Unit V: Principles of Object-Oriented Programming

Introduction to Object Oriented Programming principle, Benefits and applications of OOP

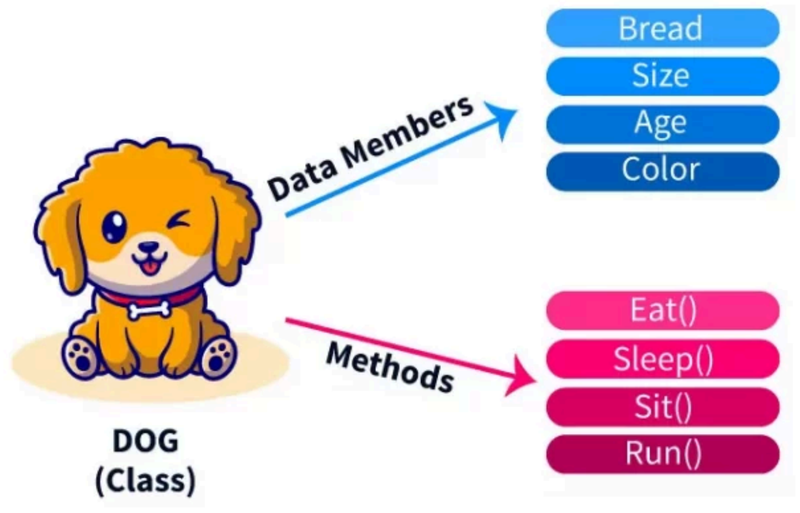
Object Oriented Programming (OOP) is a programming paradigm that is based on the concepts of “Objects” and “Classes”. OOP concepts in python makes the codes more reusable and easier to work with larger programs.
Some applications of OOP:

- Software Development
- Game Development
- Web Development
- Data Analysis and Machine Learning

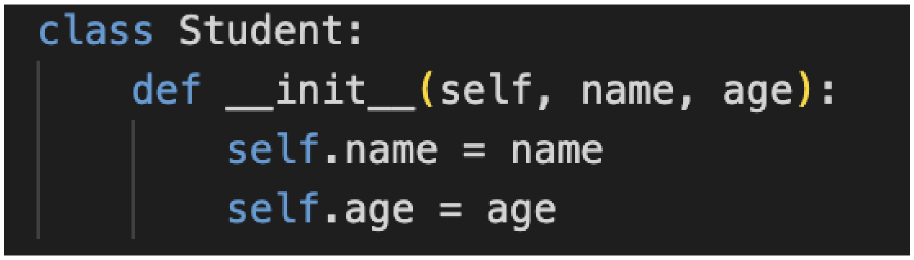
An **object** is a self-contained entity that encapsulates data (data members/ attributes/ property) and related operations (methods/ behaviors) that act on that data.

Think of an object as a real-world entity, like a car or a book. A car object would have attributes like color and model, and methods like accelerate, brake, and turn.

A **class** is a blueprint or template for creating objects. It defines the attributes and methods that all objects of that class will share. You can think of a class as a factory that produces objects. The class defines the specifications, and each object created from the class is an instance with its own set of data.



A class is a blueprint for creating objects. An object is an instance of a class.
Let's create a simple Student class having **attributes/properties** *name* and *age*



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

The __init__() method is called automatically when a new object is created. It is called a **constructor.** Constructors and destructors are special methods used for initializing and cleaning up objects, respectively.

Types of constructors :

Default constructor: The default constructor is a simple constructor which doesn't accept any arguments. Its definition has only one argument which is a reference to the instance being constructed known as 'self'.

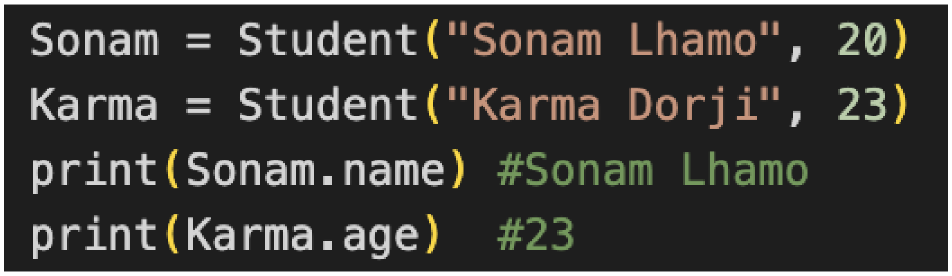
Parameterized constructor: constructor with parameters is known as parameterized constructor. The parameterized constructor takes its first argument as a reference to the instance being constructed known as 'self' and the rest of the arguments are provided by the programmer.

self is a convention used to represent the instance of the class itself. When you define a method within a class, and you want to access or modify the attributes of that instance, you use self as the first parameter to refer to that instance.

Destructors are called when an object gets destroyed. In Python, destructors are not needed as much as in C++ because Python has a garbage collector that handles memory management automatically.

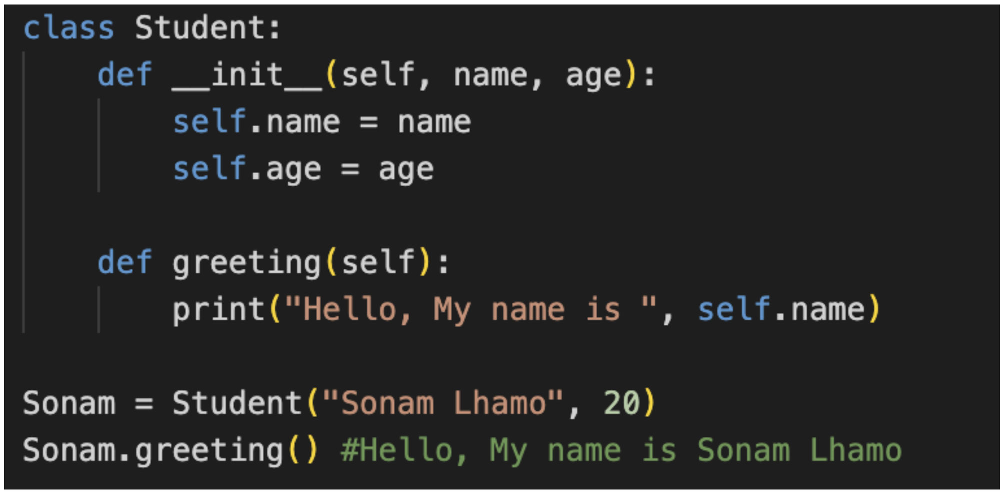
The __del__() method is known as a destructor method in Python. It is called when all references to the object have been deleted i.e when an object is garbage collected.

Create Student objects by calling the Student() class:



```
Sonam = Student("Sonam Lhamo", 20)
Karma = Student("Karma Dorji", 23)
print(Sonam.name) #Sonam Lhamo
print(Karma.age)  #23
```

Class Method
Classes can also contain methods/behaviors which are functions defined inside the class.
Let's add a method to our Student class:



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

    def greeting(self):
        print("Hello, My name is ", self.name)

Sonam = Student("Sonam Lhamo", 20)
Sonam.greeting() #Hello, My name is Sonam Lhamo
```

NOTE: The greeting() method can access the object's attributes using self. This allows encapsulation of data and functions.

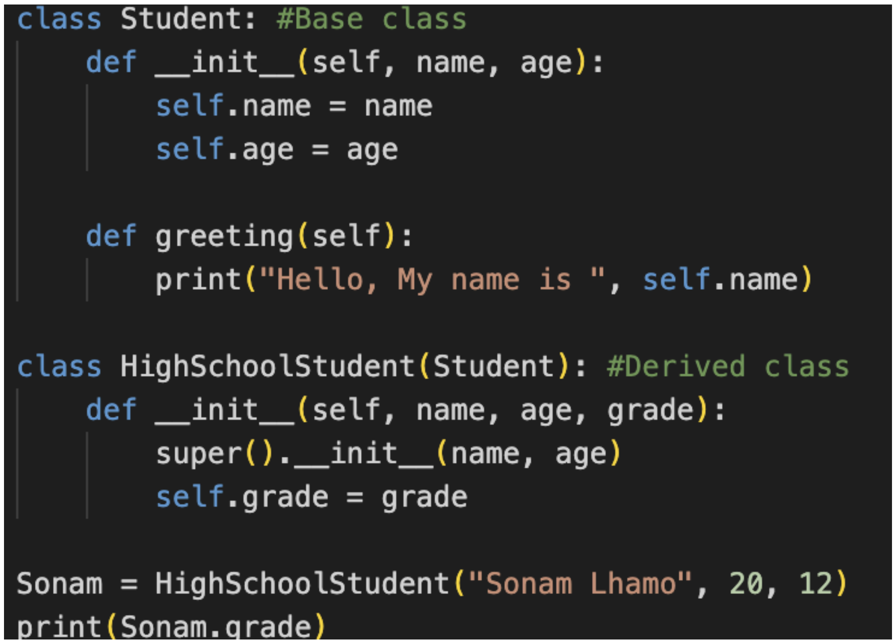
Principles of OOP

Encapsulation - Binding data and functions into a single unit called class.
Abstraction - Hiding internal details and showing only essential features.
Inheritance - Ability to create new classes from existing classes.
Polymorphism - Ability to use common operations in different forms for different data inputs.

1. Inheritance

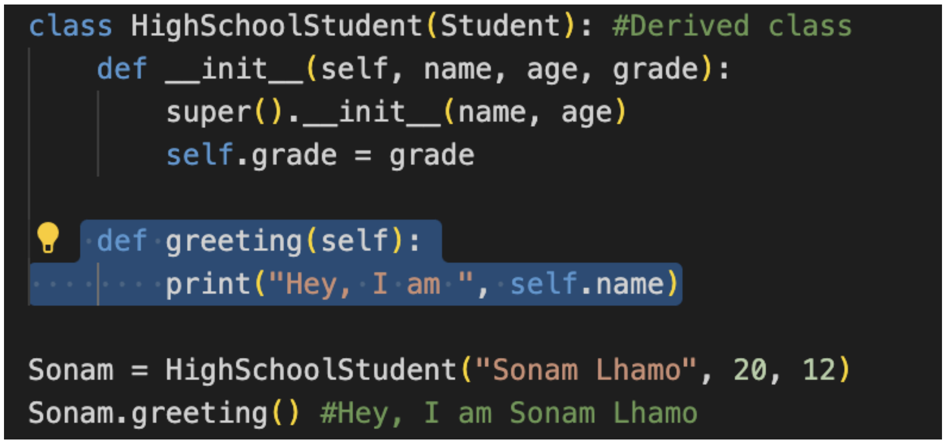
Inheritance is a way of creating a new class for using details of an existing class without modifying it. The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class).

Let's create a HighSchoolStudent class that inherits from Student:



```python
class Student: #Base class
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greeting(self):
        print("Hello, My name is ", self.name)

class HighSchoolStudent(Student): #Derived class
    def __init__(self, name, age, grade):
        super().__init__(name, age)
        self.grade = grade

Sonam = HighSchoolStudent("Sonam Lhamo", 20, 12)
print(Sonam.grade)
```

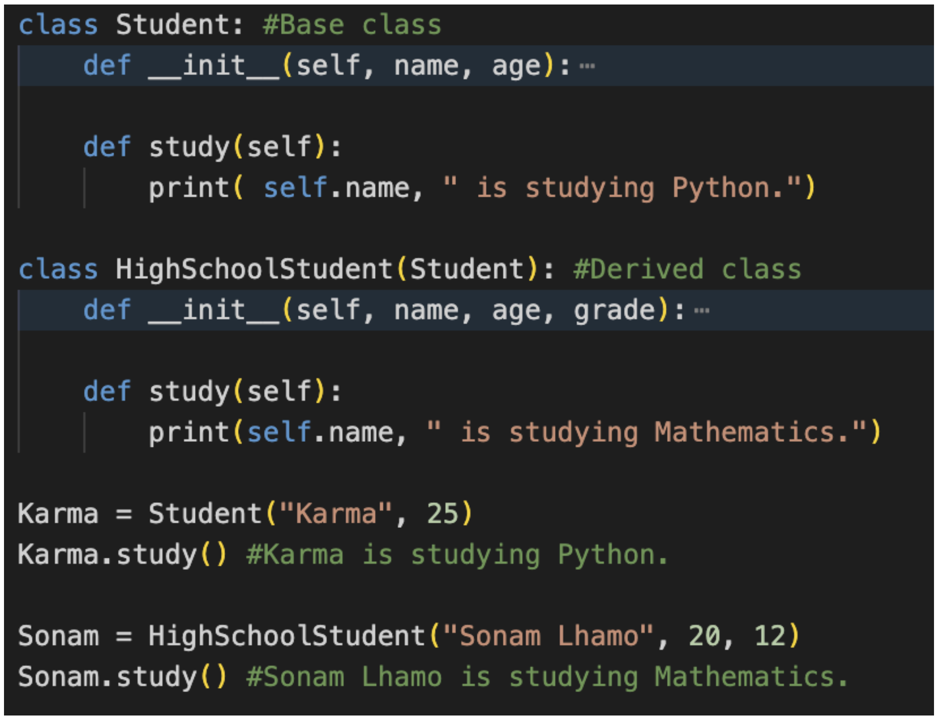
We can also override methods in the derived class:



```python
class HighSchoolStudent(Student): #Derived class
    def __init__(self, name, age, grade):
        super().__init__(name, age)
        self.grade = grade

    def greeting(self):
        print("Hey, I am ", self.name)

Sonam = HighSchoolStudent("Sonam Lhamo", 20, 12)
Sonam.greeting() #Hey, I am Sonam Lhamo
```

2. Polymorphism

Polymorphism allows common operations to take on different forms for different data inputs. For example, we can have a common study() method that prints different outputs for different student objects:

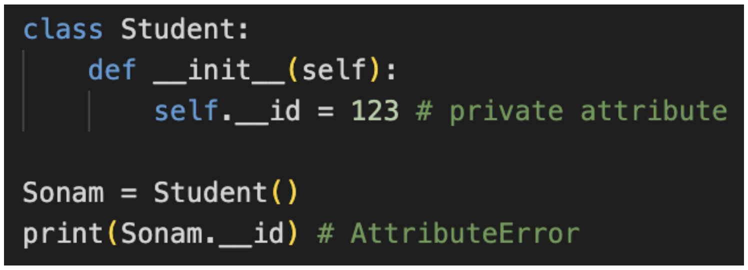


```
class Student: #Base class
    def __init__(self, name, age):⋯

    def study(self):
        print( self.name, " is studying Python.")

class HighSchoolStudent(Student): #Derived class
    def __init__(self, name, age, grade):⋯

    def study(self):
        print(self.name, " is studying Mathematics.")

Karma = Student("Karma", 25)
Karma.study() #Karma is studying Python.

Sonam = HighSchoolStudent("Sonam Lhamo", 20, 12)
Sonam.study() #Sonam Lhamo is studying Mathematics.
```

The study() method exhibits polymorphic behavior based on the object calling it. This allows common methods to work in different ways for derived classes.

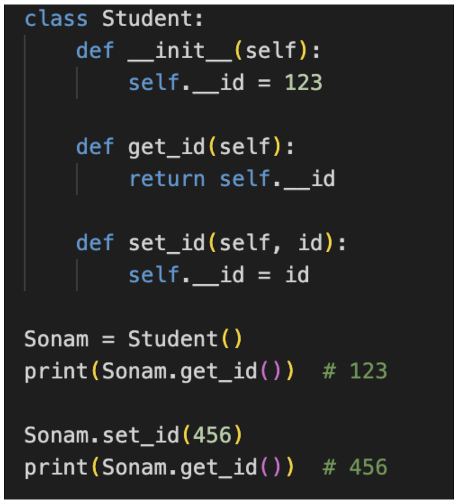
3. Encapsulation

Encapsulation refers to the bundling of attributes and methods inside a single class. A private attribute is an attribute of a class that is intended to be restricted in access, meaning it cannot be accessed or modified directly from outside the class. In Python, we denote private attributes using double underscore prefixes. For example:



```python
class Student:
    def __init__(self):
        self.__id = 123 # private attribute

Sonam = Student()
print(Sonam.__id) # AttributeError
```

To access or modify private attributes, we use setter and getter methods:



```python
class Student:
    def __init__(self):
        self.__id = 123

    def get_id(self):
        return self.__id

    def set_id(self, id):
        self.__id = id

Sonam = Student()
print(Sonam.get_id())  # 123

Sonam.set_id(456)
print(Sonam.get_id())  # 456
```

This encapsulates the data and provides controlled access to class attributes.

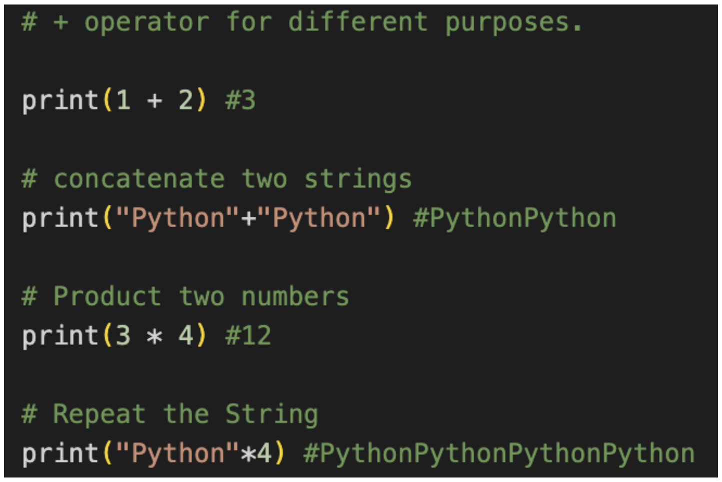
4. Abstraction

Abstraction focuses on necessary attributes and behaviors hiding unnecessary details. It allows us to focus on what an object does rather than how it does it. We can achieve abstraction in Python using abstract base classes and interfaces.

While it's true that you can achieve abstraction without using abstract methods and classes, leveraging them provides a more structured and explicit approach, which can lead to better code organization, readability, and maintainability in larger projects or when working collaboratively with others.

Operator Overloading

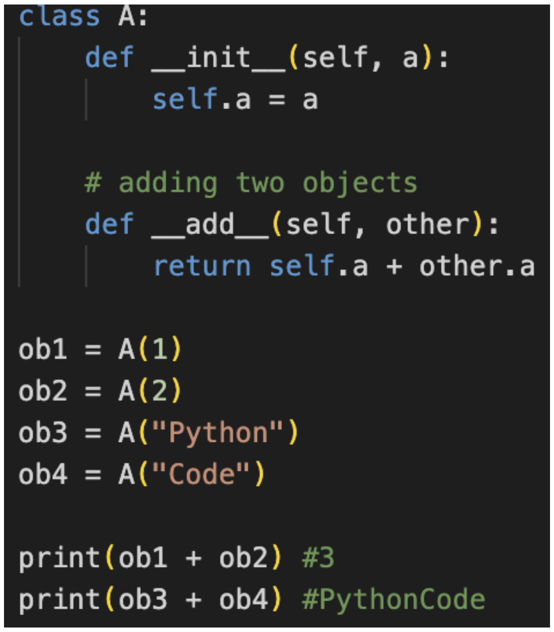
Operator Overloading means giving extended meaning beyond their predefined operational meaning. For example operator + is used to add two integers as well as join two strings and merge two lists. It is achievable because ‘+’ operator is overloaded by int class and str class.

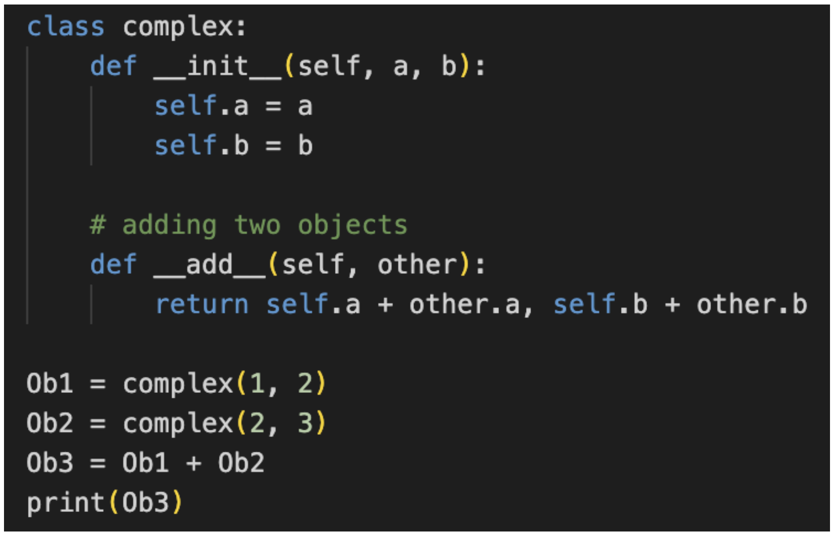


```
# + operator for different purposes.

print(1 + 2) #3

# concatenate two strings
print("Python"+"Python") #PythonPython

# Product two numbers
print(3 * 4) #12

# Repeat the String
print("Python"*4) #PythonPythonPythonPython
```

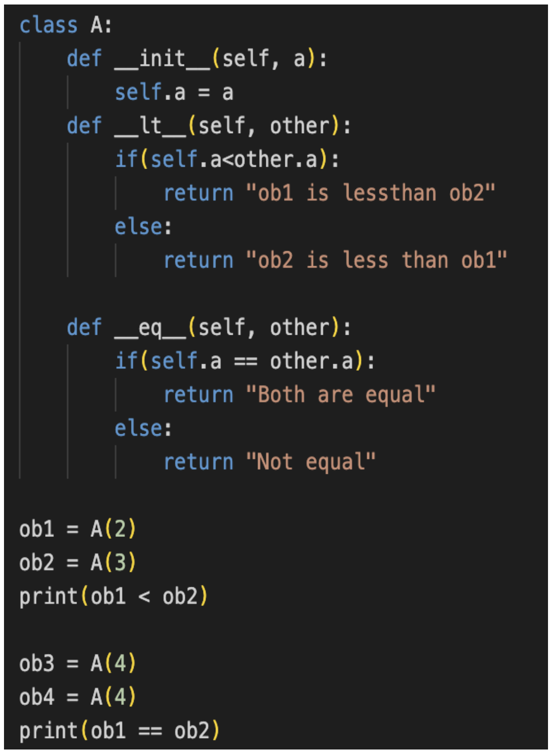
How to overload the operators in Python?
Consider that we have two objects which are a physical representation of a class (user-defined data type) and we have to add two objects with binary ‘+’ operator, it throws an error, because the compiler doesn't know how to add two objects. So we define a method for an operator and that process is called operator overloading. We can overload all existing operators but we can’t create a new operator.

To perform operator overloading, Python provides some special function or magic function that is automatically invoked when it is associated with that particular operator. For example, when we use + operator, the magic method __add__ is automatically invoked in which the operation for + operator is defined.

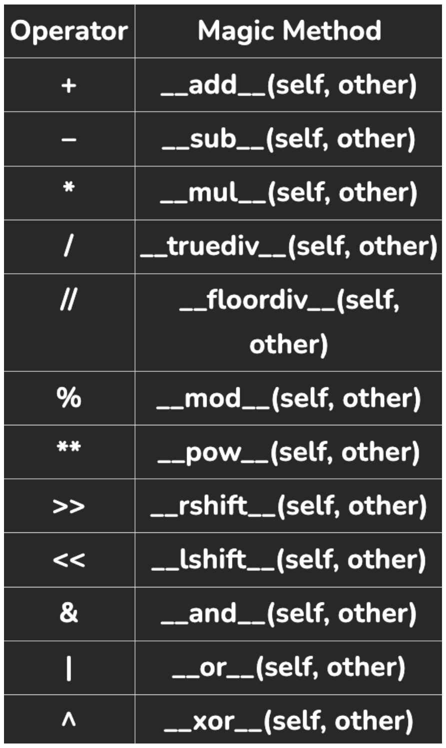

```python
class A:
    def __init__(self, a):
        self.a = a

    # adding two objects
    def __add__(self, other):
        return self.a + other.a

ob1 = A(1)
ob2 = A(2)
ob3 = A("Python")
ob4 = A("Code")

print(ob1 + ob2) #3
print(ob3 + ob4) #PythonCode
```


```python
class complex:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    # adding two objects
    def __add__(self, other):
        return self.a + other.a, self.b + other.b

Ob1 = complex(1, 2)
Ob2 = complex(2, 3)
Ob3 = Ob1 + Ob2
print(Ob3)
```

The output for the above example is (3, 5)



```python
class A:
    def __init__(self, a):
        self.a = a
    def __lt__(self, other):
        if(self.a<other.a):
            return "ob1 is lessthan ob2"
        else:
            return "ob2 is less than ob1"

    def __eq__(self, other):
        if(self.a == other.a):
            return "Both are equal"
        else:
            return "Not equal"

ob1 = A(2)
ob2 = A(3)
print(ob1 < ob2)

ob3 = A(4)
ob4 = A(4)
print(ob1 == ob2)
```

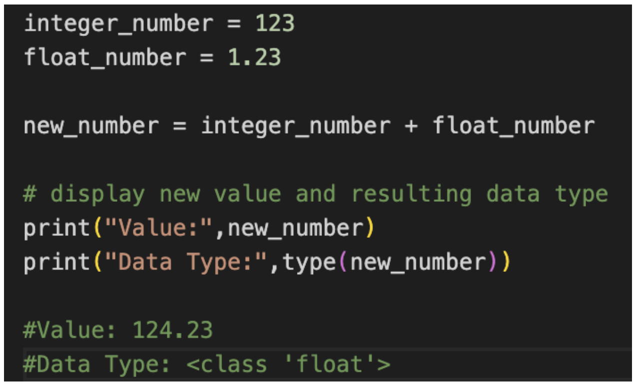


Operator	Magic Method
+	__add__(self, other)
–	__sub__(self, other)
*	__mul__(self, other)
/	__truediv__(self, other)
//	__floordiv__(self, other)
%	__mod__(self, other)
**	__pow__(self, other)
>>	__rshift__(self, other)
<<	__lshift__(self, other)
&	__and__(self, other)
|	__or__(self, other)
^	__xor__(self, other)

Type conversions
Type conversion is the process of converting data of one type to another.
For example: converting *int* data to *str*.

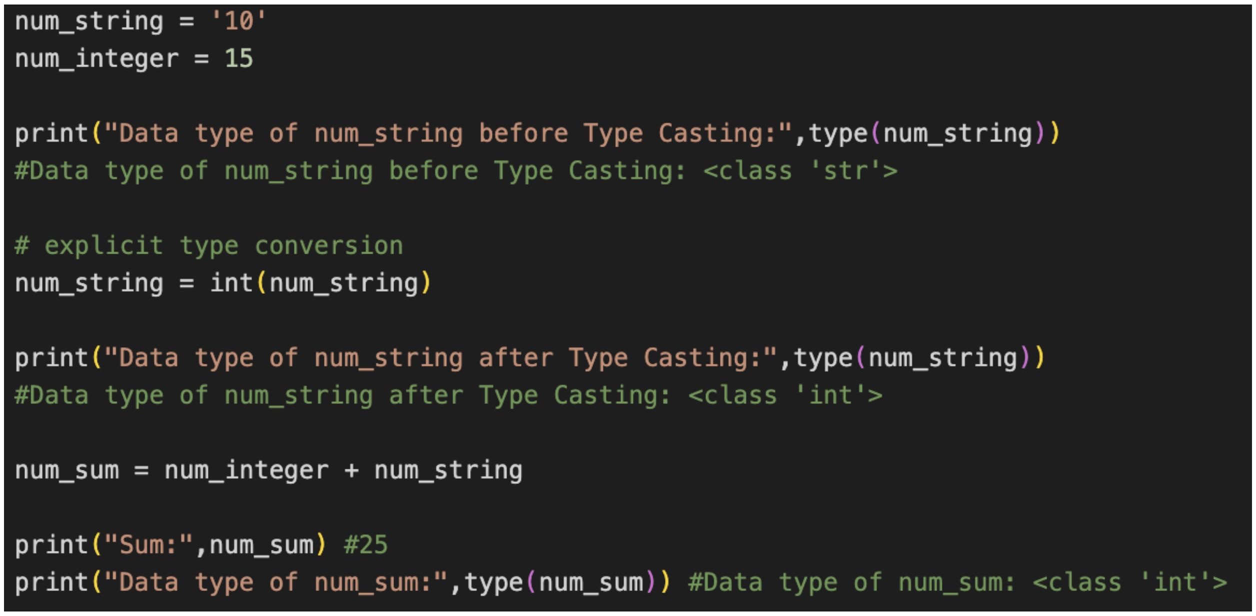
There are two types of type conversion in Python.
Implicit Conversion - automatic type conversion
Explicit Conversion - manual type conversion

Implicit conversion
In certain situations, Python automatically converts one data type to another.
Example 1: Converting integer to float



```python
integer_number = 123
float_number = 1.23

new_number = integer_number + float_number

# display new value and resulting data type
print("Value:",new_number)
print("Data Type:",type(new_number))

#Value: 124.23
#Data Type: <class 'float'>
```

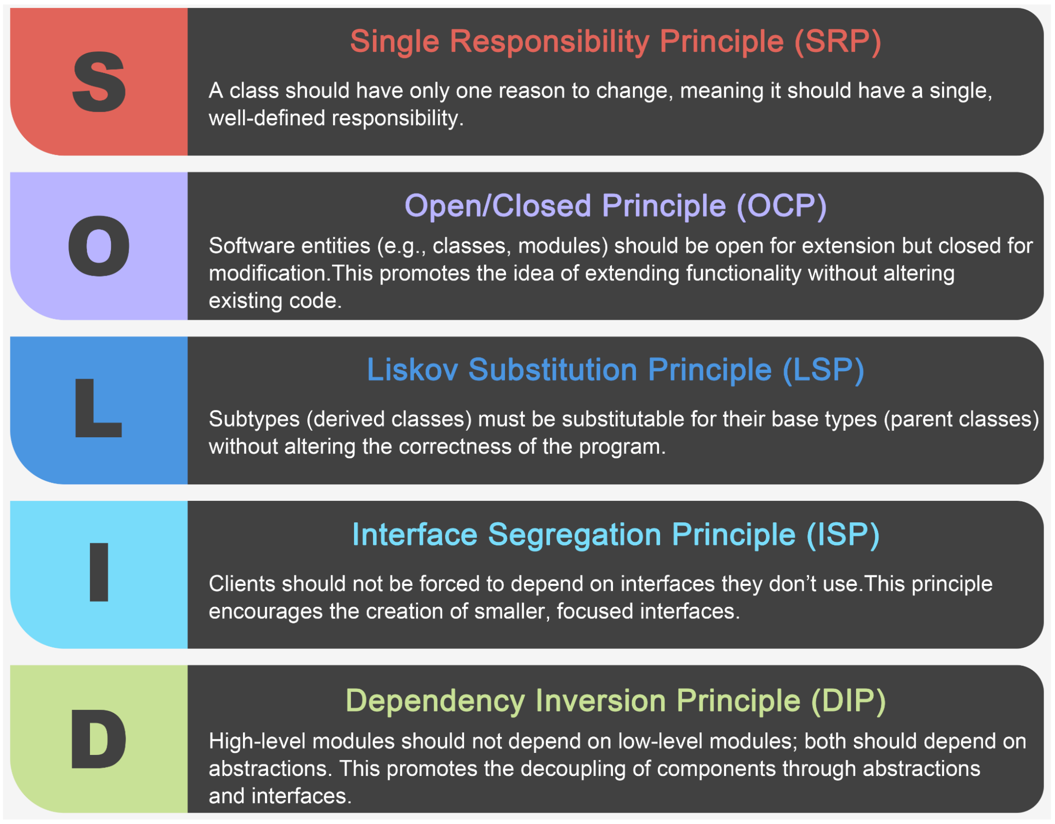
Explicit Conversion
In Explicit Type Conversion, users convert the data type of an object to required data type. We use the built-in functions like *int(), float(), str(),* etc to perform explicit type conversion. This type of conversion is also called **typecasting** because the user casts (changes) the data type of the objects.



```python
num_string = '10'
num_integer = 15

print("Data type of num_string before Type Casting:",type(num_string))
#Data type of num_string before Type Casting: <class 'str'>

# explicit type conversion
num_string = int(num_string)

print("Data type of num_string after Type Casting:",type(num_string))
#Data type of num_string after Type Casting: <class 'int'>

num_sum = num_integer + num_string

print("Sum:",num_sum) #25
print("Data type of num_sum:",type(num_sum)) #Data type of num_sum: <class 'int'>
```

SOLID Principle
When you build a Python project using object-oriented programming (OOP), planning how the different classes and objects will interact to solve your specific problems is an important part of the job. This planning is known as **object-oriented design (OOD)**.

SOLID is a set of five object-oriented design principles that can help you write more **maintainable, flexible, and scalable** code based on **well-designed, cleanly structured classes.**



S
Single Responsibility Principle (SRP)
A class should have only one reason to change, meaning it should have a single, well-defined responsibility.

O
Open/Closed Principle (OCP)
Software entities (e.g., classes, modules) should be open for extension but closed for modification.This promotes the idea of extending functionality without altering existing code.

L
Liskov Substitution Principle (LSP)
Subtypes (derived classes) must be substitutable for their base types (parent classes) without altering the correctness of the program.

I
Interface Segregation Principle (ISP)
Clients should not be forced to depend on interfaces they don't use.This principle encourages the creation of smaller, focused interfaces.

D
Dependency Inversion Principle (DIP)
High-level modules should not depend on low-level modules; both should depend on abstractions. This promotes the decoupling of components through abstractions and interfaces.

1. **Single Responsibility Principle (SRP)**
   A class should have only one reason to change.

   This means that a class should have only one responsibility, as expressed through its methods. If a class takes care of more than one task, then you should separate those tasks into separate classes.

   Consider a restaurant kitchen. The chef is responsible for cooking the food, while the waiter is responsible for taking orders and serving the food to customers. The chef does not take orders, and the waiter does not cook the food. Each person has a single responsibility, making the system more organized and efficient.

2. **Open/Closed Principle (OCP)**
   Classes should be open for extension and closed to modification.

   Means you should be able to extend a class behavior, without modifying it.

   Think about a TV remote control. You can add new features or functionalities to the remote control by adding new buttons or interfaces, without modifying the existing buttons or circuits. For example, you can add a button to control the volume without changing the channel buttons.

3. **Liskov Substitution Principle (LSP)**
   Subclasses should be substitutable for their base classes.

   This means that if you have a base class and a derived class, you should be able to use objects of the derived class wherever objects of the base class are expected, without breaking the program.

4. **Interface Segregation Principle (ISP)**
   Many client-specific interfaces are better than one general-purpose interface.

   Means that it's better to have multiple specific interfaces than a single general-purpose interface.

   Imagine a multi-function printer that can print, scan, and fax documents. Instead of having a single, general-purpose interface for the printer, it would be better to have separate interfaces for printing, scanning, and faxing. This way, a client that only needs to print documents does not depend on the scanning or faxing interfaces, making the system more modular and easier to maintain.

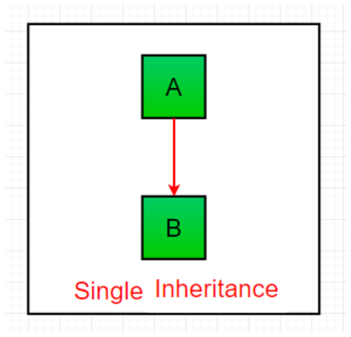
5. **Dependency Inversion Principle (DIP)**
   Classes should depend upon interfaces or abstract classes instead of concrete classes and functions.

Types of Inheritance
Types of Inheritance depend upon the number of child and parent classes involved. There are five types of inheritance in Python:

1. Single Inheritance
2. Multiple Inheritance
3. Multilevel Inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance

1. Single Inheritance

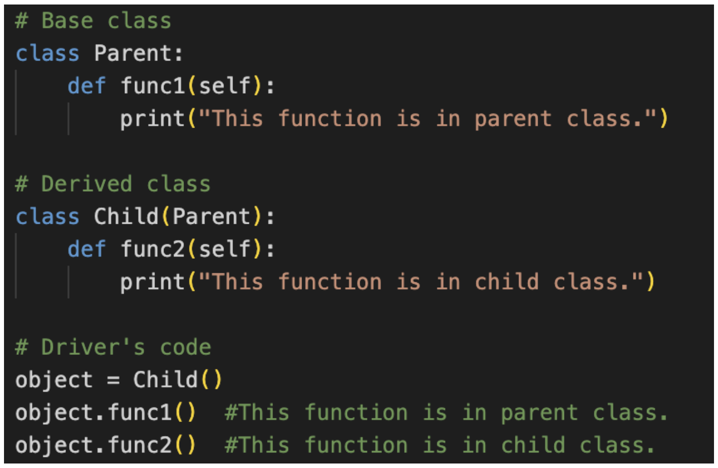
Single inheritance enables a derived class to inherit properties from a single parent class, thus enabling code reusability and the addition of new features to existing code.



Syntax:

```
class ParentClass:
   # Parent class definition

class ChildClass(ParentClass):
   # Child class definition
```

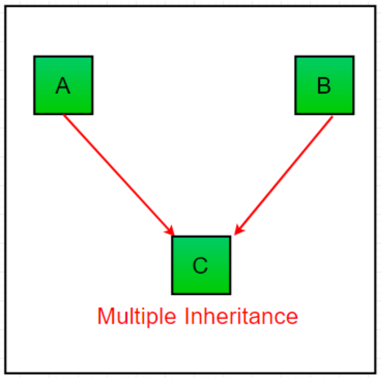
Example:



```
# Base class
class Parent:
    def func1(self):
        print("This function is in parent class.")

# Derived class
class Child(Parent):
    def func2(self):
        print("This function is in child class.")

# Driver's code
object = Child()
object.func1()  #This function is in parent class.
object.func2()  #This function is in child class.
```

2. Multiple Inheritance

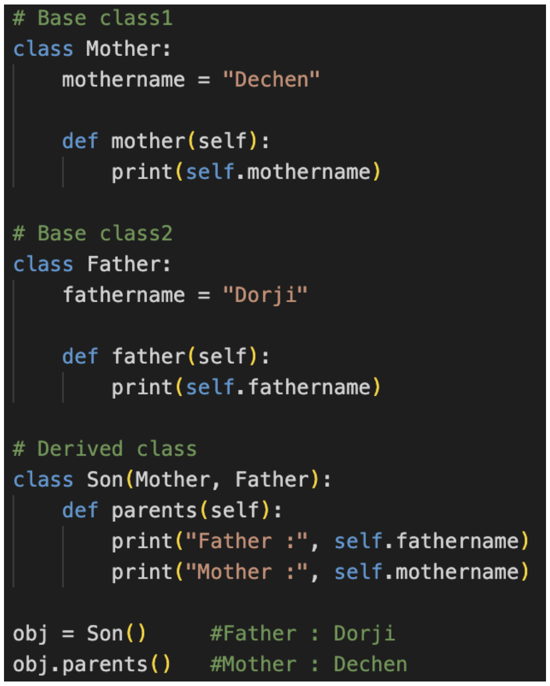
When a class can be derived from more than one base class this type of inheritance is called multiple inheritances. In multiple inheritances, all the features of the base classes are inherited into the derived class.



Syntax:

```
class ParentClass1:
   # Parent class 1 definition

class ParentClass2:
   # Parent class 2 definition

class ChildClass(ParentClass1, ParentClass2):
   # Child class definition
```

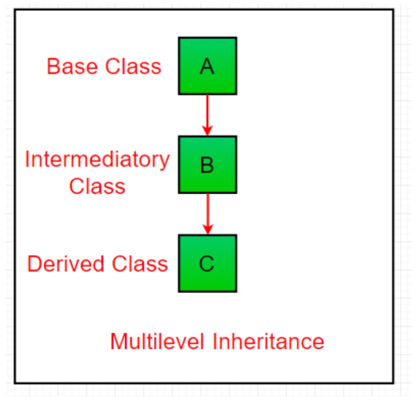
Example:



```
# Base class1
class Mother:
    mothername = "Dechen"

    def mother(self):
        print(self.mothername)

# Base class2
class Father:
    fathername = "Dorji"

    def father(self):
        print(self.fathername)

# Derived class
class Son(Mother, Father):
    def parents(self):
        print("Father :", self.fathername)
        print("Mother :", self.mothername)

obj = Son()      #Father : Dorji
obj.parents()    #Mother : Dechen
```

3. **Multilevel Inheritance**

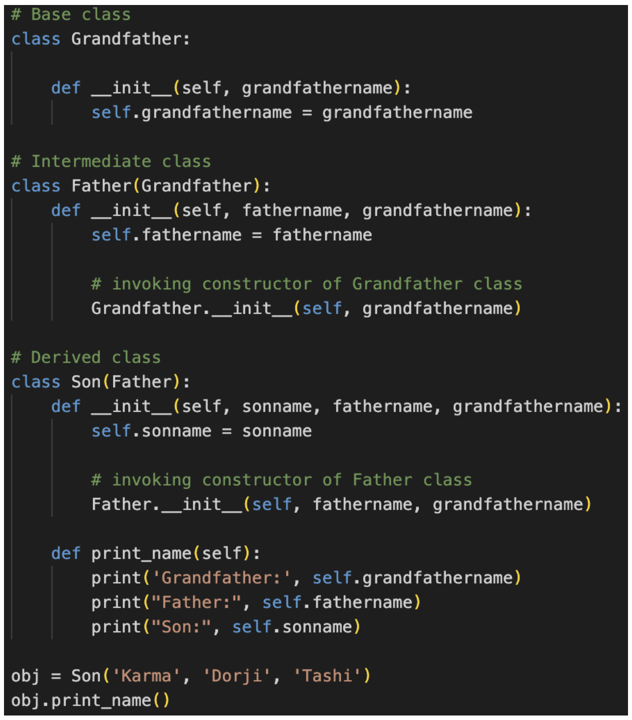
In multilevel inheritance, features of the base class and the derived class are further inherited into the new derived class. This is similar to a relationship representing a child and a grandfather.



Syntax:

```
class GrandParentClass:
    # Grandparent class definition

class ParentClass(GrandParentClass):
    # Parent class definition

class ChildClass(ParentClass):
    # Child class definition
```

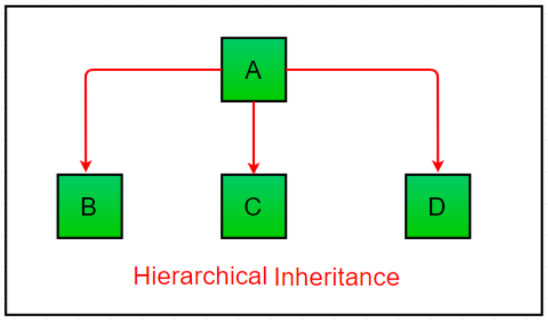
Example:



```
# Base class
class Grandfather:

    def __init__(self, grandfathername):
        self.grandfathername = grandfathername

# Intermediate class
class Father(Grandfather):
    def __init__(self, fathername, grandfathername):
        self.fathername = fathername

        # invoking constructor of Grandfather class
        Grandfather.__init__(self, grandfathername)

# Derived class
class Son(Father):
    def __init__(self, sonname, fathername, grandfathername):
        self.sonname = sonname

        # invoking constructor of Father class
        Father.__init__(self, fathername, grandfathername)

    def print_name(self):
        print('Grandfather:', self.grandfathername)
        print("Father:", self.fathername)
        print("Son:", self.sonname)

obj = Son('Karma', 'Dorji', 'Tashi')
obj.print_name()
```

4. **Hierarchical Inheritance**

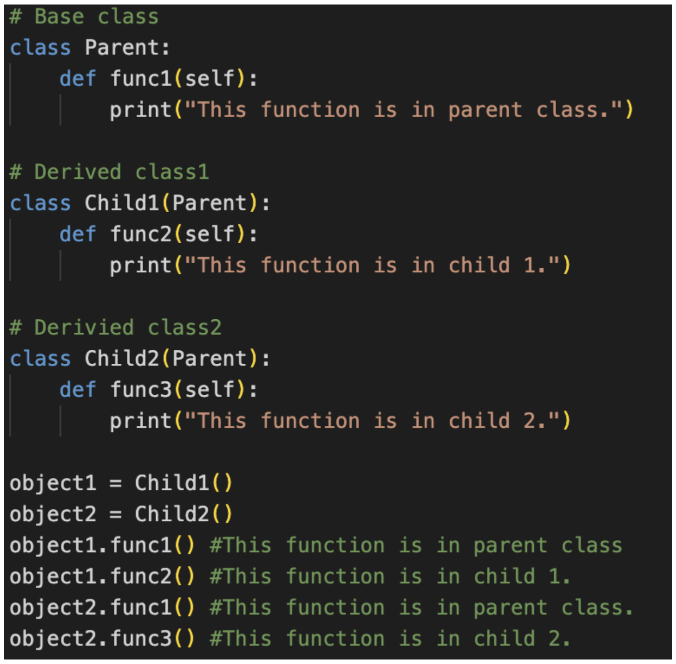
   When more than one derived class is created from a single base this type of inheritance is called hierarchical inheritance. In this program, we have a parent (base) class and two child (derived) classes.



Syntax:

```
class ParentClass:
# Parent class definition

class ChildClass1(ParentClass):
# Child class 1 definition

class ChildClass2(ParentClass):
# Child class 2 definition
```

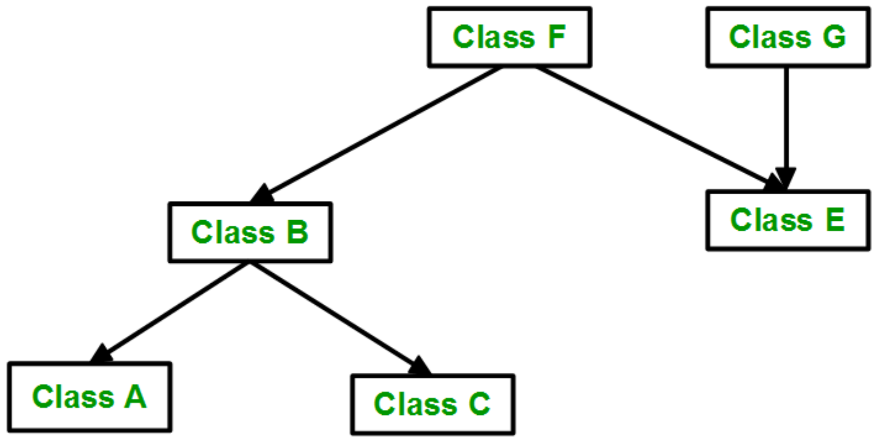
Example:



```
# Base class
class Parent:
    def func1(self):
        print("This function is in parent class.")

# Derived class1
class Child1(Parent):
    def func2(self):
        print("This function is in child 1.")

# Derivied class2
class Child2(Parent):
    def func3(self):
        print("This function is in child 2.")

object1 = Child1()
object2 = Child2()
object1.func1() #This function is in parent class
object1.func2() #This function is in child 1.
object2.func1() #This function is in parent class.
object2.func3() #This function is in child 2.
```

5. **Hybrid Inheritance**

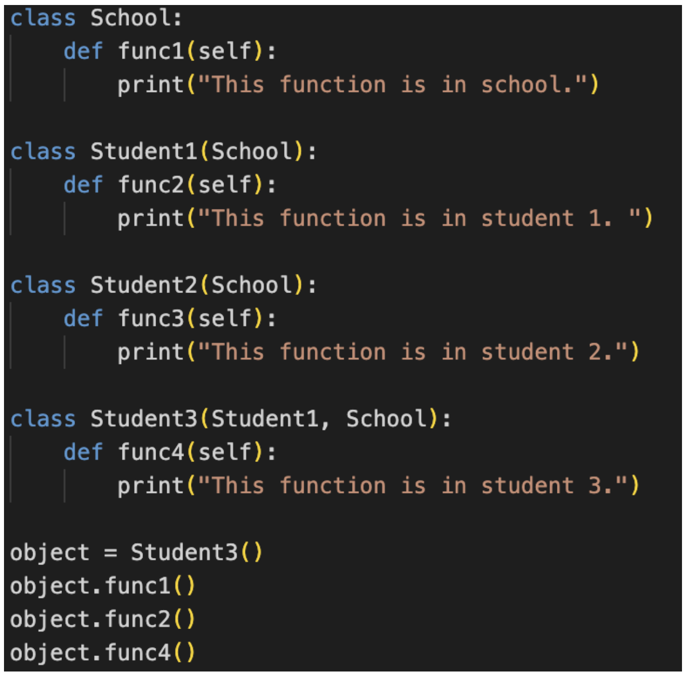
Inheritance consisting of multiple types of inheritance is called hybrid inheritance.



Syntax:

```
class ParentClass:
# Parent class definition

class ChildClass1(ParentClass):
# Child class 1 definition

class ChildClass2(ParentClass, ChildClass1):
# Child class 2 definition
```

Example:



```
class School:
    def func1(self):
        print("This function is in school.")

class Student1(School):
    def func2(self):
        print("This function is in student 1. ")

class Student2(School):
    def func3(self):
        print("This function is in student 2.")

class Student3(Student1, School):
    def func4(self):
        print("This function is in student 3.")

object = Student3()
object.func1()
object.func2()
object.func4()
```

Reference

Guttag, J. V. (2013). Introduction to computation and programming using Python. http://cds.cern.ch/record/2622221

Bader, D., Jablonski, J., & Heisler, F. (2021). Python Basics: A Practical Introduction to Python 3. Real Python (Realpython.Com).

GeeksforGeeks. (2024a, March 15). SOLID principles in programming understand with real life examples. GeeksforGeeks.

Python, R. (2023, June 16). SOLID Principles: Improve Object-Oriented Design in Python. https://realpython.com/solid-principles-python/

GeeksforGeeks. (2022, July 7). Types of inheritance Python. GeeksforGeeks. https://www.geeksforgeeks.org/types-of-inheritance-python/