Module 15) Advance Python Programming

1. Printing on screen

- **Introduction to the Print( ) function**
  The print() function in Python is used to display output on the screen. It is one of the most commonly used functions, especially for debugging and interacting with users.
  Syntax :
  print(value, ……, sep = ‘ ‘,end = ‘\n’,file = sys.stdout, flush=False)

  where,
  value : The data or variables to be printed.
  sep : The separator between multiple values (default is a space).
  end : The string added at the end of the output (default is a newline \n).
  file : The output stream (default is sys.stdout, which prints to the console).
  flush : If True, the output buffer is forcibly flushed.

- **Formatting outputs using f-strings and format**
  When printing output in Python, formatting strings properly helps make the output readable and professional.
  There are two common ways to format strings:
  1.Using f-string (formatted string literals)
  f-strings allow you to embed expressions inside string literals using curly braces {}.

  Ex:
  name = "Alice"
  age = 25
  print(f"My name is {name} and I am {age} years old.")

  O/P:
  My name is Alice and I am 25 years old.

  2.The .format( ) method
  The .format() method was widely used before f-strings were introduced.

  Ex:
  name = "Bob"
  age = 30
  print("My name is {} and I am {} years old.".format(name, age))

  O/P:
  My name is Bob and I am 30 years old.

2. Reading Data from keyboard

- **Using the input() function to read user input from the keyboard.**
  The input() function allows users to enter data from the keyboard during program execution. The input is always read as a string by default, but we can convert it into other data types as needed.
  The input() function always returns data as a **string**.

  Ex:
  ```
  name = input("Enter your name: ")
  print(f"Hello, {name}!")
  ```

  O/P:
  ```
  Enter your name: Alice
  Hello, Alice!
  ```

- **Converting user input into different data types (e.g., int, float, etc.).**
  Ex : converting into an integer
  ```
  age = int(input("Enter your age: "))
  print(f"You are {age} years old.")
  ```

  O/P :
  ```
  Enter your age: 25
  You are 25 years old.
  ```

  Note : If the user enters something that is **not a number**, it will cause an error.

  Ex : converting into a float
  ```
  price = float(input("Enter the price of the product: "))
  print(f"The price is ${price:.2f}")
  ```

  O/P :
  ```
  Enter the price of the product: 19.99
  The price is $19.99
  ```

  Ex : converting into a Boolean
  ```
  is_hungry = bool(int(input("Are you hungry? (1 for Yes, 0 for No): ")))
  print(f"Hungry: {is_hungry}")
  ```

  O/P :
  ```
  Are you hungry? (1 for Yes, 0 for No): 1
  Hungry: True
  ```

Note : Here, we first convert the input to an **integer**, then to a **boolean** (0 is False, any other number is True).

3. Opening and Closing files

- **Opening files in different modes ('r', 'w', 'a', 'r+', 'w+').**
  In Python, the open() function is used to work with files. It allows us to **read**, **write**, **append**, or **modify** files in different modes.
  syntax :
  file = open("filename.txt" , mode)

  where,
  filename.txt = Name of the file
  mode = specifies how the file should be opened

Mode	Description
`'r'`	Read mode (default). Opens the file for reading. Fails if the file doesn't exist.
`'w'`	Write mode. Creates a new file or **overwrites** an existing file.
`'a'`	Append mode. Adds data to the end of a file without deleting existing content.
`'r+'`	Read and write mode. Reads and updates the file. Fails if the file doesn't exist.
`'w+'`	Write and read mode. Creates a new file or overwrites an existing file.
`'a+'`	Append and read mode. Opens a file for both reading and appending.

- **Using the open() function to create and access files.**
  The open() function in Python is used to create, open, and access files. It allows us to read, write, or modify file contents.
  Creating and Accessing a file :
  The `open()` function takes two main arguments:
  file = open("filename.txt",mode)

  filename.txt = The name of the file.
  mode =The mode in which the file is opened ('r', 'w', 'a', etc.).

  Creating a file("w" or "a" mode)
  if file does not exist,it will be created.

  ```
  file = open("myfile.txt", "w")  # Creates a new file or overwrites if it exists
  file.write("Hello, this is a new file!\n")  # Write content to the file
  file.close()  # Close the file
  ```

  Opening an existing file("r" mode)
  ```
  file = open("myfile.txt", "r")  # Open file in read mode
  ```

```
content = file.read()  # Read file contents
print(content)
file.close()  # Close the file
```

- **Closing files using close().**
  After opening a file, it is important to **close it** using file.close().
  This releases system resources and ensures that the file is properly saved.
  If a file is not closed properly, changes may not be saved correctly.

  Ex:
```
file = open("example.txt", "w")
file.write("This is some text.")
file.close()  # Close the file after writing
```

  We can check if a file is closed using file.closed.
  Ex :
```
file = open("example.txt", "r")
print(file.closed)  # Output: False (file is open)
file.close()
print(file.closed)  # Output: True (file is closed)
```

4. Reading and Writing files

- **Reading from a file using read(), readline(), readlines().**
  Python provides several methods to **read** and **write** files using the open() function.
  To read data from a file, Python provides three main methods:
    - read( )
    - readline( )
    - readlines( )

  Using read( ) (Reads the whole file)
  Reads the entire content of the file as a **single string**.
  Can accept an optional argument (read(n)) to read n characters.

  Ex :
```
with open("example.txt", "r") as file:
   content = file.read()  # Reads the entire file
   print(content)
```

  Ex : Reading specific number of character
```
with open("example.txt", "r") as file:
   print(file.read(10))  # Reads the first 10 characters
```

  Using readline( ) (Reads one line at a time)
  Reads a **single line** from the file.

Each time you call readline(), it reads the **next** line.

Ex :

```
with open("example.txt", "r") as file:
    print(file.readline())  # Reads the first line
    print(file.readline())  # Reads the second line
```

Advantage : Useful for reading large files **line by line** instead of loading the entire file into memory.

Using readlines( ) (Reads all line into a List)
Reads **all** lines and returns them as a **list of strings**.

Ex :

```
with open("example.txt", "r") as file:
    lines = file.readlines()  # Returns a list of lines
    print(lines)
```

O/P :

```
['Line 1\n', 'Line 2\n', 'Line 3\n']
```

note : Notice that **\n (newline character)** is included at the end of each line.

- **Writing to a file using write() and writelines().**
  Python provides two main methods to write data into a file:
  - write( )
  - writelines( )

Using write( ) (writes a string to the file)
Writes **a single string** to the file.
If the file **already exists**, it **overwrites** its content.

Ex :

```
with open("output.txt", "w") as file:
    file.write("Hello, World!\n")
    file.write("Writing to a file in Python.\n")
```

note : If "output.txt" does not exist, Python will **create** it.

Using writelines( ) (writes a list of strings)
Writes **multiple lines** to the file from a **list of strings.**

Ex :

```
lines = ["Line 1\n", "Line 2\n", "Line 3\n"]
```

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

        file.writelines(lines)  # Writes all lines at once
```

Advantage : writelines() is faster when writing **multiple lines** compared to calling write() multiple times.

Appending to a file ('a' mode)
If you **don't** want to overwrite the file but **add new content**, use 'a' (append mode). This **preserves** the existing content and adds new data **at the end**.

Ex :

```
with open("output.txt", "a") as file:
   file.write("This is an additional line.\n"
```

5. Exception Handling

- **Introduction to exceptions and how to handle them using try, except, and finally.**
  In Python, **exceptions** are errors that occur during the execution of a program. Instead of crashing the program, we can **handle** these exceptions using try, except, and finally.
  An exception occurs when Python encounters an error during execution. Common exceptions include:

Exception	Description
`ZeroDivisionError`	Raised when dividing by zero.
`ValueError`	Raised when an invalid value is provided.
`TypeError`	Raised when an operation is performed on an incompatible type.
`FileNotFoundError`	Raised when trying to open a file that does not exist.
`IndexError`	Raised when accessing an invalid index in a list.

Ex : unhandled exception
```
print(10 / 0)  # ZeroDivisionError
```

O/P :
ZeroDivisionError: division by zero

Handling Exception with try and except
try → Contains code that may cause an exception.
except → Handles the exception and prevents program crashes

Ex :

```
try:
    result = 10 / 0  # This will cause a ZeroDivisionError
except ZeroDivisionError:
    print("Error: Cannot divide by zero!")
```

O/P :
Error: Cannot divide by zero!.

Handling Multiple exception:
You can handle **different exceptions separately**:

Ex :

```
try:
    num = int(input("Enter a number: "))  # May raise ValueError
    result = 10 / num  # May raise ZeroDivisionError
except ValueError:
    print("Error: Please enter a valid number!")
except ZeroDivisionError:
    print("Error: Cannot divide by zero!")
```

Ex : catching multiple exception in one except block

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except (ValueError, ZeroDivisionError):
    print("Error: Invalid input or division by zero!")
```

Using finally (Always Executes)
The finally block **always runs**, whether an exception occurs or not.
Useful for **clean-up actions**, like closing files or releasing resources.

Ex :

```
try:
    file = open("data.txt", "r")  # Try to open a file
    content = file.read()
except FileNotFoundError:
    print("Error: File not found!")
finally:
    print("Closing resources...")  # Runs always
```

O/P :
Error: File not found!
Closing resources...

- **Understanding multiple exceptions and custom exceptions.**
  Python allows handling **multiple exceptions** and also provides the ability to **define custom exceptions** to handle specific errors more effectively.

  Handling Multiple exception
  Sometimes, different types of exceptions may occur in a program. Python allows handling them using **multiple `except` blocks** or **a single block** handling multiple exceptions.

  Ex : handling different exceptions seperately

```
try:
    num = int(input("Enter a number: "))  # May raise ValueError
    result = 10 / num  # May raise ZeroDivisionError
except ValueError:
    print("Error: Please enter a valid number!")
except ZeroDivisionError:
    print("Error: Cannot divide by zero!")
except Exception as e:  # Catch-all for any other exceptions
    print(f"An unexpected error occurred: {e}")
```

  O/P :
  Enter a number: 0
  Error: Cannot divide by zero!

  Ex : handling multiple exceptions in one except block

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except (ValueError, ZeroDivisionError):
    print("Error: Invalid input or division by zero!")
```

  Using except Exception as e to catch any exception
  If you don’t know which error might occur, you can use:

  Ex :

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except Exception as e:
    print(f"An error occurred: {e}")
```

note : This **catches all exceptions** but should be used carefully to avoid hiding important errors.

Creating Custom exception

Python allows creating **user-defined exceptions** by subclassing the Exception class. Using custom exceptions makes error handling more meaningful and specific.

Ex : Defining and raising a custom exception

```
class NegativeNumberError(Exception):
    """Custom exception for negative numbers."""
    pass  # No additional code needed

try:
    num = int(input("Enter a positive number: "))
    if num < 0:
        raise NegativeNumberError("Negative numbers are not allowed!")
    print(f"You entered: {num}")
except NegativeNumberError as e:
    print("Custom Exception:", e)
```

O/P :

```
Enter a positive number: -5
Custom Exception: Negative numbers are not allowed!
```

Ex : Custom Exception with __init__() for Extra Information

```
class AgeError(Exception):
    def __init__(self, age, message="Age cannot be negative or zero!"):
        self.age = age
        self.message = message
        super().__init__(self.message)

try:
    age = int(input("Enter your age: "))
    if age <= 0:
        raise AgeError(age)
    print(f"Your age is {age}")
except AgeError as e:
    print(f"Custom Exception: {e.message} (Entered: {e.age})")
```

O/P :

```
Enter your age: -3
Custom Exception: Age cannot be negative or zero! (Entered: -3)
```

6. Class and Object (OOP Concept)

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

  Python is an **object-oriented programming (OOP)** language, which means it allows us to create and use **classes and objects** to model real-world concepts.

  A **class** is a **blueprint** for creating objects. It defines:
  - **Attributes (variables)** → Represent the properties of an object.
  - **Methods (functions)** → Define the behavior of the object.

  Ex : defining a class

```
class Car:
    """A simple Car class"""
    def __init__(self, brand, model, year):
        self.brand = brand          # Attribute
        self.model = model          # Attribute
        self.year = year            # Attribute

    def display_info(self):         # Method
        print(f"{self.year} {self.brand} {self.model}")
```

  An **object** is an **instance of a class**. It is created from a class and has its own values for the defined attributes.

  Ex : creating an object

```
class Car:
    """A simple Car class"""
    def __init__(self, brand, model, year):
        self.brand = brand          # Attribute
        self.model = model          # Attribute
        self.year = year            # Attribute

    def display_info(self):         # Method
        print(f"{self.year} {self.brand} {self.model}")

my_car = Car("Toyota", "Camry", 2023)  # Creating an object
my_car.display_info()  # Calling the method
```

  O/P :
  2023 Toyota Camry

  Attributes store **data** about an object.
  Attributes can be **accessed and modified** using dot notation.

  Ex :

```
class Car:
    """A simple Car class"""
    def __init__(self, brand, model, year):
        self.brand = brand          # Attribute
        self.model = model          # Attribute
```

```
        self.year = year                # Attribute

    def display_info(self):         # Method
        print(f"{self.year} {self.brand} {self.model}")

my_car = Car("Toyota", "Camry", 2023)  # Creating an object
#my_car.display_info()  # Calling the method

print(my_car.brand)  # Accessing an attribute
my_car.year = 2024   # Modifying an attribute
print(my_car.year)
```

O/P :
Toyota
2024

Methods define **behaviors (actions)** that objects can perform.

Ex : defining method

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

    def greet(self):  # Method
        print(f"Hello, my name is {self.name} and I am {self.age} years old.")

# Creating an object
person1 = Person("Alice", 30)
person1.greet()
```

O/P :
Hello, my name is Alice and I am 30 years old.

The __init__( ) method (constructor)
The __init__() method is a **special method** that is called **automatically** when an object is created.

Ex :

```
class Dog:
    def __init__(self, name, breed):
        self.name = name
        self.breed = breed

    def bark(self):
        print(f"{self.name} is barking!")

# Creating an object
dog1 = Dog("Buddy", "Golden Retriever")
```

```
dog1.bark()
```
O/P :
Buddy is barking!

- **Difference between local and global variables**.
  In Python, **variables can have different scopes**, meaning they can be **local** to a function or **global** across the entire program.
  A **global variable** is declared **outside any function** and can be accessed **anywhere** in the program.
  Global variables can be used inside functions without redefining them.

  Ex :
```
x = 10  # Global variable

def display():
    print("Global x:", x)  # Accessing global variable inside a function

display()
print("Outside function, x:", x)  # Accessing global variable outside function
```

  O/P :
  Global x: 10
  Outside function, x: 10

  A **local variable** is declared **inside a function** and **only exists within that function**.
  Local variables CANNOT be accessed outside the function.

  Ex :
```
def display():
    y = 5  # Local variable
    print("Local y:", y)

display()
print("Outside function, y:", y)  # This will cause an error
```

  O/P :
  NameError: name 'y' is not defined

  If you want to **modify** a global variable inside a function, use the **global keyword**.
  Without global, Python **creates a local variable** instead of modifying the global one.
  Ex :

```
x = 10            # Global variable

def modify():
    global x      # Declare that we're modifying the global x
    x = 20        # Modifying global variable
    print("Inside function, x:", x)
```

```
modify()
print("Outside function, x:", x)          # The value of x has changed globally
```

O/P:
Inside function, x: 20
Outside function, x: 20

Using the nonlocal Keyword (For Nested Functions)
The nonlocal keyword allows modifying **a variable from an outer function** (but not a global variable).
nonlocal **modifies the variable** in the **enclosing function**.

Ex :

```
def outer():
    x = 10  # Local variable of outer function

    def inner():
        nonlocal x  # Modify outer function's variable
        x = 20
        print("Inside inner function, x:", x)

    inner()
    print("Inside outer function, x:", x)

outer()
```

O/P :
Inside inner function, x: 20
Inside outer function, x: 20

7. Inheritance

- **Single, Multilevel, Multiple, Hierarchical, and Hybrid inheritance in Python.**
  Inheritance is a key concept in Object-Oriented Programming (OOP) that allows a class to **inherit** properties and behaviors (attributes and methods) from another class.

  - **Single Inheritance**
    A child class inherits from a single parent class.

    Ex :

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

# Child class (inherits from Animal)
class Dog(Animal):
    def bark(self):
        print("Dog barks")

# Creating an object of Dog
```

```
dog = Dog()
dog.speak()      # Inherited method
dog.bark()       # Own method
```

O/P :
Animal makes a sound
Dog barks

- **Multilevel Inheritance**
  A class inherits from another class, which itself is inherited from another class (like a chain).

  Ex :
```
# Grandparent class
class Animal:
    def speak(self):
        print("Animal makes a sound")

# Parent class (inherits from Animal)
class Dog(Animal):
    def bark(self):
        print("Dog barks")

# Child class (inherits from Dog)
class Puppy(Dog):
    def weep(self):
        print("Puppy weeps")

# Creating an object of Puppy
puppy = Puppy()
puppy.speak()  # Inherited from Animal
puppy.bark()   # Inherited from Dog
puppy.weep()   # Own method
```

  O/P :
  Animal makes a sound
  Dog barks
  Puppy weeps

- **Multiple Inheritance**
  A child class inherits from more than one parent class.

  Ex :
```
# Parent class 1
class Father:
    def show_father(self):
        print("Father's property")

# Parent class 2
class Mother:
```

```
    def show_mother(self):
        print("Mother's property")

# Child class (inherits from both Father and Mother)
class Child(Father, Mother):
    def show_child(self):
        print("Child's property")

# Creating an object of Child
child = Child()
child.show_father()  # Inherited from Father
child.show_mother()  # Inherited from Mother
child.show_child()   # Own method
```

O/P :
Father's property
Mother's property
Child's property

- **Hierarchical inheritance**
  One parent class is inherited by multiple child classes.

Ex :

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

# Child class 1
class Dog(Animal):
    def bark(self):
        print("Dog barks")

# Child class 2
class Cat(Animal):
    def meow(self):
        print("Cat meows")

# Creating objects
dog = Dog()
cat = Cat()

dog.speak()  # Inherited from Animal
dog.bark()   # Own method

cat.speak()  # Inherited from Animal
cat.meow()   # Own method
```

O/P :
Animal makes a sound

Dog barks
Animal makes a sound
Cat meows

- **Hybrid Inheritance**
  A combination of two or more types of inheritance.

  Example: Multiple + Multilevel inheritance.

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

# Parent class 2
class Wild:
    def habitat(self):
        print("Wild animals live in forests")

# Child class (inherits from Animal)
class Dog(Animal):
    def bark(self):
        print("Dog barks")

# Grandchild class (inherits from Dog and Wild)
class Wolf(Dog, Wild):
    def howl(self):
        print("Wolf howls")

# Creating an object of Wolf
wolf = Wolf()
wolf.speak()  # Inherited from Animal
wolf.bark()   # Inherited from Dog
wolf.habitat()  # Inherited from Wild
wolf.howl()   # Own method
```

  O/P :
  Animal makes a sound
  Dog barks
  Wild animals live in forests
  Wolf howls

- **Using the super() function to access properties of the parent class.**
  The super() function in Python is used to call methods from a **parent class** inside a **child class**.
  This is useful when:
  The child class **inherits** from the parent class.
  You want to **reuse** the parent class's methods **without rewriting them**.
  You need to **extend** the functionality of the parent class.

- **Using super() to Call the Parent Class Constructor (__init__())**

  Ex :

```
# Parent class
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

# Child class using super()
class Student(Person):
    def __init__(self, name, age, student_id):
        super().__init__(name, age)          # Call Parent class __init__()
        self.student_id = student_id         # New attribute in child class

    def display(self):
        print(f"Name: {self.name}, Age: {self.age}, Student ID: {self.student_id}")

# Creating an object of Student
student1 = Student("Alice", 20, "S123")
student1.display()
```

  O/P :

```
Name: Alice, Age: 20, Student ID: S123
```

  - **Using super() to Call Parent Class Methods**

    Ex :

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

# Child class using super()
class Dog(Animal):
    def speak(self):
        super().speak()  # Call the parent class's speak() method
        print("Dog barks")

# Creating an object of Dog
dog = Dog()
dog.speak()
```

    O/P :

```
Animal makes a sound
Dog barks
```

  - **Using super() in Multilevel Inheritance**

    Ex :

```
# Grandparent class
class A:
    def show(self):
        print("Class A")

# Parent class
class B(A):
    def show(self):
        super().show()  # Call A's method
        print("Class B")

# Child class
class C(B):
    def show(self):
        super().show()  # Call B's method
        print("Class C")

# Creating an object of class C
obj = C()
obj.show()
```

O/P :
Class A
Class B
Class C

8. Method Overloading and Overriding

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

  Method Overloading means defining **multiple methods** with the **same name** but **different parameters** in a class.
  Python **does not support** method overloading **directly** like Java or C++.
  However, it can be **achieved using default arguments or variable-length arguments (*args, **kwargs)**.

  Using Default Arguments for Method Overloading
  Python allows **method overloading** by using **default values** for parameters.
  Python allows different function calls by handling missing parameters using default values.

  Ex :

```
class Calculator:
    def add(self, a, b=0, c=0):  # Default values allow different numbers of arguments
        return a + b + c

# Creating an object
calc = Calculator()
```

```
# Calling method with different numbers of arguments
print(calc.add(5))        # Calls add(a), uses default b=0, c=0
print(calc.add(5, 10))    # Calls add(a, b), uses default c=0
print(calc.add(5, 10, 15))  # Calls add(a, b, c)
```

O/P :
5
15
30

Using *args for Method Overloading
Python can handle **variable numbers of arguments** using *args.
Using *args, the method can accept an unlimited number of parameters.

Ex :
```
class Calculator:
    def add(self, *args):  # Accepts any number of arguments
        return sum(args)

# Creating an object
calc = Calculator()

# Calling method with different numbers of arguments
print(calc.add(5))
print(calc.add(5, 10))
print(calc.add(5, 10, 15))
print(calc.add(1, 2, 3, 4, 5))
```

O/P :
5
15
30
15

Using @staticmethod for Different Argument Types
Python does not allow **true method overloading** based on **data types**, but we can use @staticmethod to differentiate method behavior.
The method behaves differently based on input type.

Ex :
```
class Printer:
    @staticmethod
    def display(data):
```

```
        if isinstance(data, int):
            print("Integer:", data)
        elif isinstance(data, float):
            print("Float:", data)
        elif isinstance(data, str):
            print("String:", data)
        else:
            print("Unsupported data type")

# Calling the method with different data types
Printer.display(10)
Printer.display(3.14)
Printer.display("Hello")
```

O/P :
Integer: 10
Float: 3.14
String: Hello

- **Method overriding: redefining a parent class method in the child class**.
  Method Overriding occurs when a **child class** redefines a method that is already present in the **parent class**.
  This allows the child class to **modify** or **extend** the behavior of the inherited method.

  Ex :

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

# Child class overrides the speak() method
class Dog(Animal):
    def speak(self):
        print("Dog barks")

# Creating objects
a = Animal()
a.speak()  # Calls the parent class method

d = Dog()
d.speak()  # Calls the overridden method in Dog
```

  O/P :

Animal makes a sound
Dog barks

Using super()to call the parent class method
The super() function allows calling the **parent class method** inside the **child class**.

Ex :
```
# Parent class
class Animal:
    def speak(self):
        print("Animal makes a sound")

# Child class overrides the method but still calls the parent method
class Dog(Animal):
    def speak(self):
        super().speak()  # Calling the parent class method
        print("Dog barks")

# Creating an object
d = Dog()
d.speak()
```

O/P :
Animal makes a sound
Dog barks

Overriding __init__() constructor
The child class can **override the constructor (__init__())** to **add new attributes** while still calling the parent class constructor.

Ex :
```
# Parent class
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def display(self):
        print(f"Name: {self.name}, Age: {self.age}")

# Child class overrides __init__() to add new properties
class Student(Person):
    def __init__(self, name, age, student_id):
        super().__init__(name, age)  # Call parent constructor
```

```
        self.student_id = student_id

    def display(self):
        super().display()  # Call parent display method
        print(f"Student ID: {self.student_id}")

# Creating an object of Student
student = Student("Alice", 20, "S123")
student.display()
```

O/P :
Name: Alice, Age: 20
Student ID: S123

9. SQLite3 and PyMySQL (Database Connectors)

- **Introduction to SQLite3 and PyMySQL for database connectivity.**
  When working with databases in Python, two popular choices for database connectivity are **SQLite3** and **PyMySQL**. Both serve different use cases based on the type of application you are developing.

  - **SQLite3**
    - SQLite is a **lightweight, file-based database** that does not require a separate server.
    - It is built into Python (no extra installation needed).
    - Ideal for **small to medium** applications, testing, and local storage.

    To use SQLite in Python, you import the sqlite3 module and connect to a database file (or create one if it doesn't exist).

    Ex : Basic SqLite Operations

```
import sqlite3

# Connect to a database (creates a new one if it doesn't exist)
conn = sqlite3.connect("my_database.db")
cursor = conn.cursor()

# Create a table
cursor.execute("""
CREATE TABLE IF NOT EXISTS users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    age INTEGER
)
""")

# Insert data
cursor.execute("INSERT INTO users (name, age) VALUES (?, ?)", ("Alice", 25))
```

```
# Commit and close
conn.commit()
conn.close()
```

Limitation :

- **Not ideal for large scale application**
- **limited support for concurrent writes.**

- **PyMySQL**
  - PyMySQL is a **Python library** that allows interaction with a **MySQL database**.
  - Unlike SQLite, **MySQL requires a running database server**.
  - Suitable for **large-scale applications** that need multi-user access.

  Installing PyMySQL
  Since PyMySQL is not built into Python, you need to install it first:
  pip install pymysql

  Ex : connecting to MySQL and Performing Operations

```
import pymysql

# Connect to MySQL database
conn = pymysql.connect(
    host="localhost",
    user="root",
    password="yourpassword",
    database="test_db"
)
cursor = conn.cursor()

# Create a table
cursor.execute("""
CREATE TABLE IF NOT EXISTS employees (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    salary FLOAT
)
""")

# Insert data
cursor.execute("INSERT INTO employees (name, salary) VALUES (%s, %s)", ("John Doe", 50000))

# Commit and close
conn.commit()
conn.close()
```

- **Creating and executing SQL queries from Python using these connectors**
  - **SQLite3**
  - **Steps to use SQLite3 in python**

1. Connect to the SQLite database.
2. Create a cursor object.
3. Execute SQL queries (CREATE, INSERT, SELECT, UPDATE, DELETE).
4. Commit the changes (for write operations).
5. Close the connection.

Ex : Using SQLIte3 in python

```
import sqlite3

# Step 1: Connect to the database (creates a new file if not exists)
conn = sqlite3.connect("example.db")

# Step 2: Create a cursor object
cursor = conn.cursor()

# Step 3: Create a table
cursor.execute("""
CREATE TABLE IF NOT EXISTS employees (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    age INTEGER,
    salary REAL
)
""")

# Step 4: Insert data
cursor.execute("INSERT INTO employees (name, age, salary) VALUES (?, ?,
?)", ("Alice", 30, 50000))
conn.commit()  # Commit the transaction

# Step 5: Retrieve data
cursor.execute("SELECT * FROM employees")
rows = cursor.fetchall()
for row in rows:
    print(row)

# Step 6: Close the connection
conn.close()
```

  - **PyMySQL**
  - **Steps to use PyMySQLin Python:**
    1. Install PyMySQL (pip install pymysql).

2. Connect to a MySQL database.
3. Create a cursor object.
4. Execute SQL queries (CREATE, INSERT, SELECT, UPDATE, DELETE).
5. Commit the changes (for write operations).
6. Close the connection.

Ex : using PyMySQL in python

```
import pymysql

# Step 1: Connect to MySQL database
conn = pymysql.connect(
    host="localhost",
    user="root",
    password="yourpassword",
    database="company_db"
)

# Step 2: Create a cursor object
cursor = conn.cursor()

# Step 3: Create a table
cursor.execute("""
CREATE TABLE IF NOT EXISTS employees (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    age INT,
    salary FLOAT
)
""")

# Step 4: Insert data
cursor.execute("INSERT INTO employees (name, age, salary)
VALUES (%s, %s, %s)", ("Bob", 28, 60000))
conn.commit()  # Commit the transaction

# Step 5: Retrieve data
cursor.execute("SELECT * FROM employees")
rows = cursor.fetchall()
for row in rows:
    print(row)

# Step 6: Close the connection
conn.close(
```

10.Search and Match Functions

- **Using re.search() and re.match() functions in Python's re module for pattern matching.**

  Python's **re (regular expressions) module** allows pattern matching and text searching using various functions.

  Two commonly used functions for pattern matching are:

  1. **re.match(pattern, string)** – Matches only **at the beginning** of the string.
  2. **re.search(pattern, string)** – Searches the **entire** string for the pattern.

1. re.match() – Matches Only at the Start of the String

- Checks if the pattern is at the **beginning** of the string.
- If found, returns a **match object**; otherwise, returns None.

Ex :

```
import re

text = "Hello, welcome to Python regex!"

# Match the pattern at the start of the string
match = re.match(r"Hello", text)

if match:
   print("Match found:", match.group())
else:
   print("No match found")
```

O/P :

```
# Output: Match found: Hello
```

Key Point : If the pattern is **not at the start**, re.match() will return None.

2. re.search() – Searches the Entire String

- Looks for the **first occurrence** of the pattern **anywhere** in the string.
- Returns a **match object** if found, otherwise None.

Ex :

```
import re

text = "Hello, welcome to Python regex!"

# Search for "welcome" anywhere in the string
search_result = re.search(r"welcome", text)
```

```
if search_result:
    print("Pattern found at index:", search_result.start())
else:
    print("Pattern not found")
```

O/P:

```
# Output: Pattern found at index: 7
```

- **Difference between search and match.**

Feature	**re.search( )**	**re.match( )**
Function	Searches for a pattern anywhere in the string	Only checks for a match at the beginning of the string
Returns	The first match found (as a Match object)	Only matches if the pattern is at the start of the string
Use case	When you need to find a pattern anywhere in the text	When you need to check if the string starts with the pattern

Ex :

```
import re

text = "Python is amazing!"

# `match()` checks only at the start
match_result = re.match(r"is", text)
print("Match result:", match_result)   # Output: Match result: None

# `search()` finds "is" anywhere in the string
search_result = re.search(r"is", text)
print("Search result:", search_result.group())          # Output: is
```