This is child class
```

❖ Method Overloading and Overriding

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

Method overriding occurs when a child class redefines a method that already exists in its parent class. The method name and parameters in the child class must be the same as those in the parent class. When the overridden method is called using a child class object, the child class version is executed instead of the parent class version. This concept supports runtime polymorphism in Python. Method overriding allows a child class to modify or extend the behavior of the parent class. It is widely used in object-oriented programming.

Example:

```
class Parent:

   def display(self):

      print("This is Parent class")

class Child(Parent):

   def display(self):

      print("This is Child class")



obj = Child()

obj.display()
```

output:

```
This is Child class
```

❖ SQLite3 and PyMySQL (Database Connectors)

1. Introduction to SQLite3 and PyMySQL for database connectivity.

SQLite3 and PyMySQL are Python libraries used to connect Python programs with databases. SQLite3 is a lightweight, file-based database system that does not require a separate server. It is mainly used for small applications and local data storage. PyMySQL is a Python connector used to connect with MySQL databases. It works in a client-server environment and is suitable for large applications. Both libraries allow Python programs to execute SQL queries. Database connectivity helps in storing, retrieving, and managing data efficiently.

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

Python uses database connectors like SQLite3 and PyMySQL to execute SQL queries. A database connection is first established, and then a cursor object is created. The cursor is used to execute SQL commands such as CREATE, INSERT, UPDATE, and SELECT. After executing write operations, changes are saved using the commit() method. The fetchall() or fetchone() methods are used to retrieve data. This process allows Python programs to interact dynamically with databases.

Example Code (SQLite3):

```
import sqlite3


conn = sqlite3.connect("test.db")

cur = conn.cursor()

cur.execute("CREATE TABLE IF NOT EXISTS student(id
INTEGER, name TEXT)")

cur.execute("INSERT INTO student VALUES (1, 'Niyati')")

conn.commit()

cur.execute("SELECT * FROM student")

print(cur.fetchall())

conn.close()
```

output:

```
[(1, 'Niyati')]
```

❖ Search and Match Functions

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

The re module in Python is used for pattern matching using regular expressions. The re.search() function scans the entire string to find a matching pattern. If the pattern is found anywhere in the string, it returns a match object. The re.match() function checks for a match only at the beginning of the string. Both functions return None if no match is found. They are commonly used for data validation and text processing.

Example Code:

```
import re

text = "Python is easy"

result1 = re.search("easy", text)
result2 = re.match("Python", text)

print(result1)
print(result2)
```

output:

```
<re.Match object; span=(10, 14), match='easy'>
<re.Match object; span=(0, 6), match='Python'>
```

2. Difference between search and match

re.search() and re.match() are functions used for pattern matching in Python’s re module. The re.match() function checks for a pattern only at the beginning of the string. If the pattern is not found at the start, it returns None. The re.search() function scans the entire string to find the pattern. Therefore, search() is more flexible than match(). Both functions return a match object if the pattern is found.