Object Oriented Programming

Anab Batool Kazmi

Object Oriented Programming

- Object-oriented programming is a **programming paradigm** that provides a means of structuring programs so that **properties and behaviors** are bundled into **individual objects**.
- Python is a versatile programming language that supports various programming styles, including **object-oriented programming** (OOP) through the use of **objects** and **classes**.

Object and class

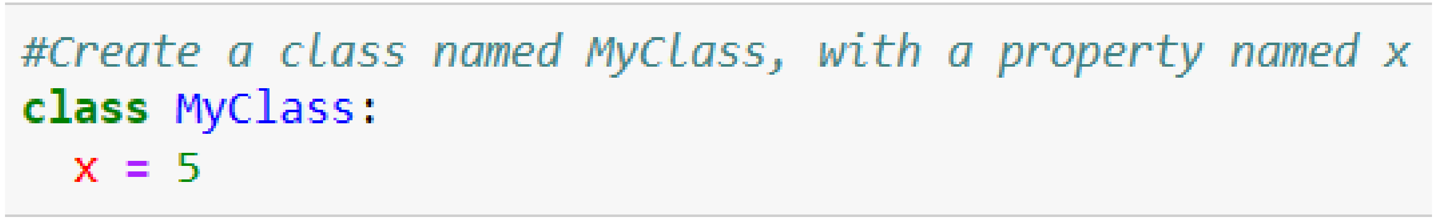
- An **object** is any entity that has **attributes** and **behaviors**. For example, a parrot is an object. It has
  - **attributes** - name, age, color, etc.
  - **behavior** - dancing, singing, etc.
- A class is a **blueprint** for that object.

Create a Class

- To create a class in Python, you use the **class keyword followed by the class name and a colon**.

class ClassName:

        #body of class



```
#Create a class named MyClass, with a property named x
class MyClass:
  x = 5
```

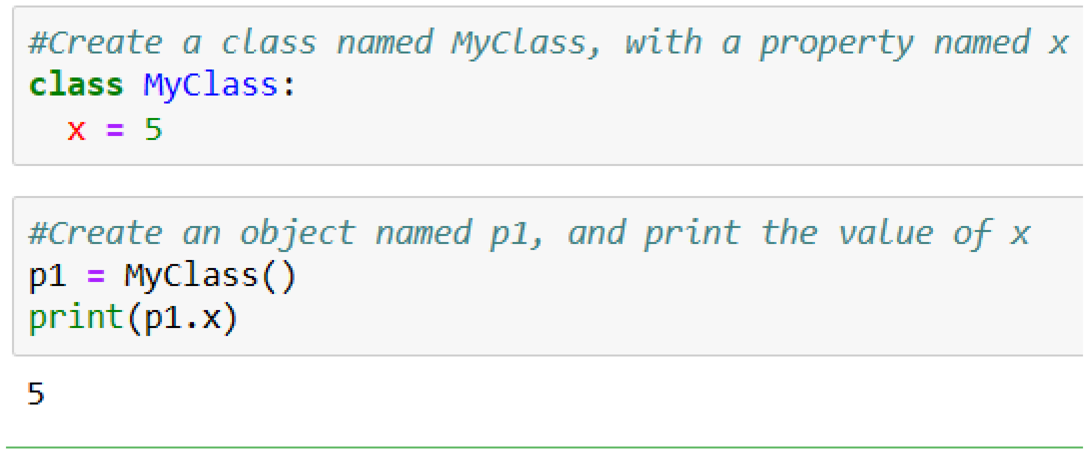
Create Object

- The syntax to create an object in Python is to use the class name followed by parentheses.

objectName=ClassName()

- You can also pass arguments to the constructor if the class has one.

objectName=ClassName(arg1,arg2,..)



```
#Create a class named MyClass, with a property named x
class MyClass:
  x = 5
```

```
#Create an object named p1, and print the value of x
p1 = MyClass()
print(p1.x)
```

```
5
```

Constructor

- In Python, a constructor is a **special method** that is called **when an object of a class is created**.
- It is used to **initialize the attributes** (variables) of an object.
- The constructor is a crucial part of object-oriented programming, as it allows you to **define the initial state** of an object and ensure that it is properly configured before being used.
- It is automatically called when a **new object of a class is created**.

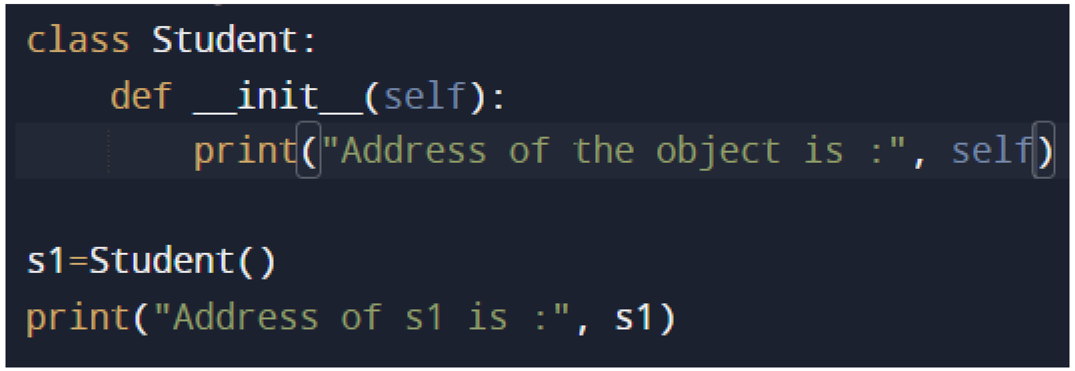
Constructor

- The constructor method is always named __init__.
- This convention makes it easy to identify and distinguish constructors from other methods in a class.
- When a new object is created, the __init__ method is automatically called, and **it receives the reference of newly created object as its first argument**.
- The first argument is traditionally named **self,** and it provides access to the object's attributes and methods.

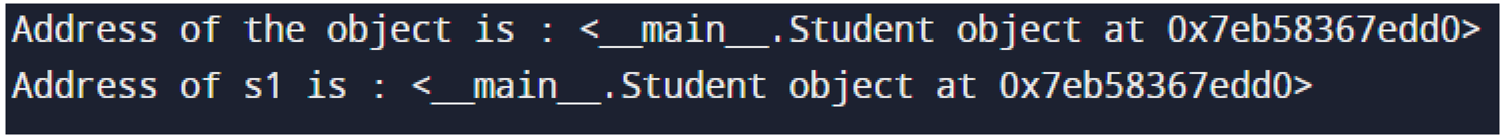
```
def __init__(self):
    #initializations
```

Constructor

- First argument contains the reference of newly created object.



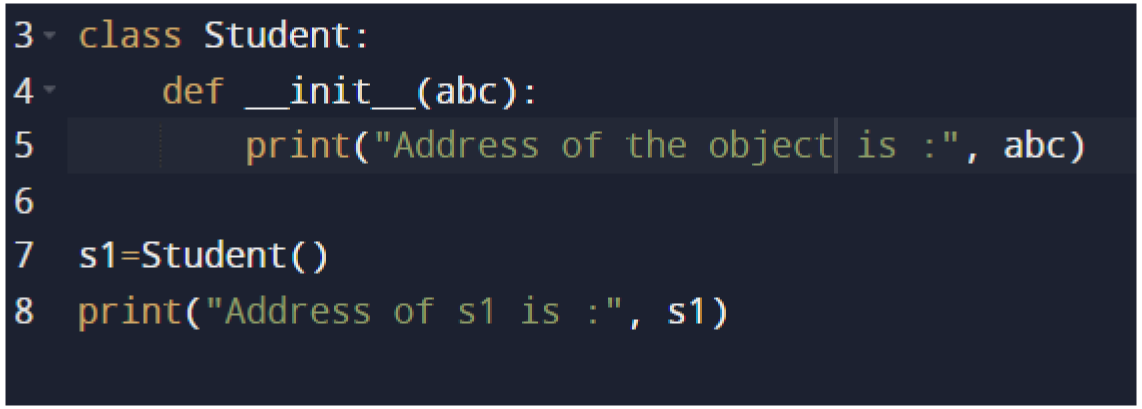
```python
class Student:
    def __init__(self):
        print("Address of the object is :", self)

s1=Student()
print("Address of s1 is :", s1)
```



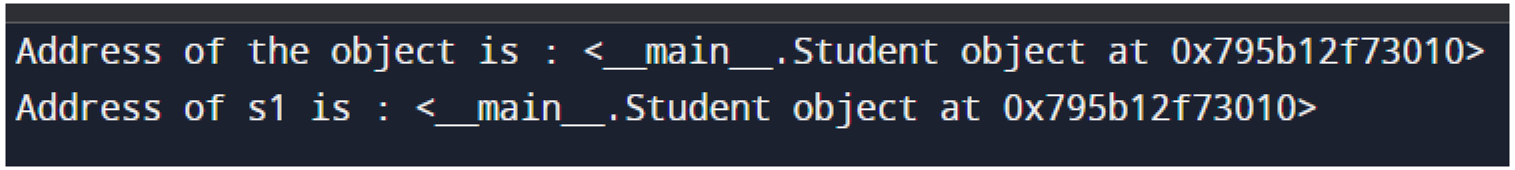
```
Address of the object is : <__main__.Student object at 0x7eb58367edd0>
Address of s1 is : <__main__.Student object at 0x7eb58367edd0>
```

Constructor

- First argument can have any name, but traditionally it is named as **self**

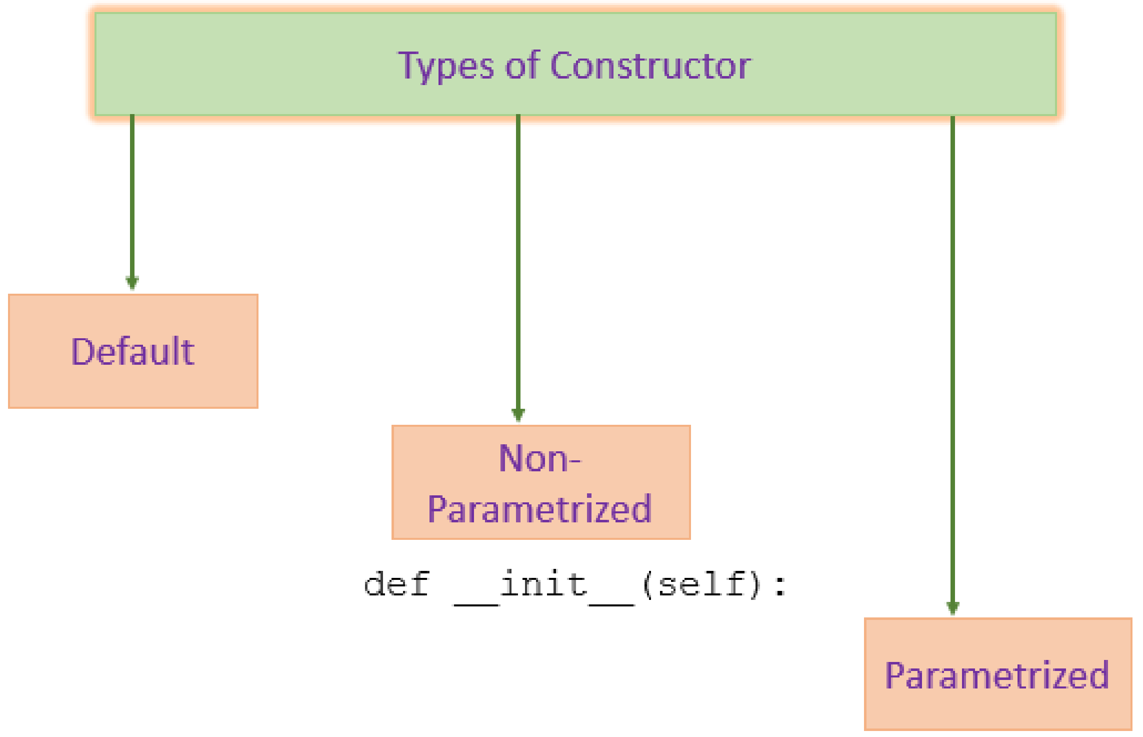


```python
class Student:
    def __init__(abc):
        print("Address of the object is :", abc)

s1=Student()
print("Address of s1 is :", s1)
```



```
Address of the object is : <__main__.Student object at 0x795b12f73010>
Address of s1 is : <__main__.Student object at 0x795b12f73010>
```

Types of constructor



```
def __init__(self):
```

```
def __init__(self, v1, v2…, vn):
```

Default Constructor

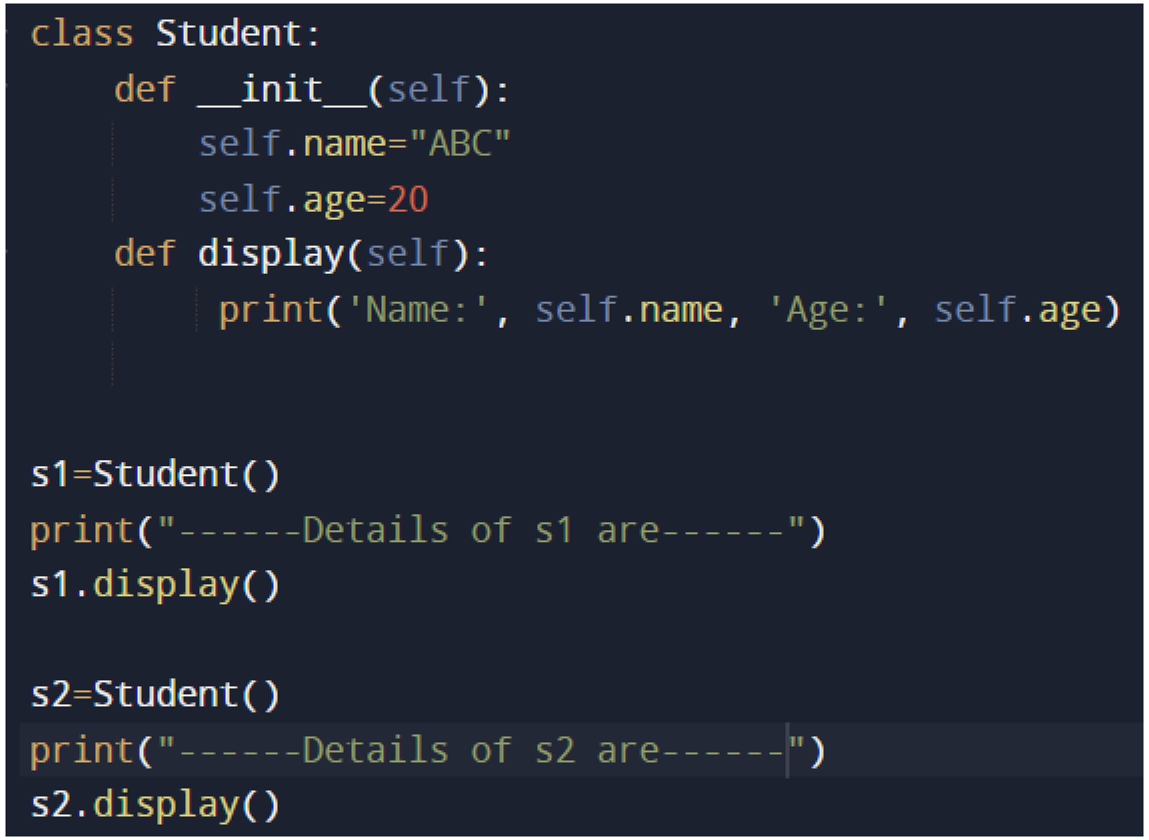
- Python will provide a default constructor if no constructor is defined.
- Python adds a default constructor when we do not include the constructor in the class or forget to declare it.
- It does not perform any task but initializes the objects.
- It is an empty constructor without a body.

Non-Parametrized Constructor

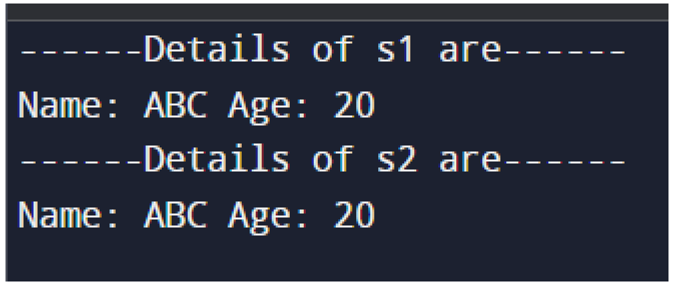
- A constructor without any arguments is called a non-parameterized constructor.
- This type of constructor is used **to initialize each object with default values.**

- This constructor doesn’t accept the arguments during object creation. Instead, it initializes every object with the same set of values.

Non-Parametrized Constructor



```
class Student:
    def __init__(self):
        self.name="ABC"
        self.age=20
    def display(self):
         print('Name:', self.name, 'Age:', self.age)


s1=Student()
print("------Details of s1 are------")
s1.display()

s2=Student()
print("------Details of s2 are------")
s2.display()
```



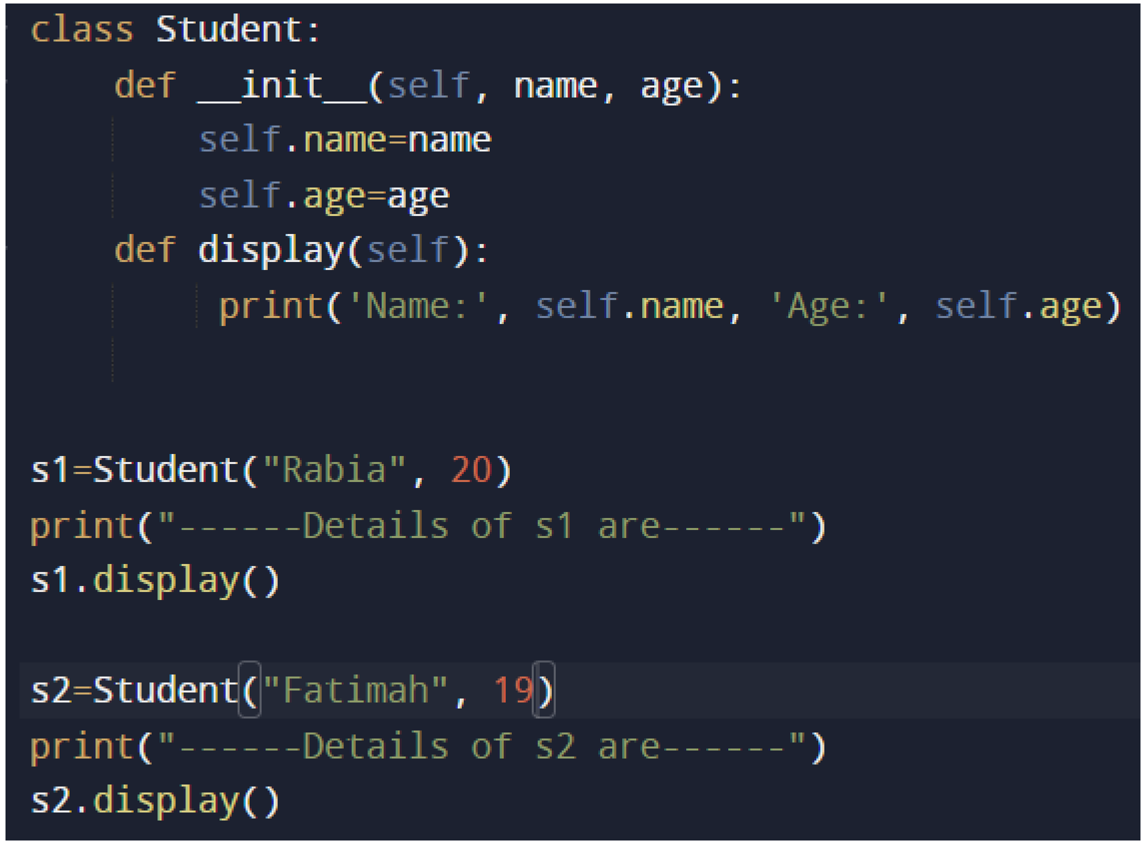
```
------Details of s1 are------
Name: ABC Age: 20
------Details of s2 are------
Name: ABC Age: 20
```

Parameterized Constructor

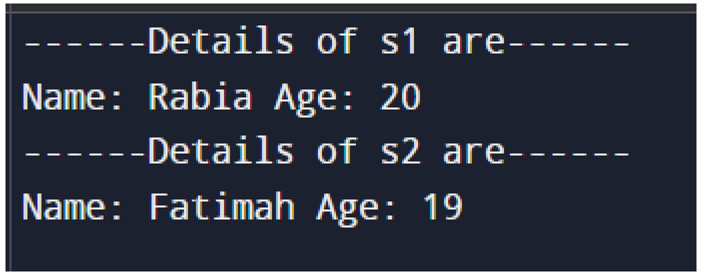
- A constructor with defined parameters or arguments is called a parameterized constructor.
- We can pass different values to each object at the time of creation using a parameterized constructor.

- The first parameter to the constructor is self which is a reference to the being constructed, and the rest of the arguments are provided by the programmer.
- A parameterized constructor can have any number of arguments.

Parameterized Constructor

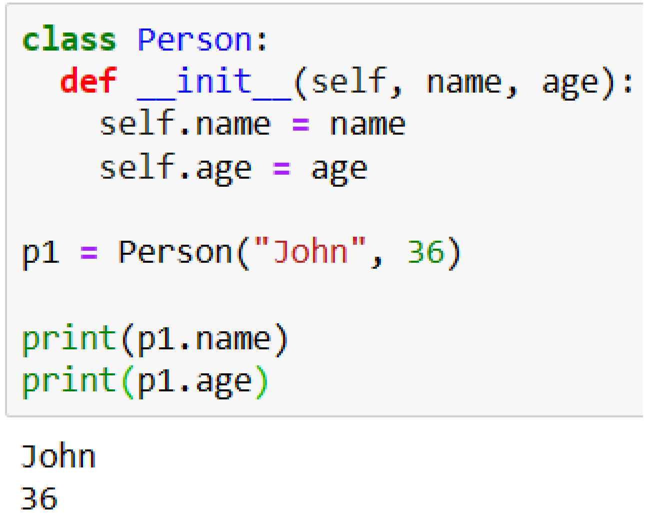


```
class Student:
    def __init__(self, name, age):
        self.name=name
        self.age=age
    def display(self):
         print('Name:', self.name, 'Age:', self.age)


s1=Student("Rabia", 20)
print("------Details of s1 are------")
s1.display()

s2=Student("Fatimah", 19)
print("------Details of s2 are------")
s2.display()
```



```
------Details of s1 are------
Name: Rabia Age: 20
------Details of s2 are------
Name: Fatimah Age: 19
```

Parameterized Constructor



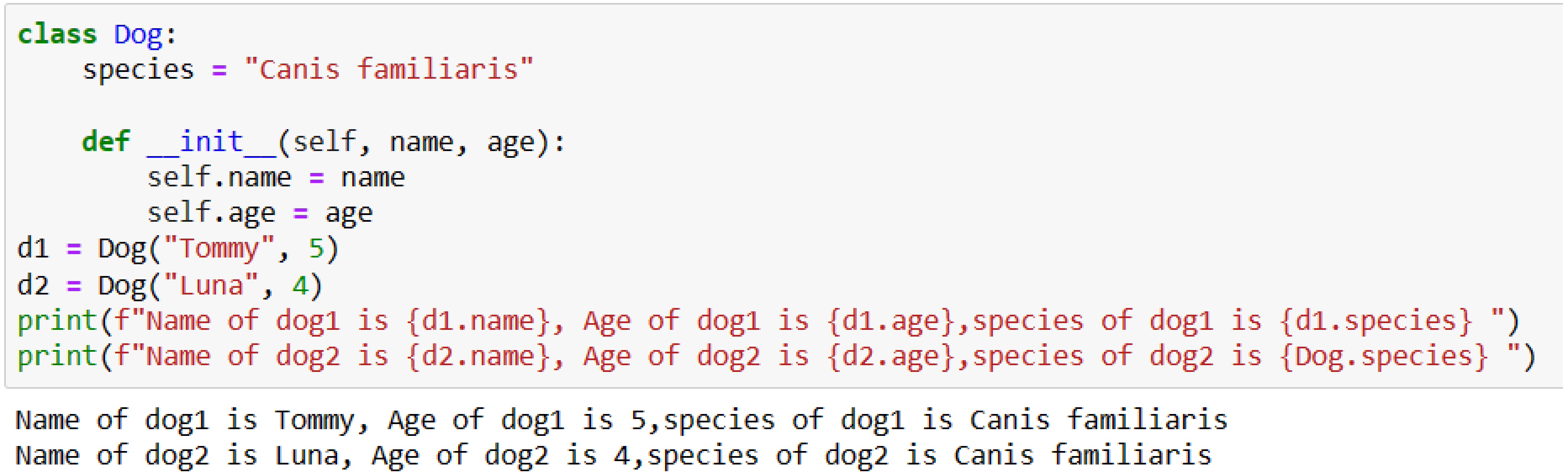
```python
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

```
John
36
```

- self.name = name **creates an attribute called name** and assigns the value of the **name parameter** to it.
- self.age = age **creates an attribute called age** and **assigns the value of the age parameter to it.**

instance attributes

- Attributes created in .__init__() are called **instance attributes**.
- An instance attribute's value is specific to a particular instance of the class.
- All Dog objects have a name and an age, but the values for the name and age attributes will vary depending on the Dog **instance**.



```python
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age
d1 = Dog("Tommy", 5)
d2 = Dog("Luna", 4)
print(f"Name of dog1 is {d1.name}, Age of dog1 is {d1.age},species of dog1 is {d1.species} ")
print(f"Name of dog2 is {d2.name}, Age of dog2 is {d2.age},species of dog2 is {Dog.species} ")
```

```
Name of dog1 is Tommy, Age of dog1 is 5,species of dog1 is Canis familiaris
Name of dog2 is Luna, Age of dog2 is 4,species of dog2 is Canis familiaris
```

class attributes

- class attributes are attributes that **have the same value for all class instances.** You can define a class attribute by assigning a value to a variable name outside of .__init__().
- You define class attributes directly beneath the first line of the class name and indent them by four spaces.
- You always need to **assign them an initial value**.
- When you create an instance of the class, then Python automatically creates and assigns class attributes to their initial values.
- You can access class attributes **either by class name or by class instance**

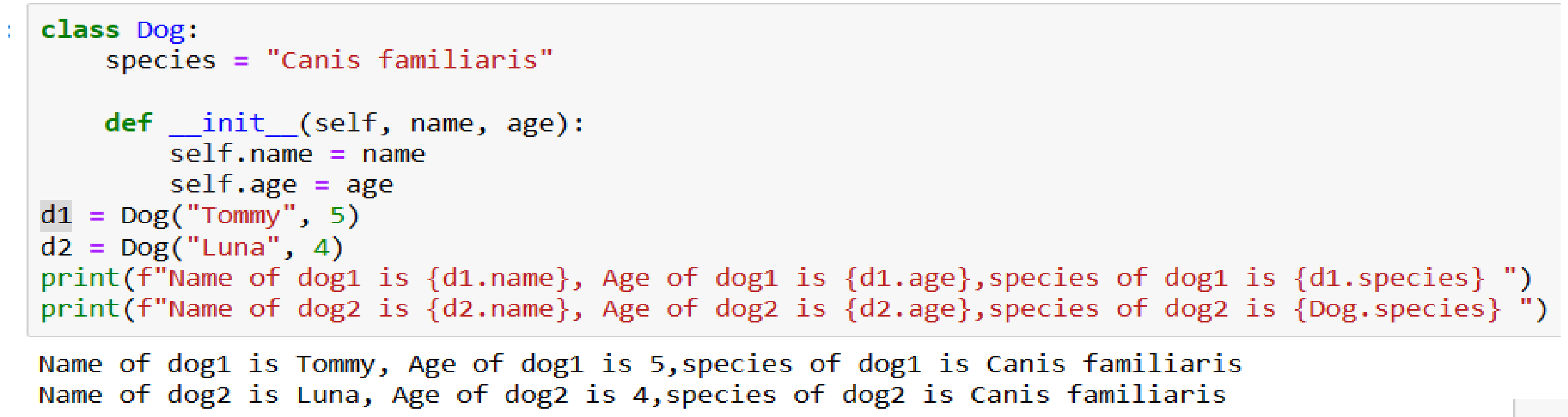
class attributes

- Use **class attributes to define properties that should have the same value for every class instance.**
- Use **instance attributes for properties that vary from one instance to another.**
- **Class attributes use the same memory location across all instances, while instance attributes have their own memory locations for each object.**

Python's __dict__ and Attribute Resolution

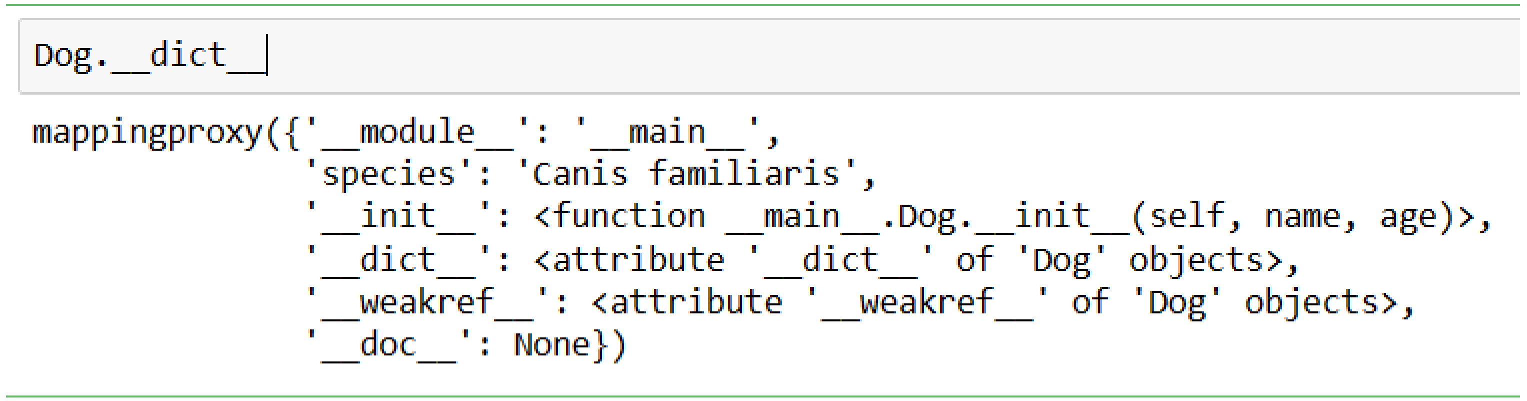
- In Python, each object has a dictionary (__dict__) that stores its attributes.
- For **instance attributes,** the __dict__ contains the names and values of all instance-specific attributes.
- On the other hand, class attributes are stored in the **class's __dict__.**
- When accessing an attribute, **Python first checks if it exists in the instance's __dict__.** If not found, **it then looks into the class's __dict__** and retrieves the attribute.

Python's __dict__ and Attribute Resolution



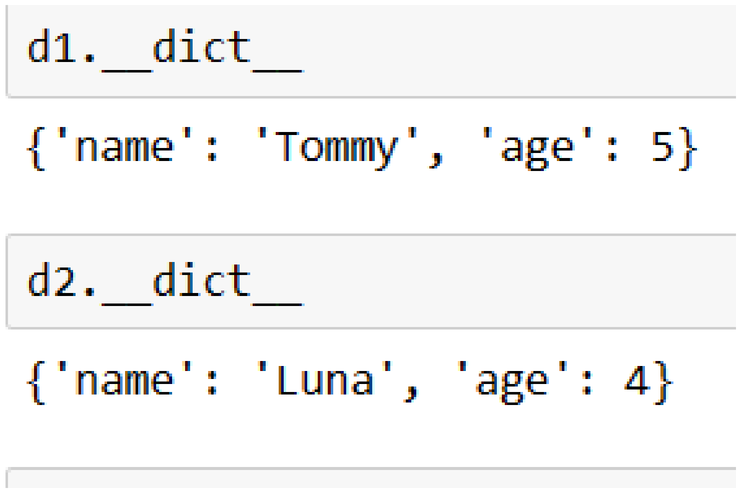
```
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age
d1 = Dog("Tommy", 5)
d2 = Dog("Luna", 4)
print(f"Name of dog1 is {d1.name}, Age of dog1 is {d1.age},species of dog1 is {d1.species} ")
print(f"Name of dog2 is {d2.name}, Age of dog2 is {d2.age},species of dog2 is {Dog.species} ")
```

```
Name of dog1 is Tommy, Age of dog1 is 5,species of dog1 is Canis familiaris
Name of dog2 is Luna, Age of dog2 is 4,species of dog2 is Canis familiaris
```



```
Dog.__dict__
```

```
mappingproxy({'__module__': '__main__',
              'species': 'Canis familiaris',
              '__init__': <function __main__.Dog.__init__(self, name, age)>,
              '__dict__': <attribute '__dict__' of 'Dog' objects>,
              '__weakref__': <attribute '__weakref__' of 'Dog' objects>,
              '__doc__': None})
```



```
d1.__dict__
```

```
{'name': 'Tommy', 'age': 5}
```

```
d2.__dict__
```

```
{'name': 'Luna', 'age': 4}
```

Destructor

- Destructor is a special method that is called when an object gets destroyed.
- Python has **a garbage collector** that handles memory management automatically. For example, it cleans up the memory when an object goes out of scope.
- But it’s not just memory that has to be freed when an object is destroyed. **We must release or close the other resources object were using, such as open files, database connections, cleaning up the buffer or cache.** To perform all those cleanup tasks we use destructor in Python.

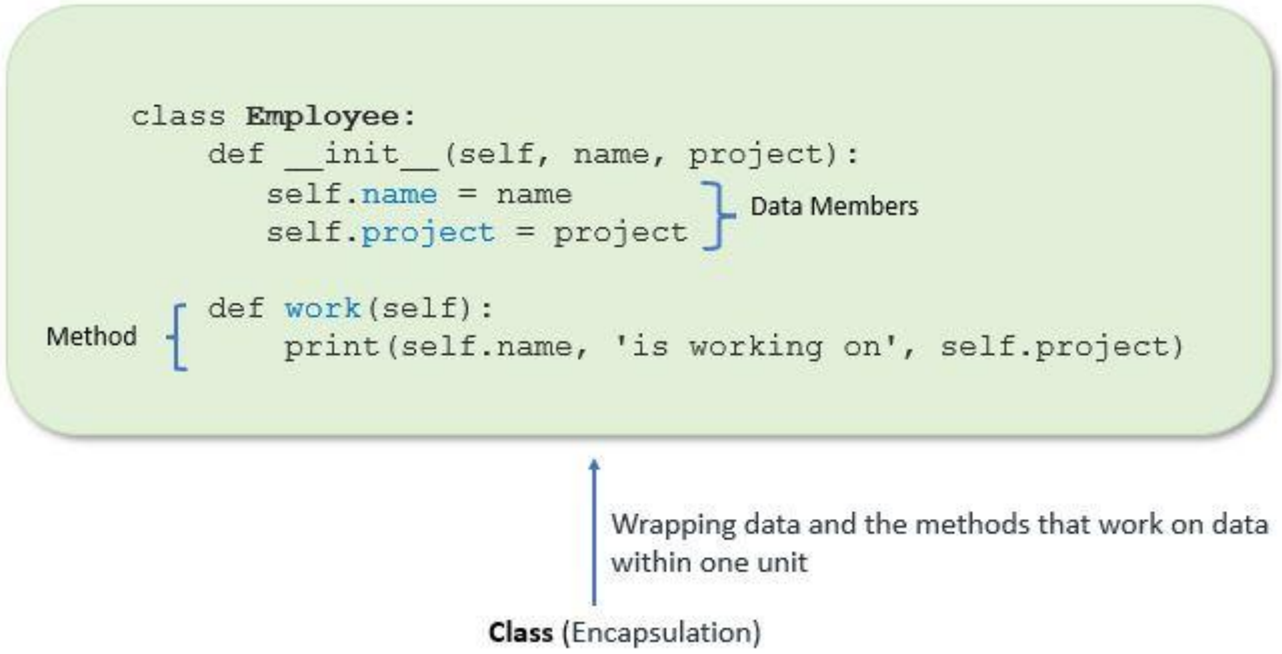
Object Oriented Programming Principles

- 4 major principles make a language Object Oriented.
  - Encapsulation
  - Data Abstraction
  - Polymorphism
  - Inheritance

Encapsulation

- Encapsulation in Python describes the concept of **bundling data and methods** within a single unit.
- So, for example, when you create a class, it means you are implementing encapsulation.
- A class is an example of encapsulation as it binds all the data members and methods into a single unit.

Encapsulation



```
class Employee:
    def __init__(self, name, project):
        self.name = name
        self.project = project

    def work(self):
        print(self.name, 'is working on', self.project)
```

Data Members
Method
Wrapping data and the methods that work on data within one unit
Class (Encapsulation)

Information Hiding in Python

- Using encapsulation, we can **hide an object’s internal representation from the outside**. This is called information hiding.
- encapsulation **allows us to restrict accessing variables and methods directly** and **prevent accidental data modification by creating private data members and methods** within a class.
- Encapsulation is a way **to restrict access to methods and variables from outside of class.** Whenever we are working with the class and dealing with sensitive data, providing access to all variables used within the class is not a good choice.

Encapsulation

- Encapsulation can be achieved by declaring the data members and methods of a class either as private or protected.

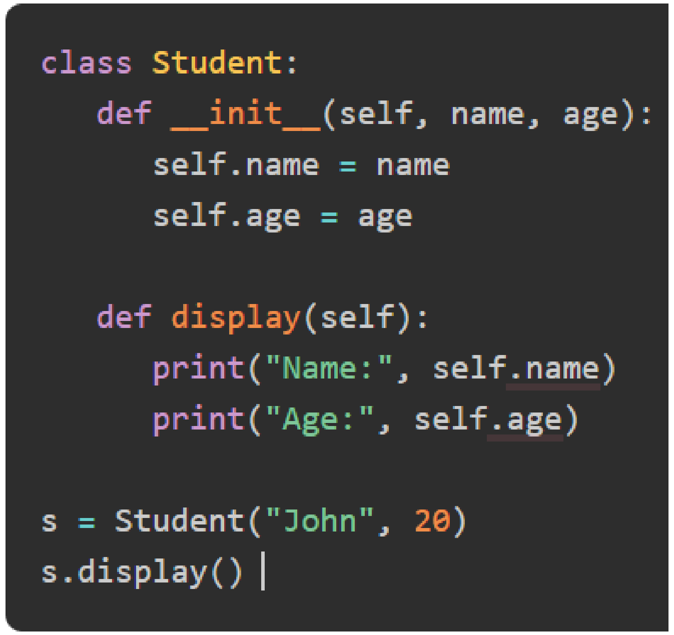
Access modifiers

- Access modifiers are used by object-oriented programming languages like C++, java, python, etc. **to restrict the access of the class member variable and methods from outside the class.**
- Python supports three types of access modifiers which are
  - Public -Accessible anywhere from outside of class.
  - Private-Accessible within the class
  - Protected-Accessible within the class and its sub-classes

Public Access Modifier

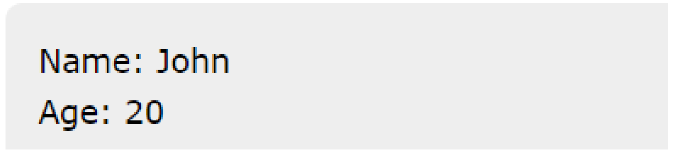
- All members i.e. variables and methods in a Python class are public by default.
- Any member can be accessed from anywhere i.e. outside or inside the class environment.
- No public keyword is required to make the class or methods and properties public.

Public Access Modifier



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

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

s = Student("John", 20)
s.display()
```

Output

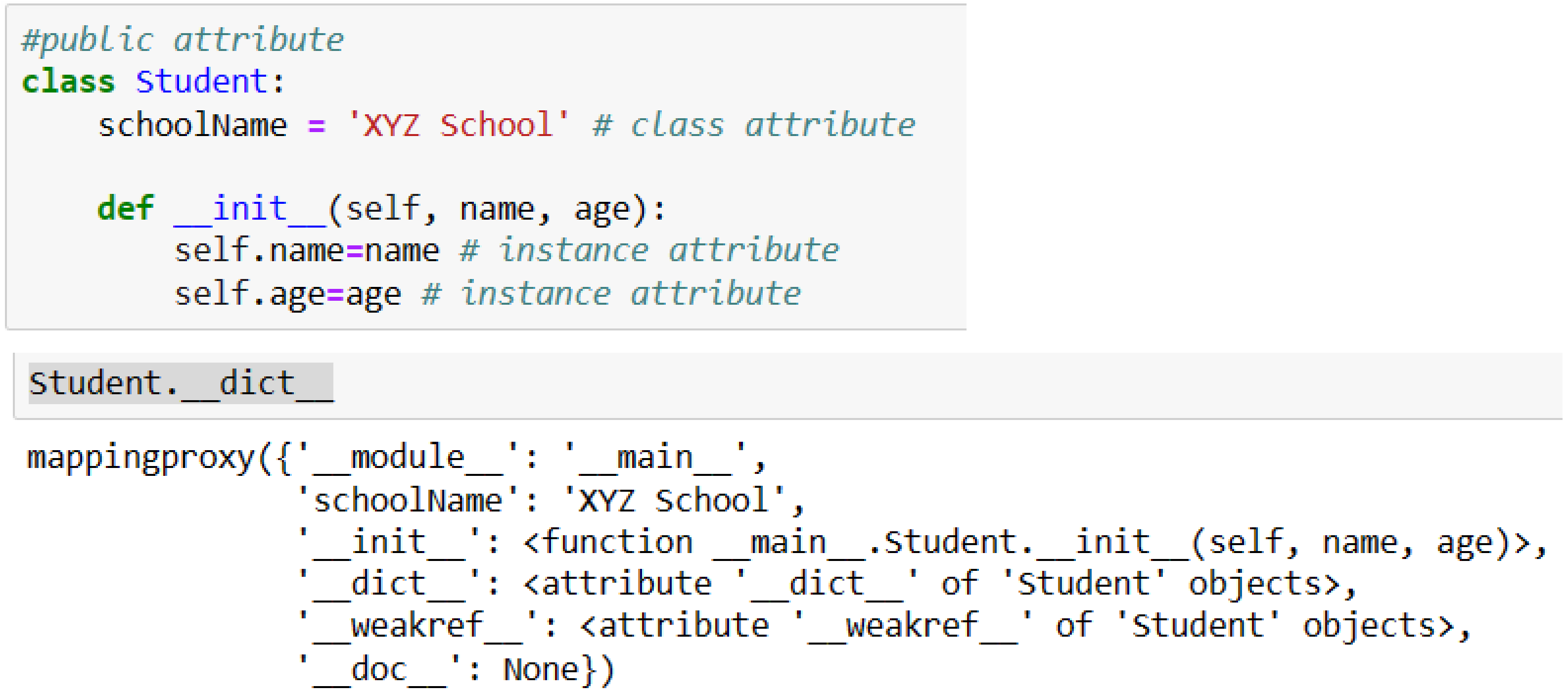


```
Name: John
Age: 20
```

The student class has two member variables, name and age and a method display which prints the member variable values.

Both these variables and the methods are public as no specific keyword is assigned to them.

Public Access Modifier

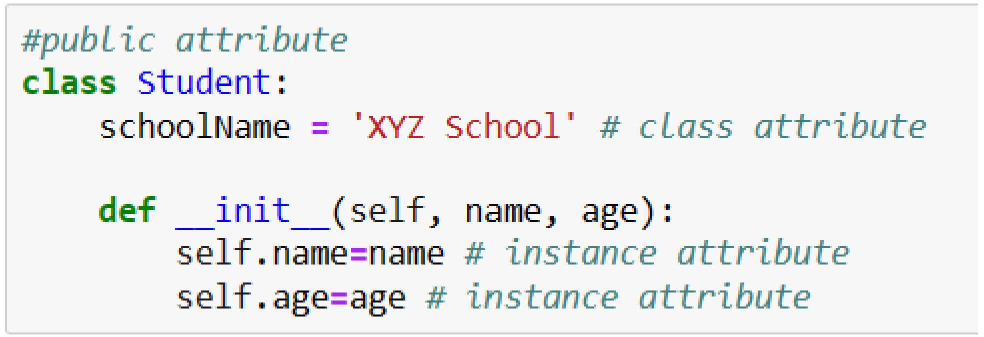


```python
#public attribute
class Student:
    schoolName = 'XYZ School' # class attribute

    def __init__(self, name, age):
        self.name=name # instance attribute
        self.age=age # instance attribute
```

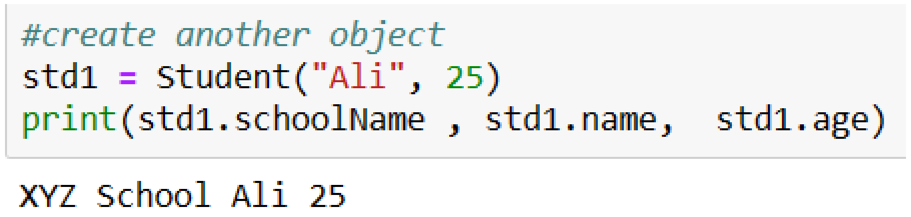
```python
Student.__dict__
```

```
mappingproxy({'__module__': '__main__',
              'schoolName': 'XYZ School',
              '__init__': <function __main__.Student.__init__(self, name, age)>,
              '__dict__': <attribute '__dict__' of 'Student' objects>,
              '__weakref__': <attribute '__weakref__' of 'Student' objects>,
              '__doc__': None})
```

Public Access Modifier

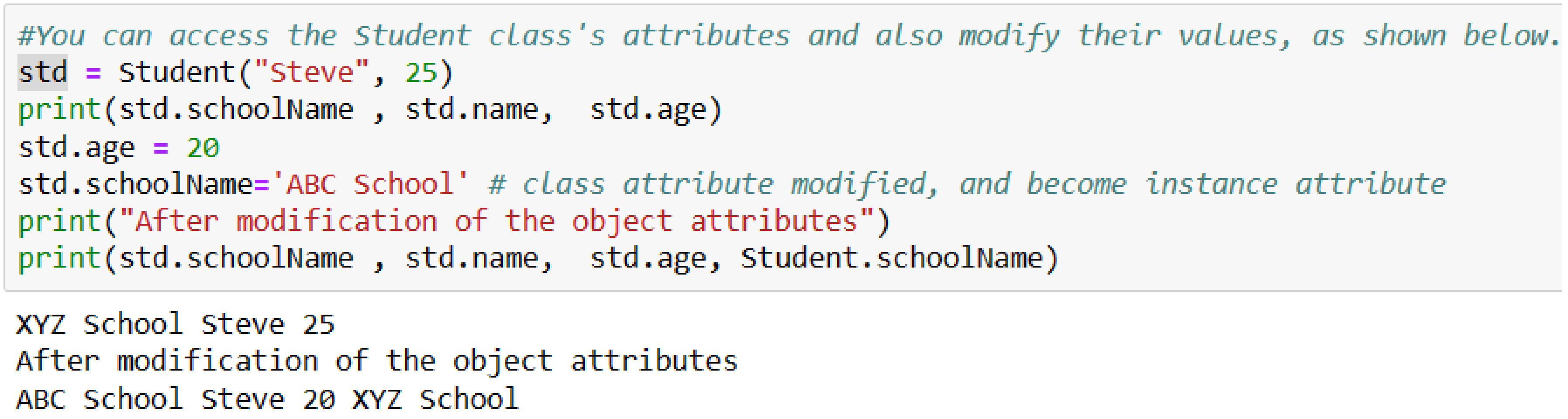


```python
#public attribute
class Student:
    schoolName = 'XYZ School' # class attribute

    def __init__(self, name, age):
        self.name=name # instance attribute
        self.age=age # instance attribute
```



```python
#create another object
std1 = Student("Ali", 25)
print(std1.schoolName , std1.name,  std1.age)
```

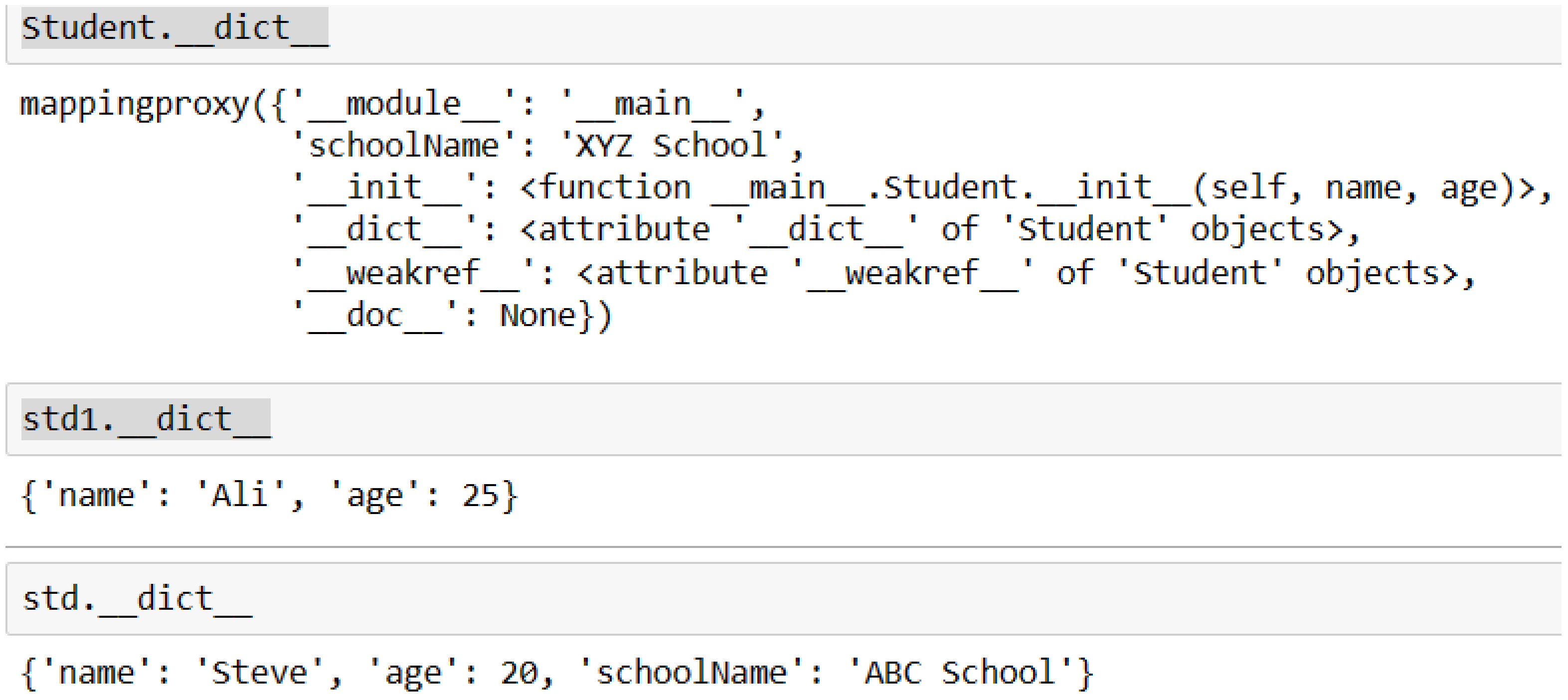
```
XYZ School Ali 25
```



```python
#You can access the Student class's attributes and also modify their values, as shown below.
std = Student("Steve", 25)
print(std.schoolName , std.name,  std.age)
std.age = 20
std.schoolName='ABC School' # class attribute modified, and become instance attribute
print("After modification of the object attributes")
print(std.schoolName , std.name,  std.age, Student.schoolName)
```

```
XYZ School Steve 25
After modification of the object attributes
ABC School Steve 20 XYZ School
```

Public Access Modifier



```
Student.__dict__
```

```
mappingproxy({'__module__': '__main__',
              'schoolName': 'XYZ School',
              '__init__': <function __main__.Student.__init__(self, name, age)>,
              '__dict__': <attribute '__dict__' of 'Student' objects>,
              '__weakref__': <attribute '__weakref__' of 'Student' objects>,
              '__doc__': None})
```

```
std1.__dict__
```

```
{'name': 'Ali', 'age': 25}
```

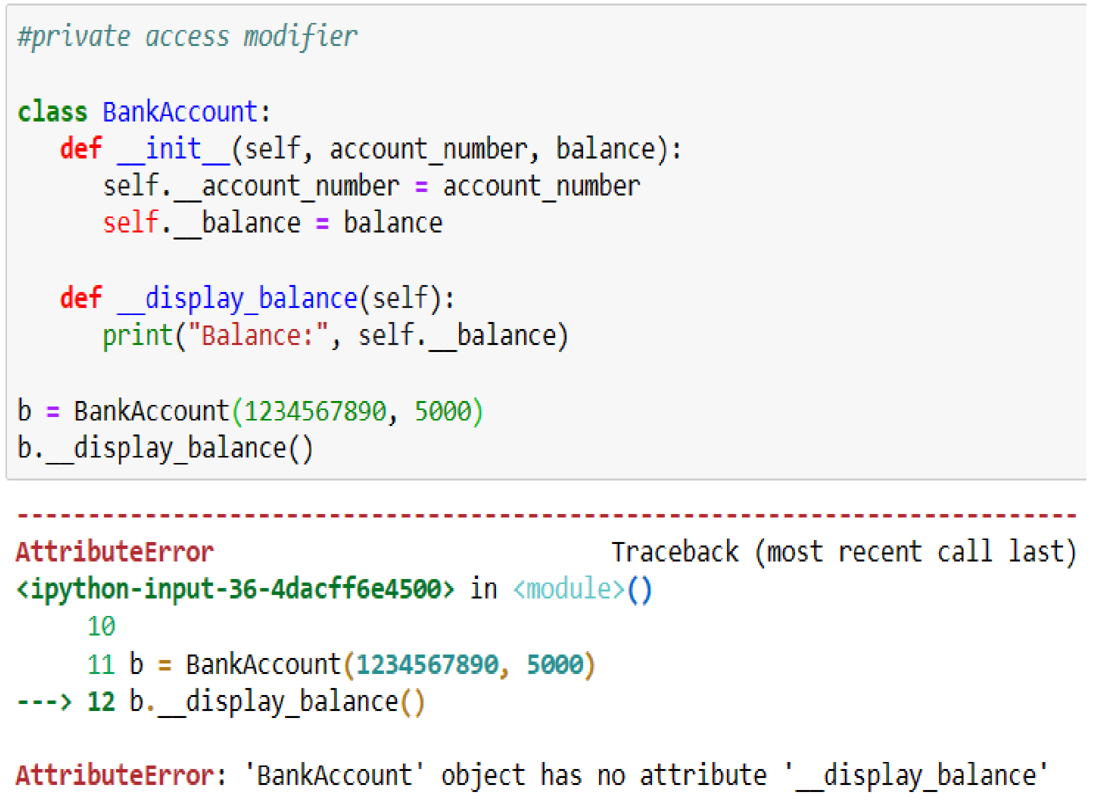
```
std.__dict__
```

```
{'name': 'Steve', 'age': 20, 'schoolName': 'ABC School'}
```

Private Access Modifier

- Class properties and methods with private access modifier can only be **accessed within the class where they are defined and cannot be accessed outside the class.**
- **The private access modifier is the most secure access modifier.**
- In Python private properties and methods are declared by adding a prefix with **two underscores**('__') before their declaration.

Private Access Modifier



```
#private access modifier

class BankAccount:
   def __init__(self, account_number, balance):
      self.__account_number = account_number
      self.__balance = balance

   def __display_balance(self):
      print("Balance:", self.__balance)

b = BankAccount(1234567890, 5000)
b.__display_balance()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-36-4dacff6e4500> in <module>()
     10
     11 b = BankAccount(1234567890, 5000)
---> 12 b.__display_balance()

AttributeError: 'BankAccount' object has no attribute '__display_balance'
```

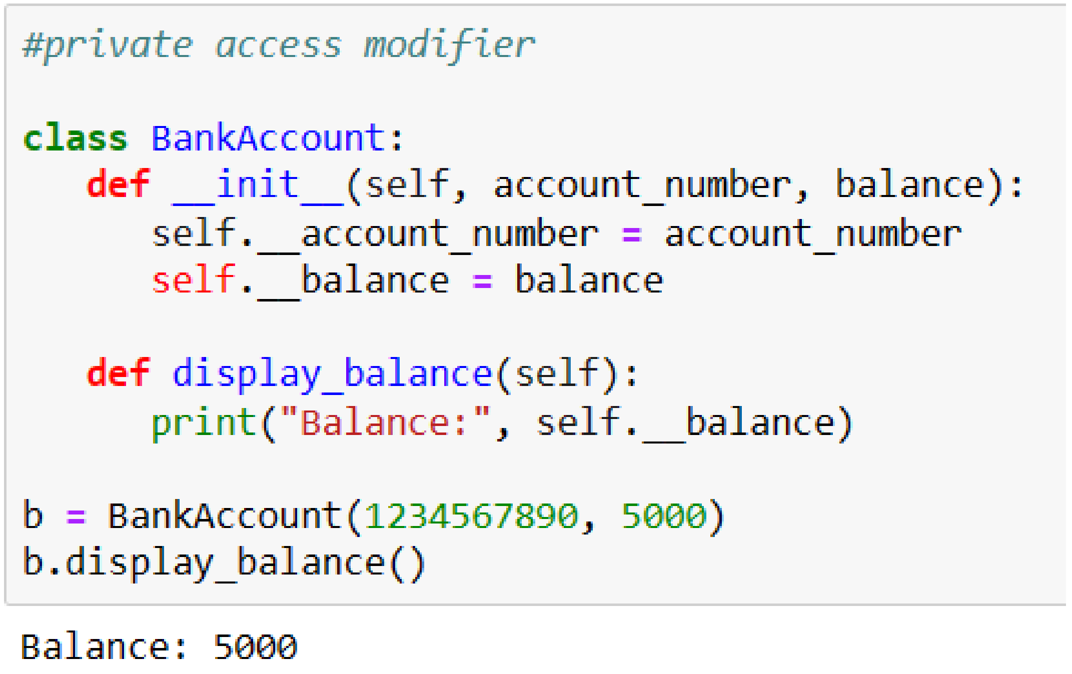
- The Class BankAccount is being declared with two private variables i.e account_number and balance and a private method display_balance which prints the balance of the bank account.
- As both the properties and method are private so while accessing them from outside the class it raises Attribute error.

How to access private members?

- We can access private members from outside of a class using the following two approaches

- Create public method to access private members
- Use name mangling

Public method to access private members

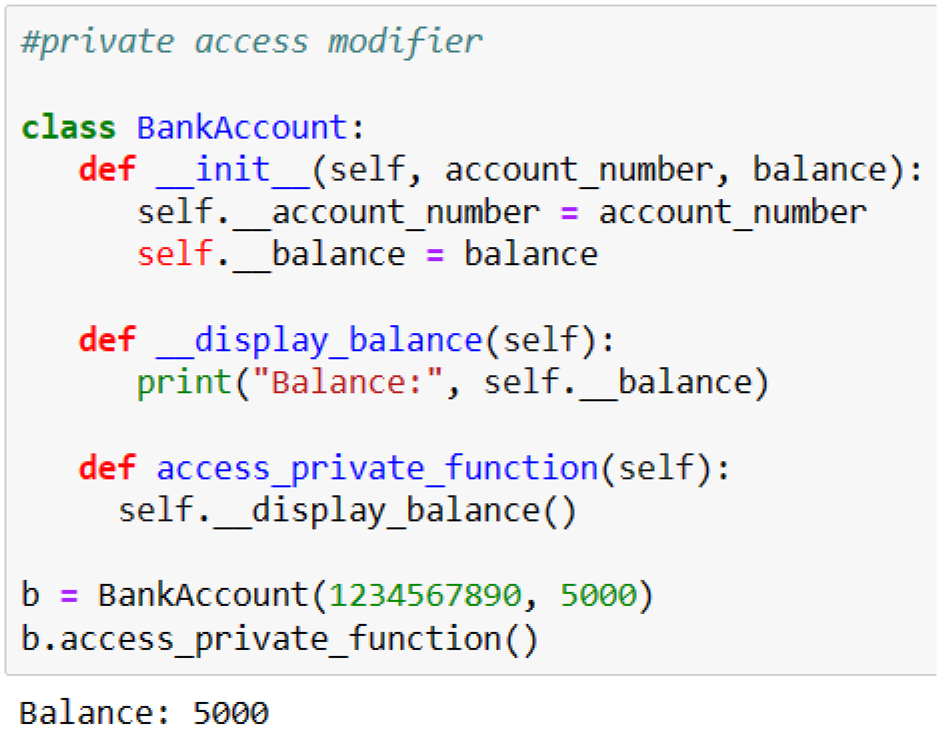


```
#private access modifier

class BankAccount:
   def __init__(self, account_number, balance):
      self.__account_number = account_number
      self.__balance = balance

   def display_balance(self):
      print("Balance:", self.__balance)

b = BankAccount(1234567890, 5000)
b.display_balance()
```

```
Balance: 5000
```

- The Class BankAccount is being declared with two private variables i.e account_number and balance and a public method display_balance which prints the balance of the bank account.
- As both the properties are private ,so we can access them from outside the class using class public function.

Public method to access private members



```
#private access modifier

class BankAccount:
   def __init__(self, account_number, balance):
      self.__account_number = account_number
      self.__balance = balance

   def __display_balance(self):
      print("Balance:", self.__balance)

   def access_private_function(self):
     self.__display_balance()

b = BankAccount(1234567890, 5000)
b.access_private_function()
```

```
Balance: 5000
```

- The Class BankAccount is being declared with two private variables i.e account_number and balance and a private method display_balance which prints the balance of the bank account. And a public method access_private_function
- Here we have accessed private method using public method

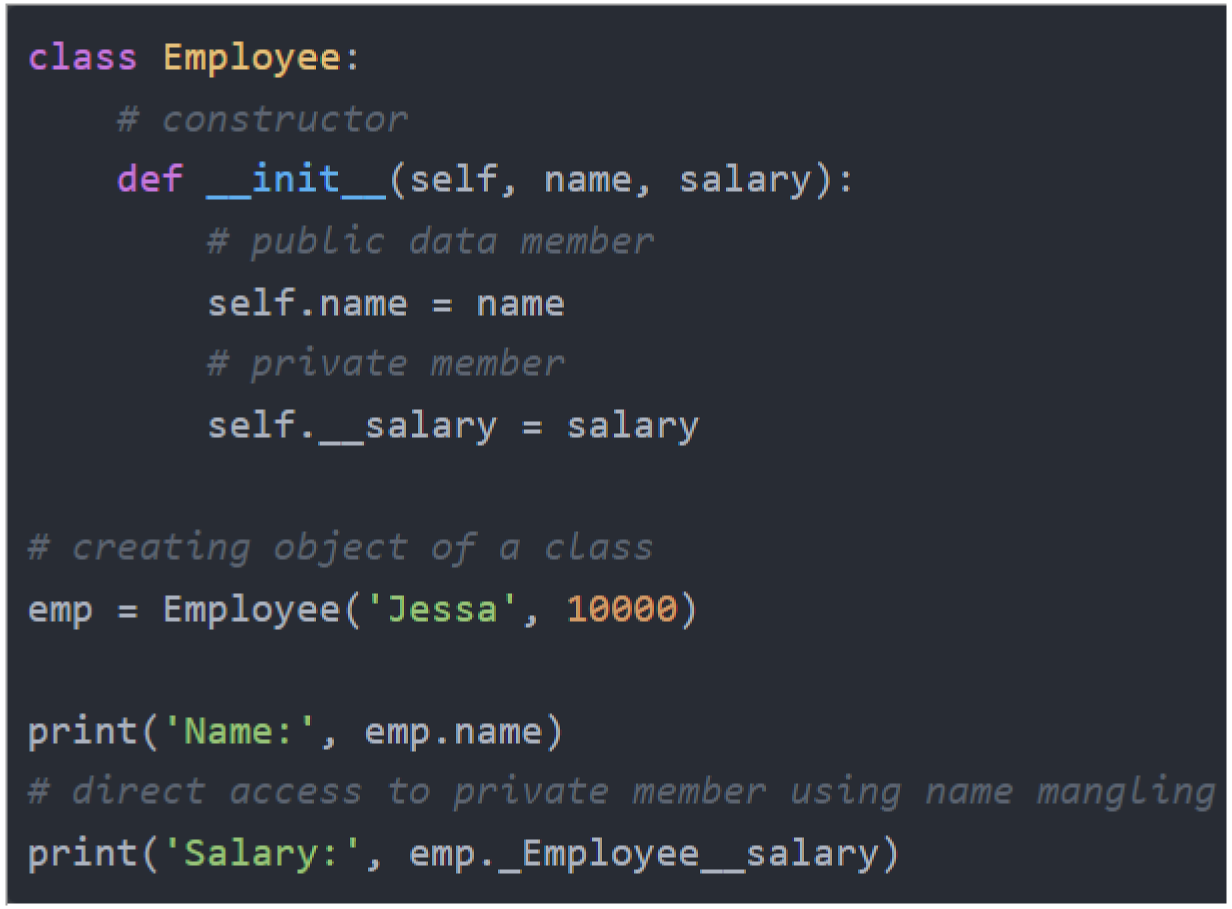
Name Mangling to access private members

- We can directly access private and protected variables from outside of a class through name mangling.
- The name mangling is created on an **identifier by adding two leading underscores and one trailing underscore**, like

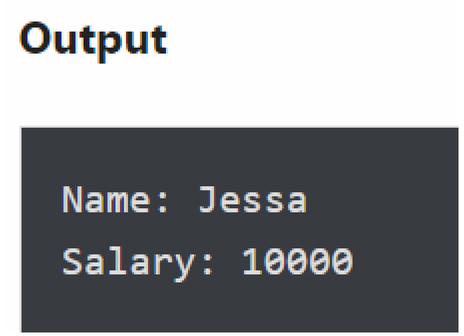
_classname__dataMember

- where classname is the current class, and data member is the private variable name.

Name Mangling to access private members



```python
class Employee:
    # constructor
    def __init__(self, name, salary):
        # public data member
        self.name = name
        # private member
        self.__salary = salary

# creating object of a class
emp = Employee('Jessa', 10000)

print('Name:', emp.name)
# direct access to private member using name mangling
print('Salary:', emp._Employee__salary)
```



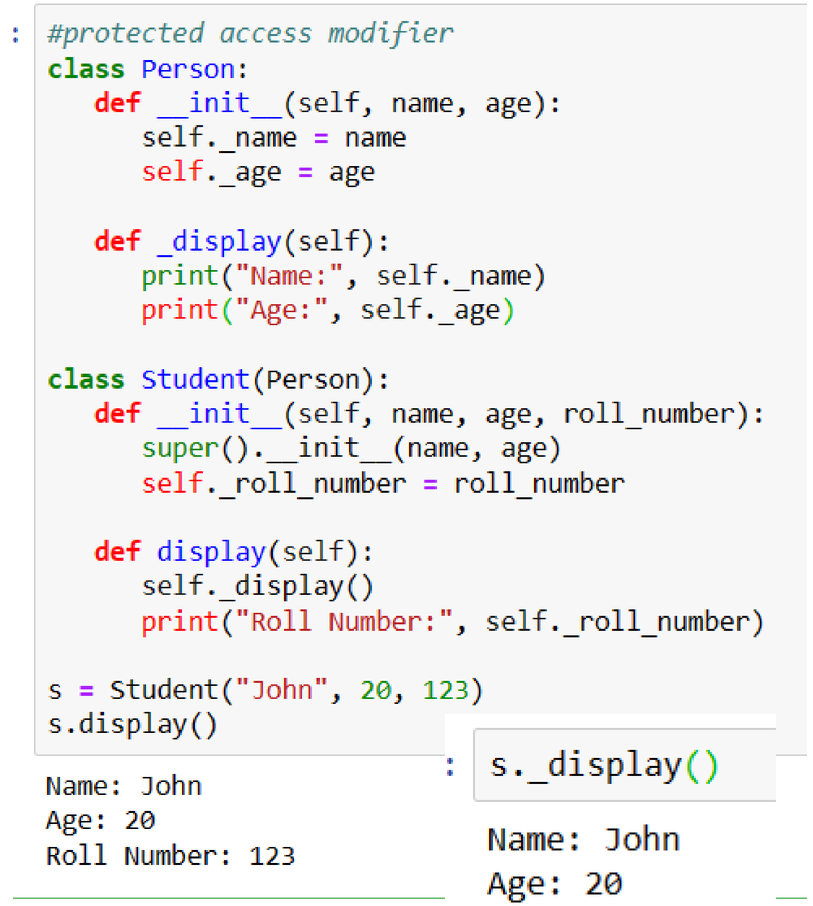
Output

```
Name: Jessa
Salary: 10000
```

Protected Access Modifier

- Protected members of a class are accessible from within the class and are also available to its sub-classes.
- No other environment is permitted access to it.
- This enables specific resources of the parent class to be inherited by the child class.
- Protected data members are used when you implement inheritance and want to allow data members access to only child classes.
- In python, protected members and methods are declared using single underscore(‘_’) as prefix before their names.

Protected Access Modifier



```
#protected access modifier
class Person:
    def __init__(self, name, age):
        self._name = name
        self._age = age

    def _display(self):
        print("Name:", self._name)
        print("Age:", self._age)

class Student(Person):
    def __init__(self, name, age, roll_number):
        super().__init__(name, age)
        self._roll_number = roll_number

    def display(self):
        self._display()
        print("Roll Number:", self._roll_number)

s = Student("John", 20, 123)
s.display()
```

```
Name: John
Age: 20
Roll Number: 123
```

```
s._display()
```

```
Name: John
Age: 20
```

The Person class has two protected properties i.e _name and _age and a protected method _display that displays the values of the properties of the person class.

The student class is inherited from Person class with an additional property i.e _roll_number which is also protected and a public method display that calls the _display method of the parent class i.e Person class

By creating an instance of the Student class we can call the display method from outside the class as the display method is public which calls the protected _display method of Person class.

Getters and Setters in Python

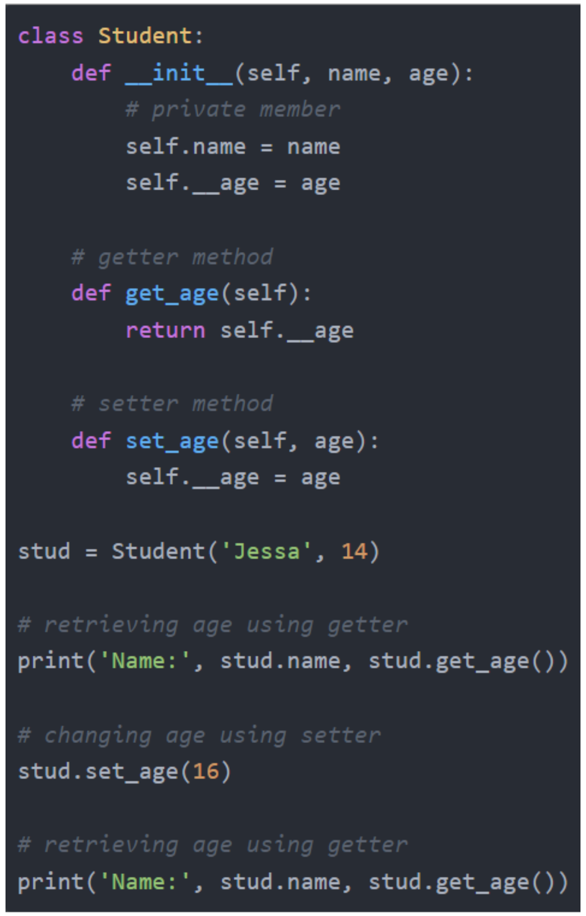
- To implement proper encapsulation in Python, we need to use setters and getters.
- The primary purpose of using getters and setters in object-oriented programs is to ensure data encapsulation.
- Use the **getter method to access data members** and **the setter methods to modify the data members.**

Getters and Setters in Python

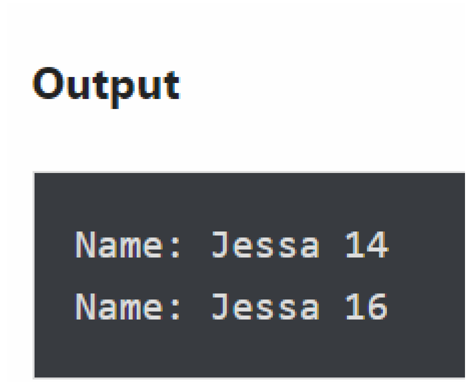
The getters and setters methods are often used when:

- When we want to avoid direct access to private variables
- To add validation logic for setting a value

Getters and Setters in Python



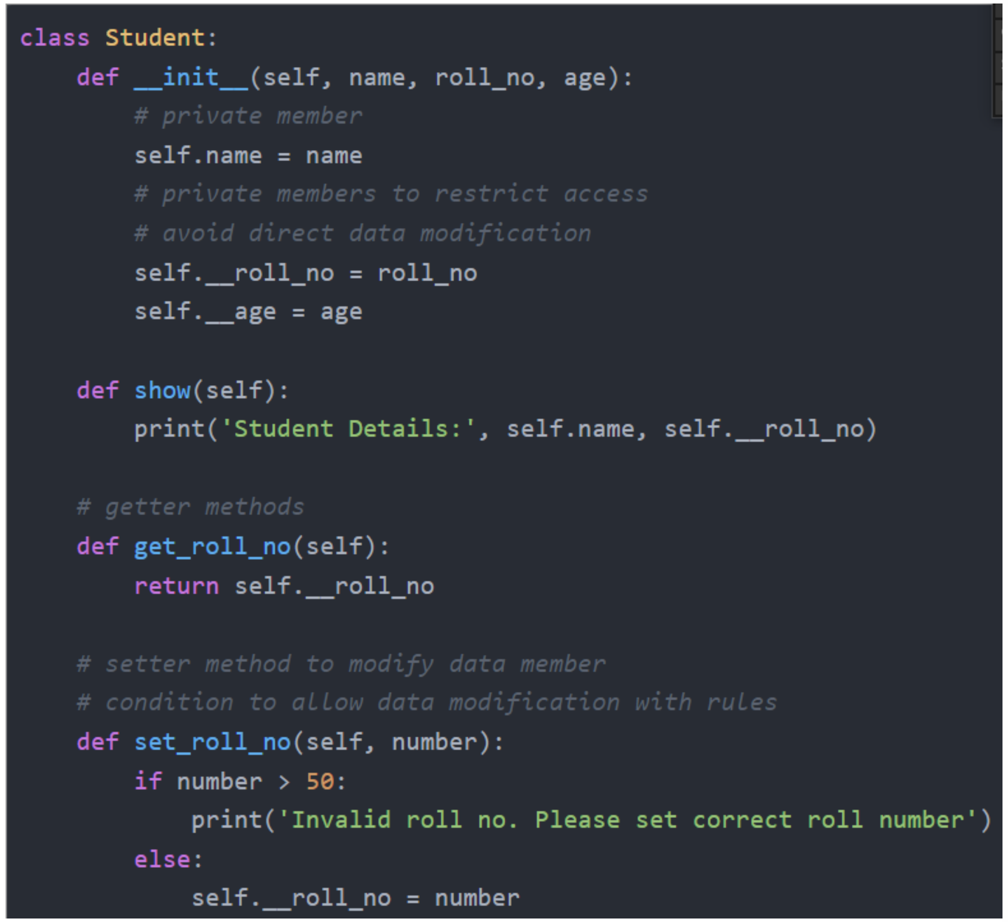
```python
class Student:
    def __init__(self, name, age):
        # private member
        self.name = name
        self.__age = age

    # getter method
    def get_age(self):
        return self.__age

    # setter method
    def set_age(self, age):
        self.__age = age

stud = Student('Jessa', 14)

# retrieving age using getter
print('Name:', stud.name, stud.get_age())

# changing age using setter
stud.set_age(16)

# retrieving age using getter
print('Name:', stud.name, stud.get_age())
```



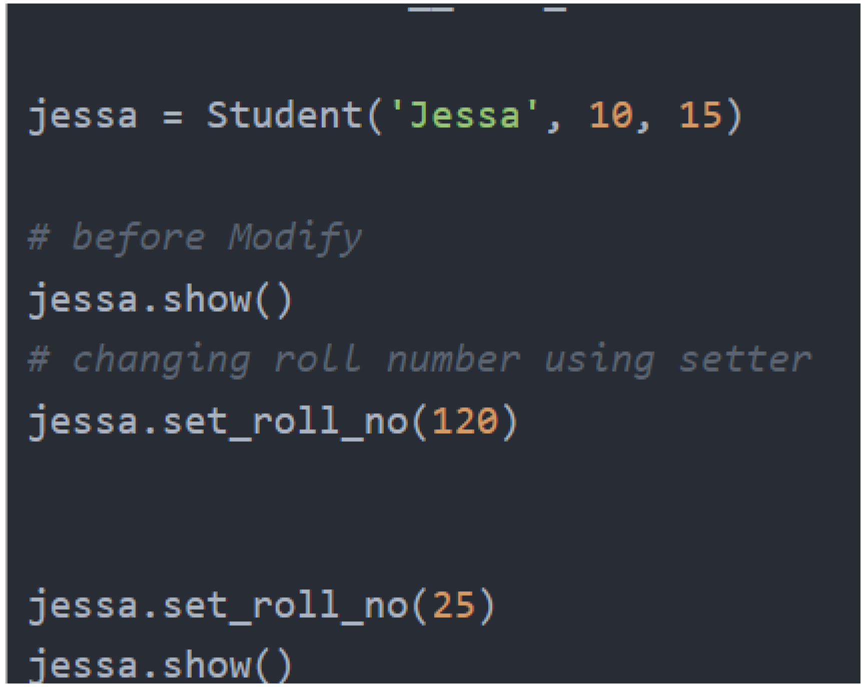
Output

```
Name: Jessa 14
Name: Jessa 16
```

Information Hiding and conditional logic for setting an object attributes

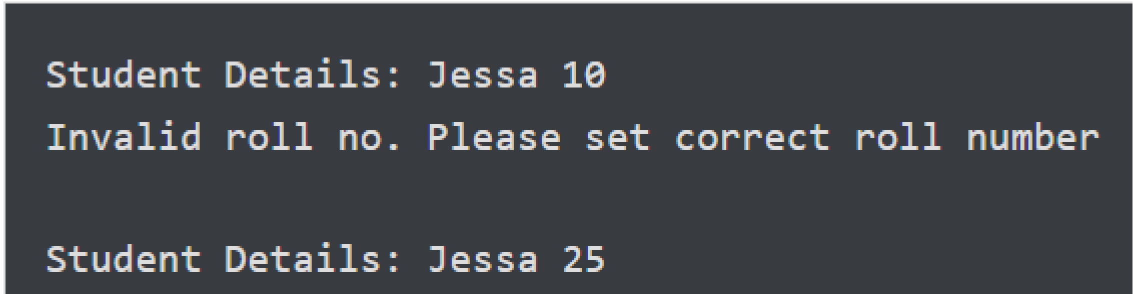


```python
class Student:
    def __init__(self, name, roll_no, age):
        # private member
        self.name = name
        # private members to restrict access
        # avoid direct data modification
        self.__roll_no = roll_no
        self.__age = age

    def show(self):
        print('Student Details:', self.name, self.__roll_no)

    # getter methods
    def get_roll_no(self):
        return self.__roll_no

    # setter method to modify data member
    # condition to allow data modification with rules
    def set_roll_no(self, number):
        if number > 50:
            print('Invalid roll no. Please set correct roll number')
        else:
            self.__roll_no = number
```



```python
jessa = Student('Jessa', 10, 15)

# before Modify
jessa.show()
# changing roll number using setter
jessa.set_roll_no(120)


jessa.set_roll_no(25)
jessa.show()
```

Output:



```
Student Details: Jessa 10
Invalid roll no. Please set correct roll number

Student Details: Jessa 25
```

Advantages of Encapsulation

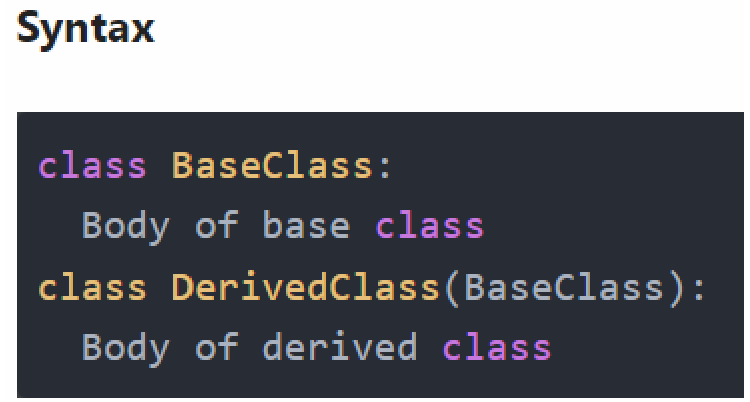
- **Security:** The main advantage of using encapsulation is the security of the data. Encapsulation protects an object from unauthorized access. It allows private and protected access levels to prevent accidental data modification.
- **Data Hiding:** The user would not be knowing what is going on behind the scene. They would only be knowing that to modify a data member, call the setter method. To read a data member, call the getter method. What these setter and getter methods are doing is hidden from them.

Advantages of Encapsulation

- **Simplicity:** It simplifies the maintenance of the application by keeping classes separated and preventing them from tightly coupling with each other.
- **Aesthetics:** Bundling data and methods within a class makes code more readable and maintainable

Inheritance in Python

- The process of **inheriting the properties of the parent class into a child class** is called inheritance.
- The existing class is called a **base class** or parent class and the new class is called a **subclass** or child class or derived class.
- the main purpose of inheritance is the **reusability** of code because we can use the existing class to create a new class instead of creating it from scratch.


Syntax

```
class BaseClass:
  Body of base class
class DerivedClass(BaseClass):
  Body of derived class
```

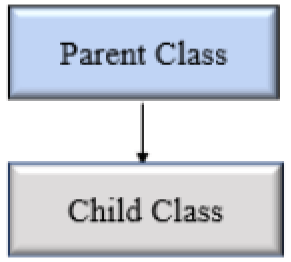
Types Of Inheritance

In Python, based upon the number of child and parent classes involved, there are five types of inheritance.

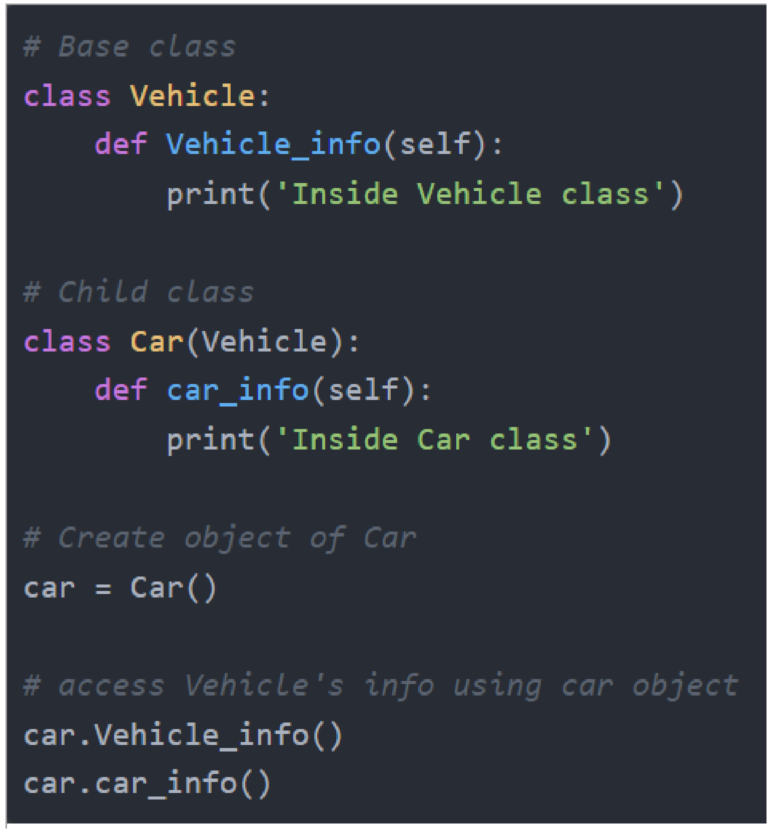
- Single inheritance
- Multiple Inheritance
- Multilevel inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

Single Inheritance

- In single inheritance, a child class inherits from a single-parent class.

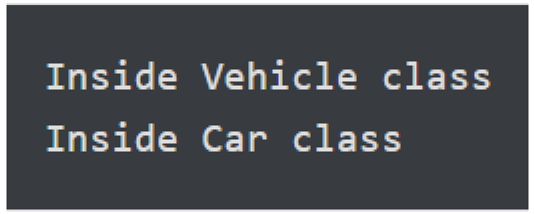


Single Inheritance



```python
# Base class
class Vehicle:
    def Vehicle_info(self):
        print('Inside Vehicle class')

# Child class
class Car(Vehicle):
    def car_info(self):
        print('Inside Car class')

# Create object of Car
car = Car()

# access Vehicle's info using car object
car.Vehicle_info()
car.car_info()
```

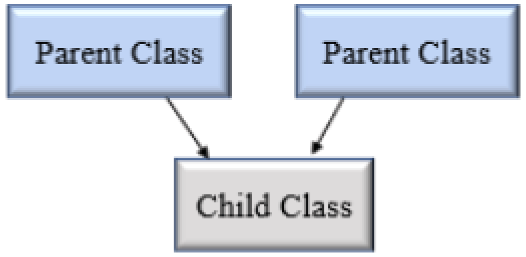
Output



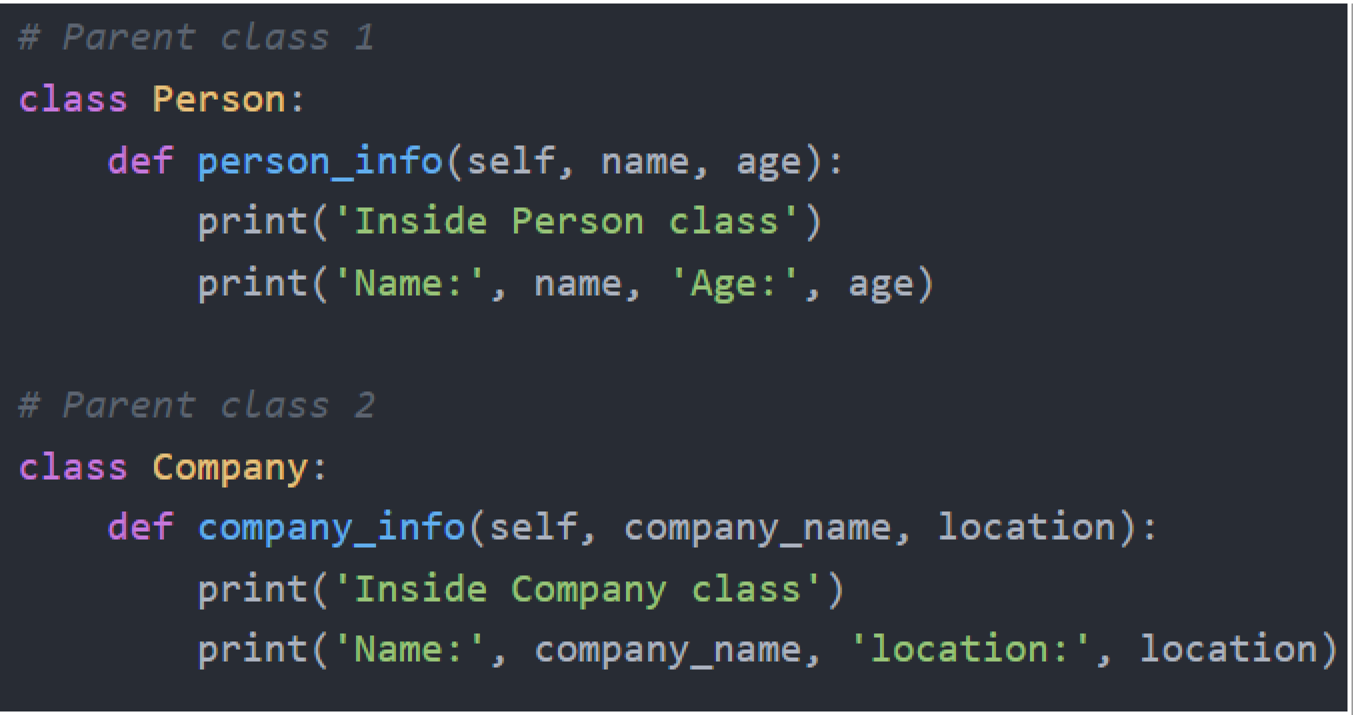
```
Inside Vehicle class
Inside Car class
```

Multiple Inheritance

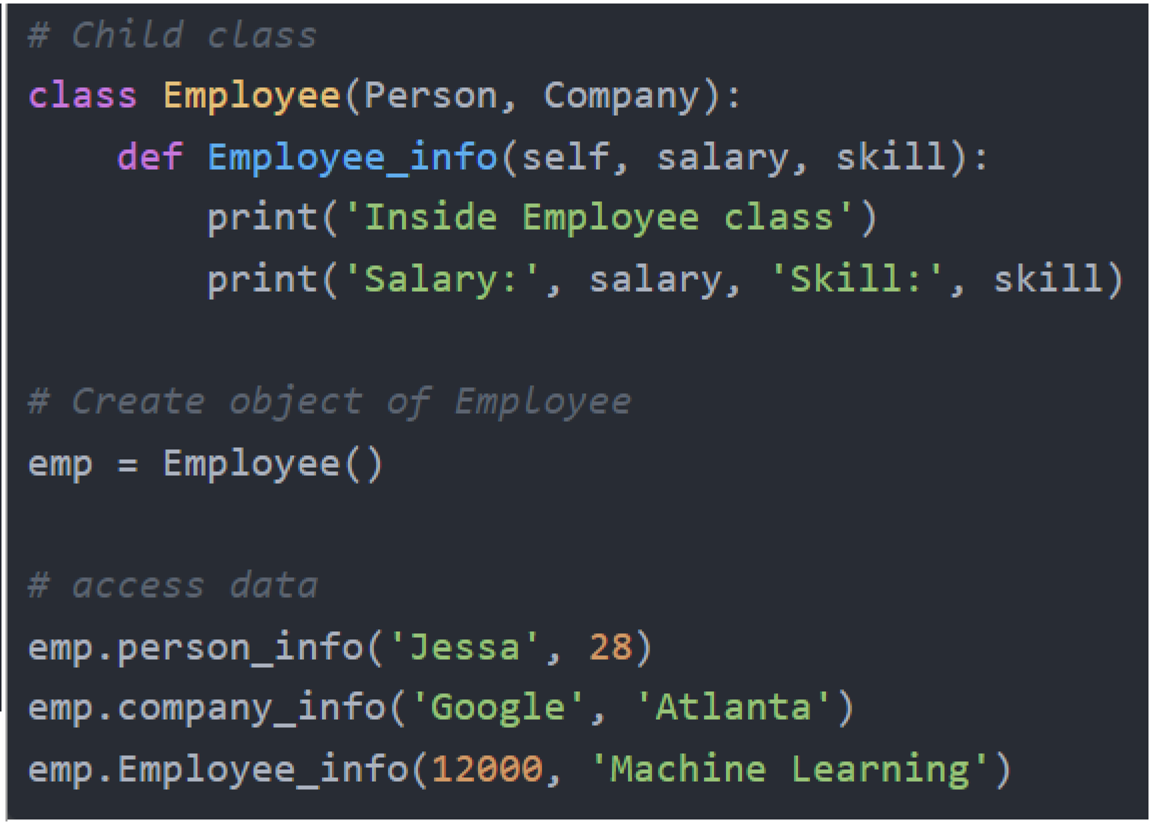
- In multiple inheritance, one child class can inherit from multiple parent classes.



Multiple Inheritance

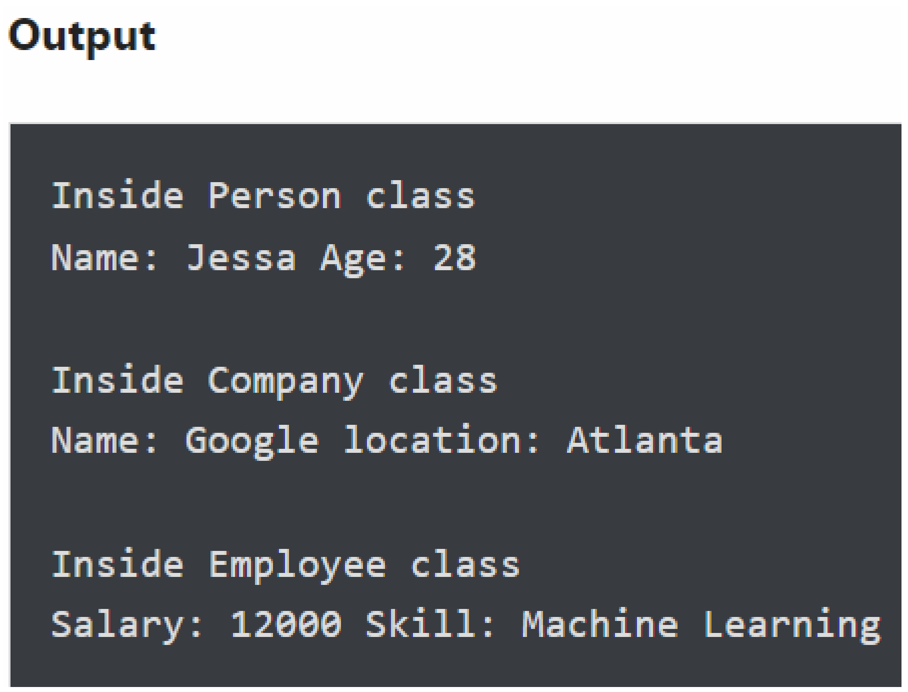


```python
# Parent class 1
class Person:
    def person_info(self, name, age):
        print('Inside Person class')
        print('Name:', name, 'Age:', age)

# Parent class 2
class Company:
    def company_info(self, company_name, location):
        print('Inside Company class')
        print('Name:', company_name, 'location:', location)
```

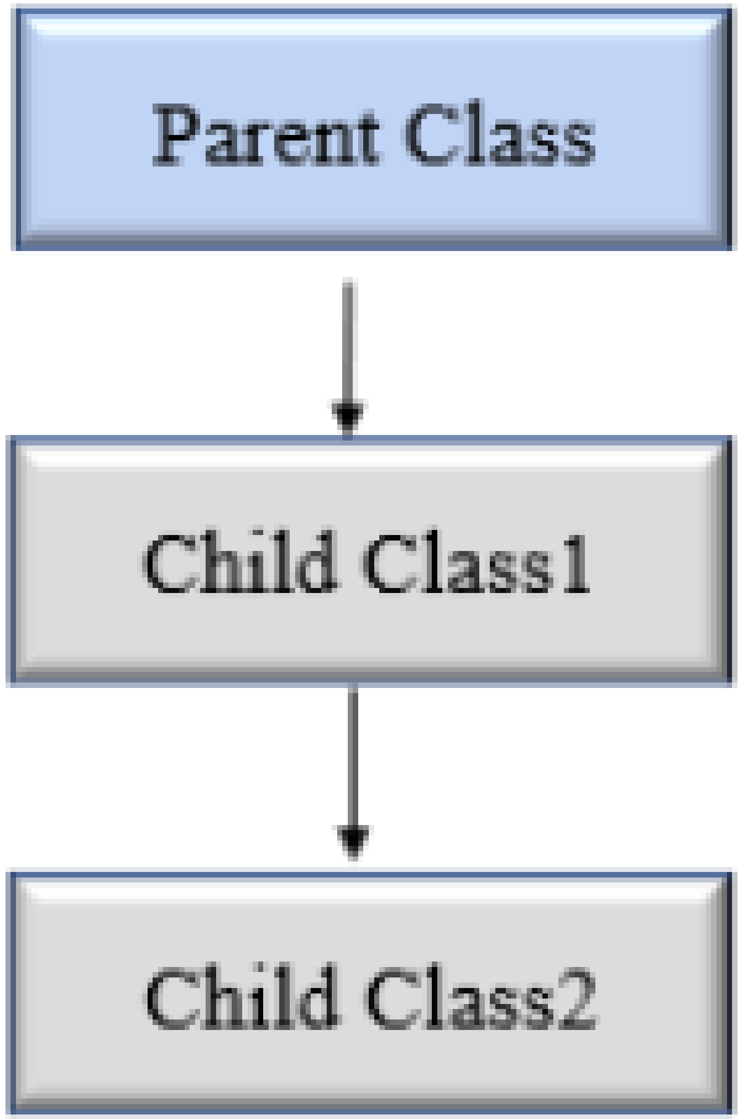


```python
# Child class
class Employee(Person, Company):
    def Employee_info(self, salary, skill):
        print('Inside Employee class')
        print('Salary:', salary, 'Skill:', skill)

# Create object of Employee
emp = Employee()

# access data
emp.person_info('Jessa', 28)
emp.company_info('Google', 'Atlanta')
emp.Employee_info(12000, 'Machine Learning')
```

Multiple Inheritance


Output

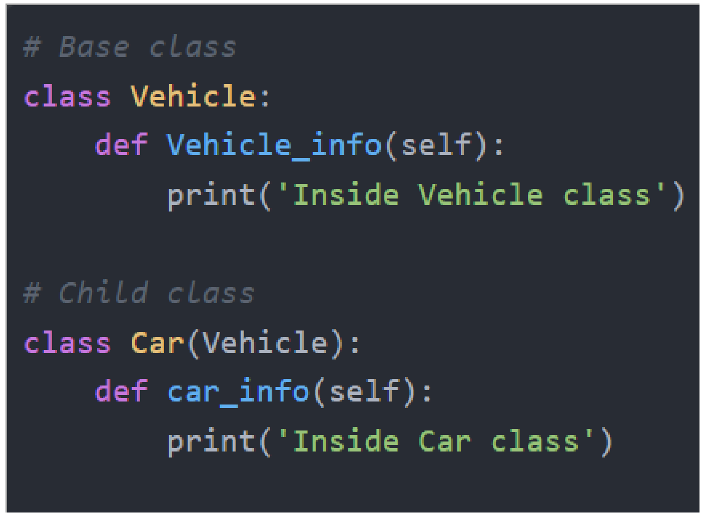
```
Inside Person class
Name: Jessa Age: 28

Inside Company class
Name: Google location: Atlanta

Inside Employee class
Salary: 12000 Skill: Machine Learning
```



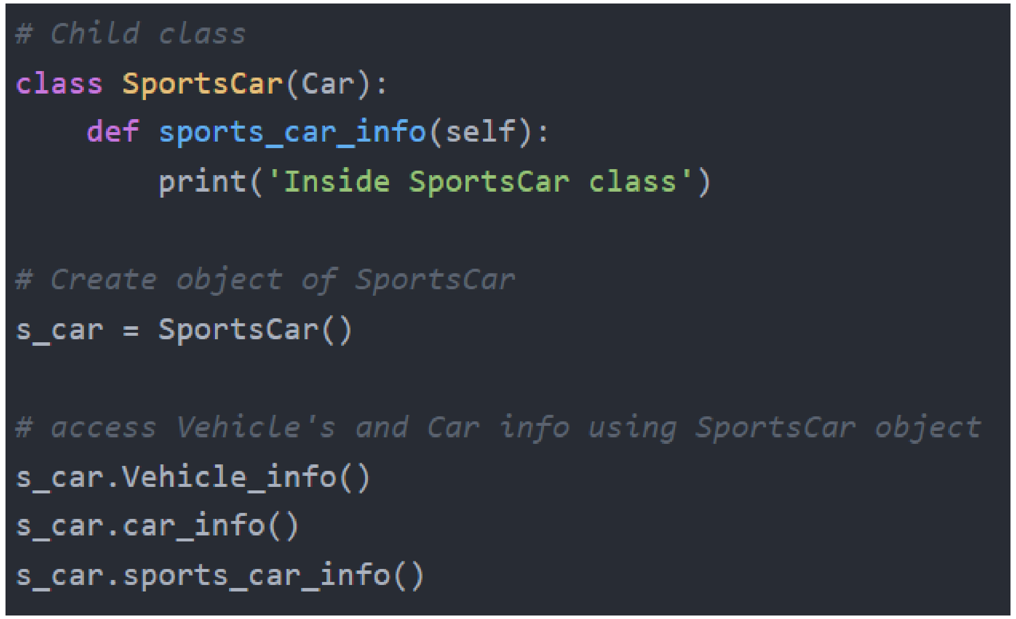
Multilevel inheritance

- In multilevel inheritance, a class inherits from a child class or derived class.
- Suppose three classes A, B, C. A is the superclass, B is the child class of A, C is the child class of B. In other words, we can say a chain of classes is called multilevel inheritance.

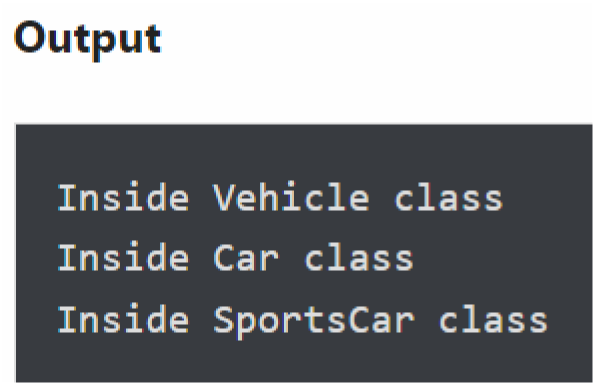
Multilevel inheritance



```python
# Base class
class Vehicle:
    def Vehicle_info(self):
        print('Inside Vehicle class')

# Child class
class Car(Vehicle):
    def car_info(self):
        print('Inside Car class')
```



```python
# Child class
class SportsCar(Car):
    def sports_car_info(self):
        print('Inside SportsCar class')

# Create object of SportsCar
s_car = SportsCar()

# access Vehicle's and Car info using SportsCar object
s_car.Vehicle_info()
s_car.car_info()
s_car.sports_car_info()
```

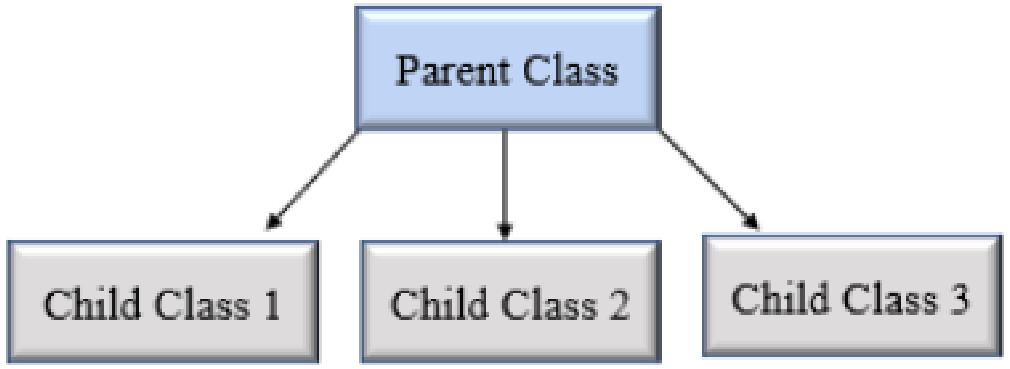


Output

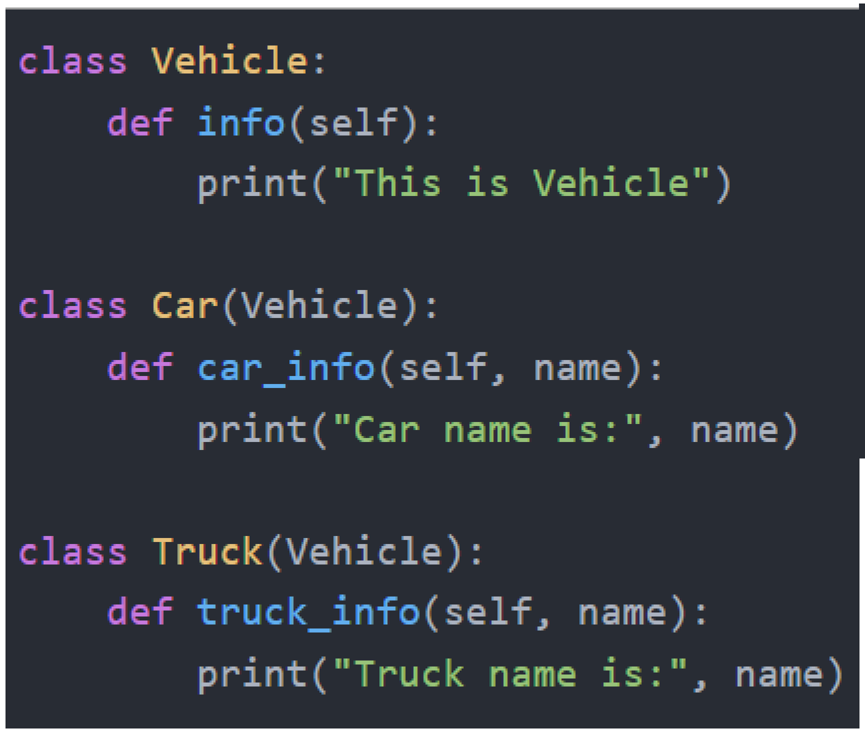
```
Inside Vehicle class
Inside Car class
Inside SportsCar class
```

Hierarchical Inheritance

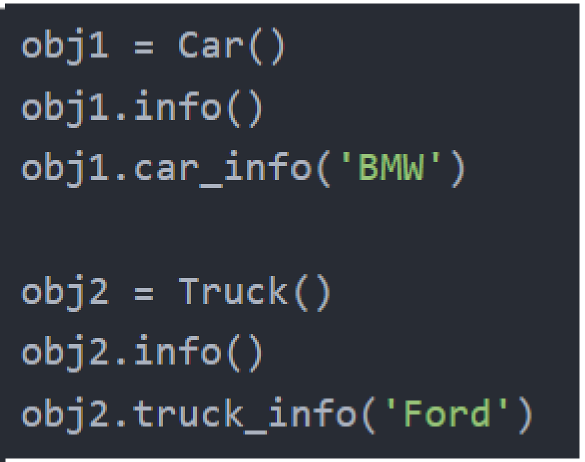
- In Hierarchical inheritance, more than one child class is derived from a single-parent class. In other words, we can say one parent class and multiple child classes.



Hierarchical Inheritance

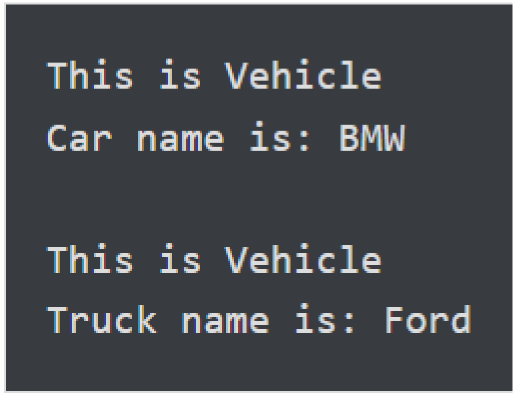


```python
class Vehicle:
    def info(self):
        print("This is Vehicle")

class Car(Vehicle):
    def car_info(self, name):
        print("Car name is:", name)

class Truck(Vehicle):
    def truck_info(self, name):
        print("Truck name is:", name)
```



```python
obj1 = Car()
obj1.info()
obj1.car_info('BMW')

obj2 = Truck()
obj2.info()
obj2.truck_info('Ford')
```

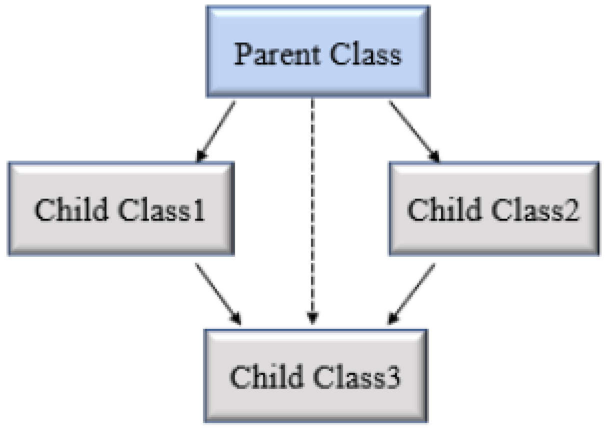
Output



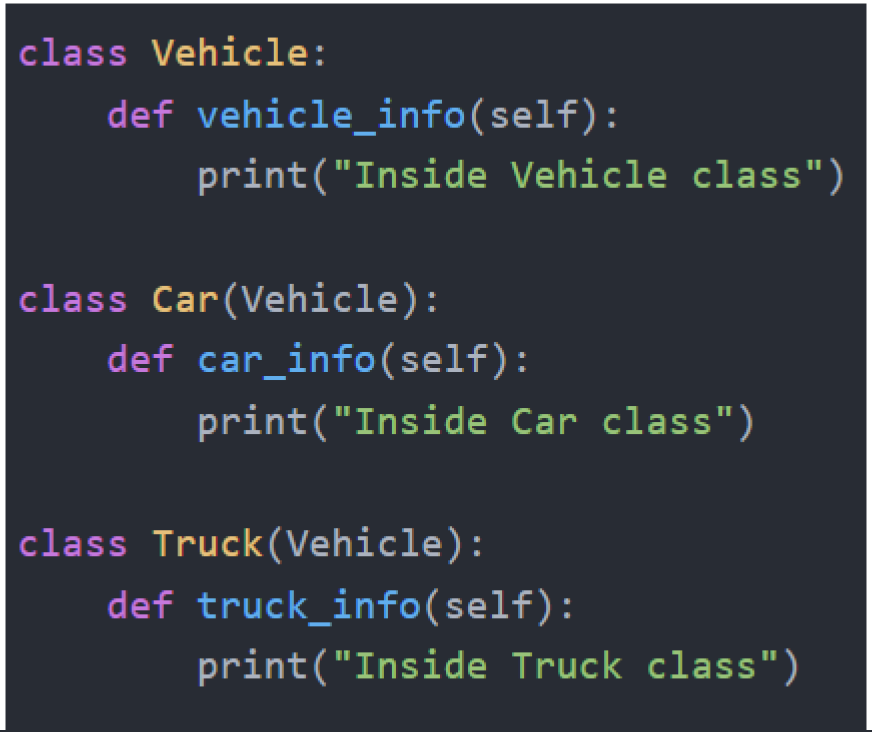
```
This is Vehicle
Car name is: BMW

This is Vehicle
Truck name is: Ford
```

Hybrid Inheritance

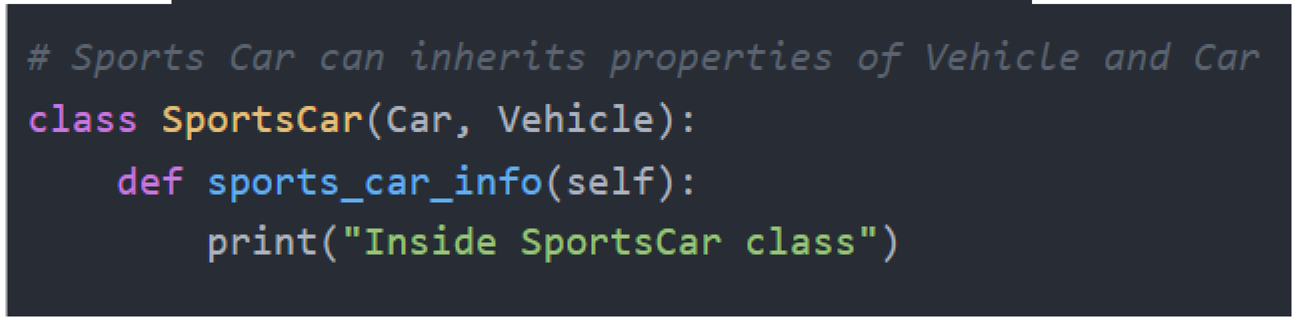
- When inheritance is consists of multiple types or a combination of different inheritance is called hybrid inheritance.



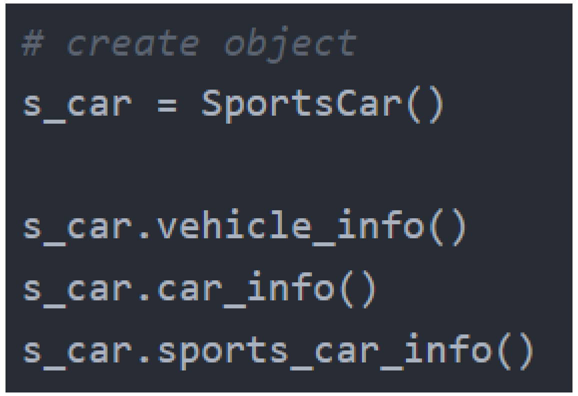
Hybrid Inheritance



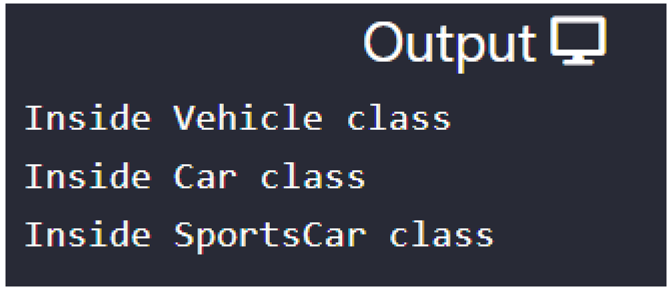
```python
class Vehicle:
    def vehicle_info(self):
        print("Inside Vehicle class")

class Car(Vehicle):
    def car_info(self):
        print("Inside Car class")

class Truck(Vehicle):
    def truck_info(self):
        print("Inside Truck class")
```



```python
# Sports Car can inherits properties of Vehicle and Car
class SportsCar(Car, Vehicle):
    def sports_car_info(self):
        print("Inside SportsCar class")
```



```python
# create object
s_car = SportsCar()

s_car.vehicle_info()
s_car.car_info()
s_car.sports_car_info()
```



```
Inside Vehicle class
Inside Car class
Inside SportsCar class
```

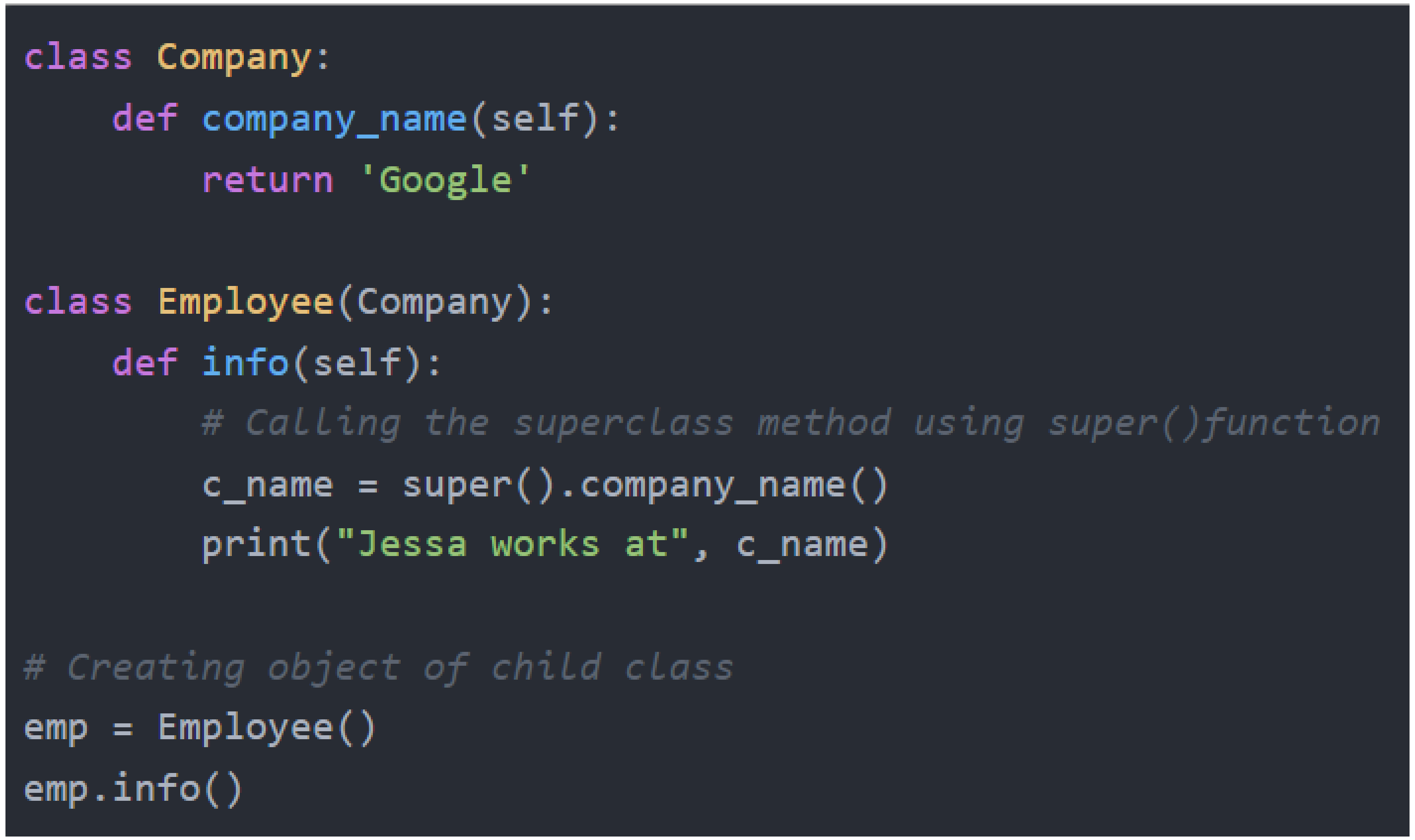
Python super() function

- When a class inherits all properties and behavior from the parent class is called inheritance. In such a case, **the inherited class is a subclass** and **the latter class is the parent class**.
- In child class, we can refer to the parent class by using the **super() function**.
- The **super function returns a temporary object of the parent class** that allows us to call a parent class method inside a child class method.

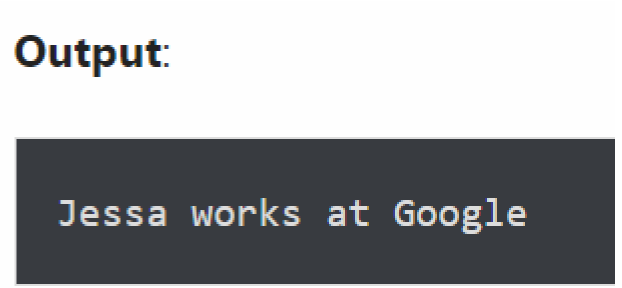
Benefits of using the super() function.

- We are not required to remember or specify the parent class name to access its methods.
- We can use the **super() function in both single and multiple inheritances.**
- The super() function support **code reusability** as there is no need to write the entire function

Python super() function



```python
class Company:
    def company_name(self):
        return 'Google'

class Employee(Company):
    def info(self):
        # Calling the superclass method using super()function
        c_name = super().company_name()
        print("Jessa works at", c_name)

# Creating object of child class
emp = Employee()
emp.info()
```

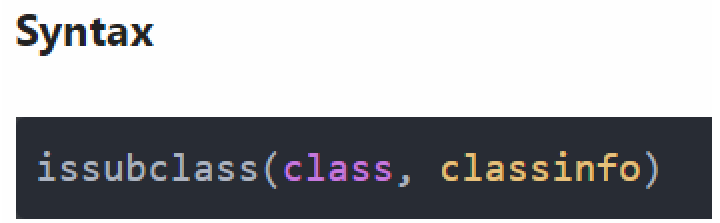


Output:

```
Jessa works at Google
```

issubclass()

- In Python, we can verify whether a particular class is a subclass of another class. For this purpose, we can use Python built-in function issubclass().
- This function returns True if the given class is the subclass of the specified class. Otherwise, it returns False.

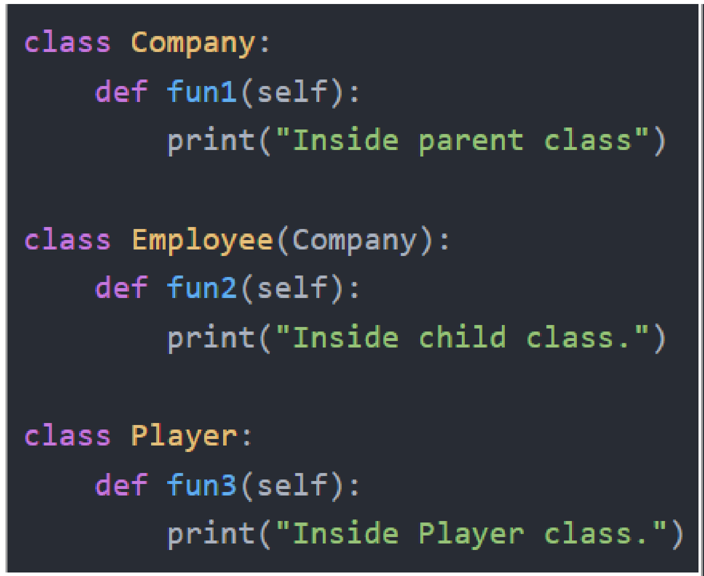

Syntax

```
issubclass(class, classinfo)
```

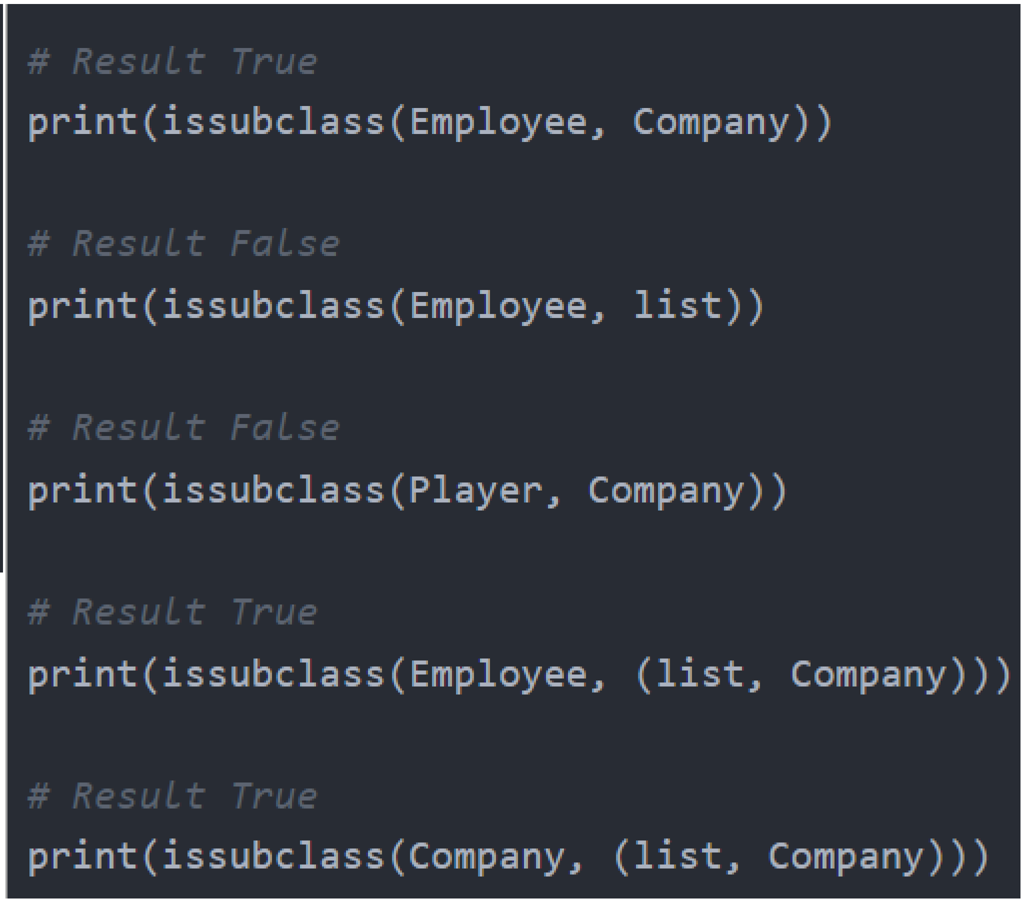
class: class to be checked.

classinfo: a class, type, or a tuple of classes or data types.

issubclass()



```python
class Company:
    def fun1(self):
        print("Inside parent class")

class Employee(Company):
    def fun2(self):
        print("Inside child class.")

class Player:
    def fun3(self):
        print("Inside Player class.")
```



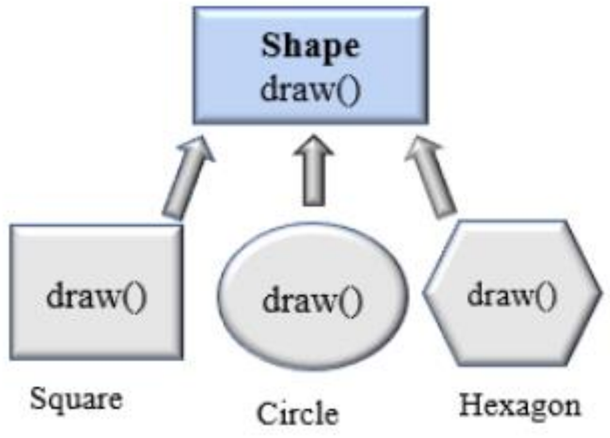
```python
# Result True
print(issubclass(Employee, Company))

# Result False
print(issubclass(Employee, list))

# Result False
print(issubclass(Player, Company))

# Result True
print(issubclass(Employee, (list, Company)))

# Result True
print(issubclass(Company, (list, Company)))
```

Method Overriding

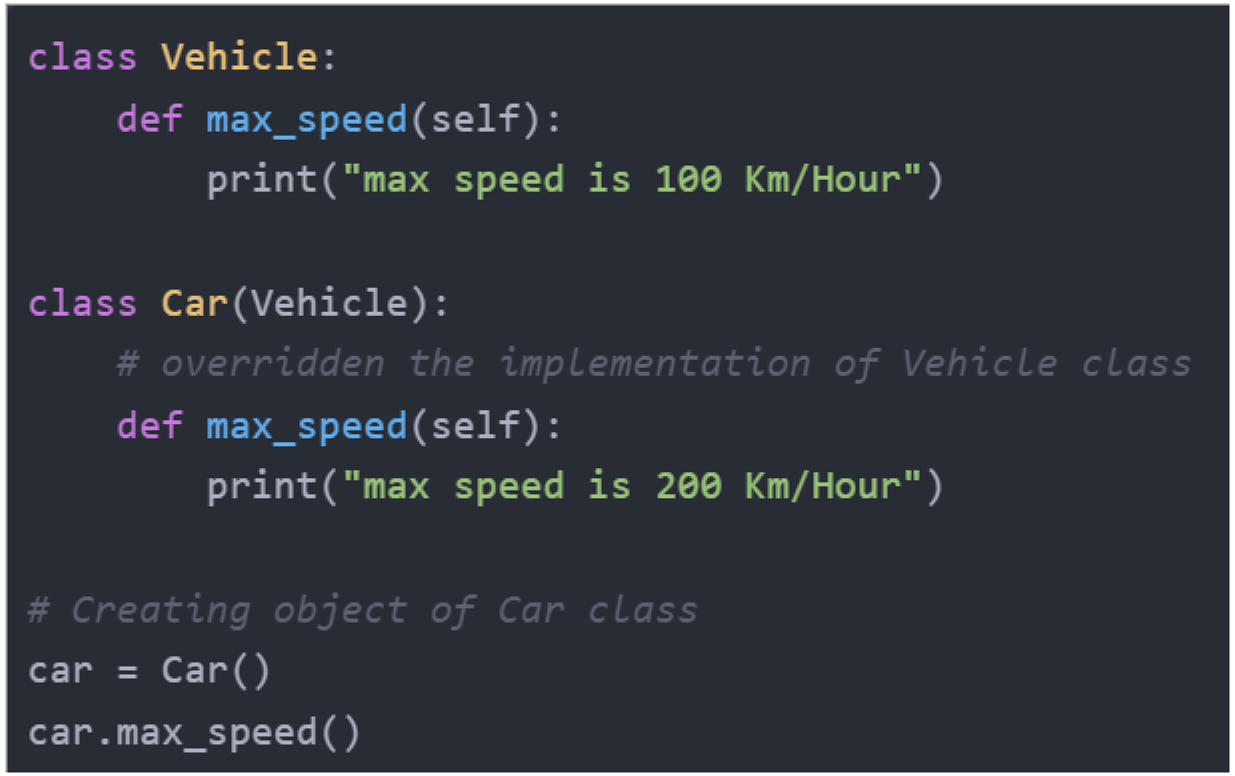
- In inheritance, all members available in the parent class are by default available in the child class. If the child class does not satisfy with parent class implementation, then the child class is allowed to redefine that method by extending additional functions in the child class. This concept is called method overriding.

Method Overriding

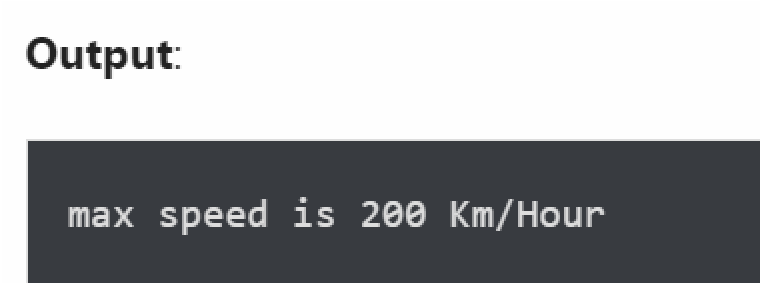
- When a child class method has the same name, same parameters, and same return type as a method in its superclass, then the method in the child is said to override the method in the parent class.



Method Overriding



```python
class Vehicle:
    def max_speed(self):
        print("max speed is 100 Km/Hour")

class Car(Vehicle):
    # overridden the implementation of Vehicle class
    def max_speed(self):
        print("max speed is 200 Km/Hour")

# Creating object of Car class
car = Car()
car.max_speed()
```



Output:

```
max speed is 200 Km/Hour
```

Method Resolution Order in Python

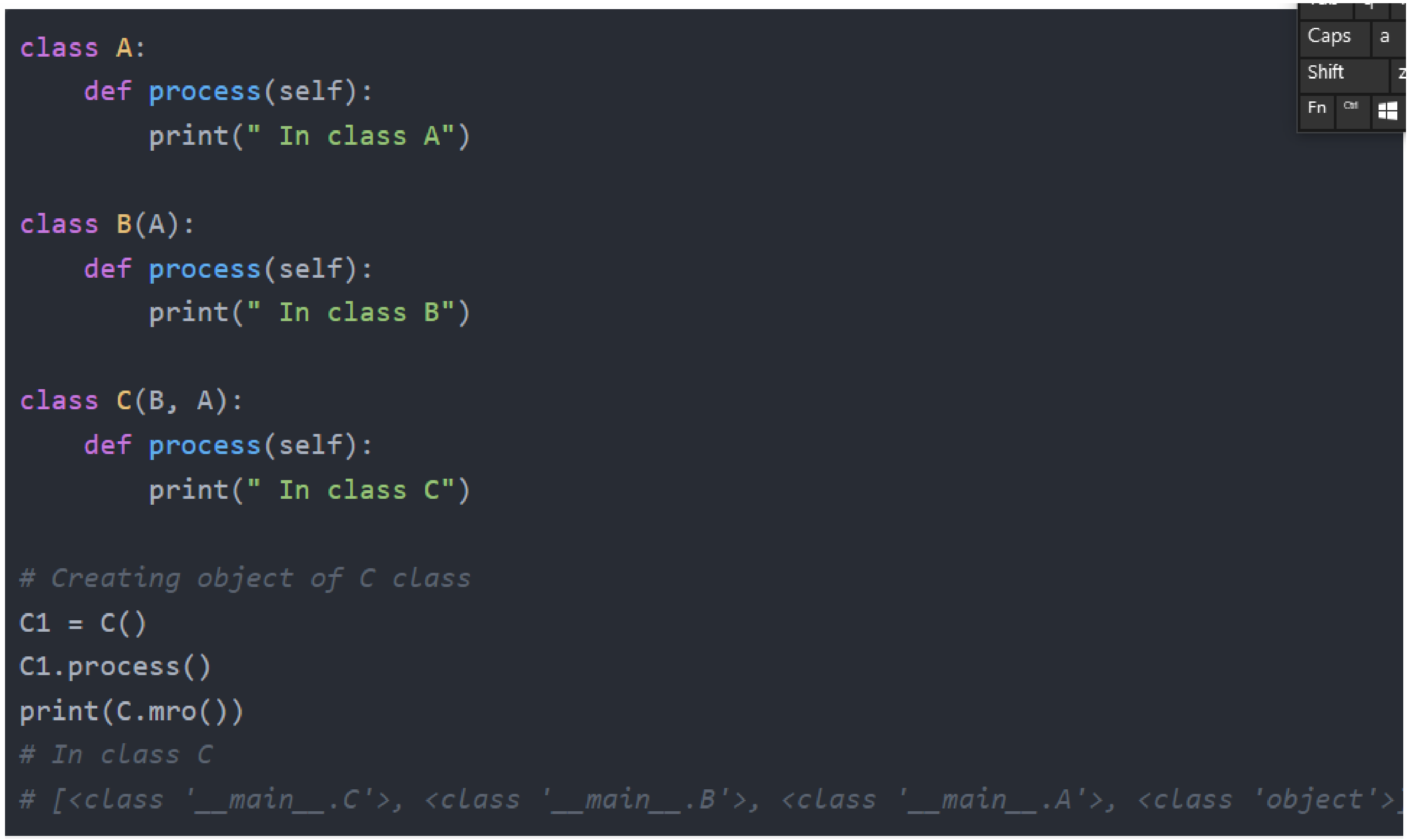
- In Python, Method Resolution Order(MRO) is the order by which Python looks for a method or attribute. First, the method or attribute is searched within a class, and then it follows the order we specified while inheriting.
- This order is also called the Linearization of a class, and a set of rules is called MRO (Method Resolution Order). The MRO plays an essential role in multiple inheritances as a single method may found in multiple parent classes.

Method Resolution Order in Python

- In multiple inheritance, the following search order is followed.

- First, it searches in the current class if not available, then searches in the parents class specified while inheriting (that is left to right.)
- We can get the MRO of a class. For this purpose, we can use either the mro attribute or the **mro() method**.

Method Resolution Order in Python



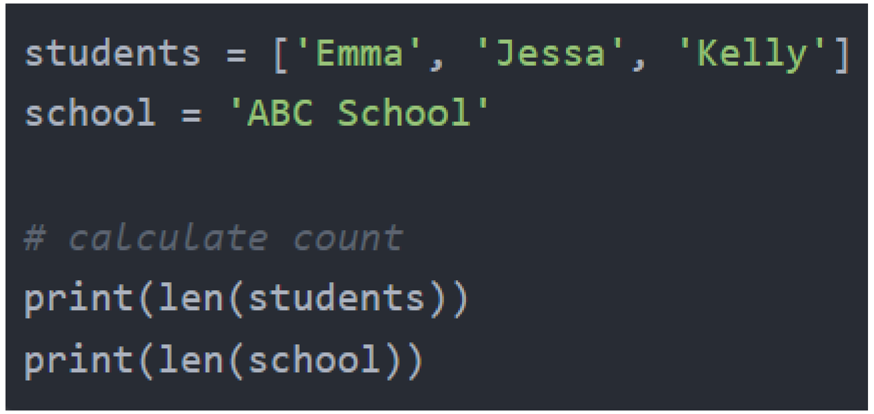
```
class A:
    def process(self):
        print(" In class A")

class B(A):
    def process(self):
        print(" In class B")

class C(B, A):
    def process(self):
        print(" In class C")

# Creating object of C class
C1 = C()
C1.process()
print(C.mro())
# In class C
# [<class '__main__.C'>, <class '__main__.B'>, <class '__main__.A'>, <class 'object'>
```

Polymorphism in Python

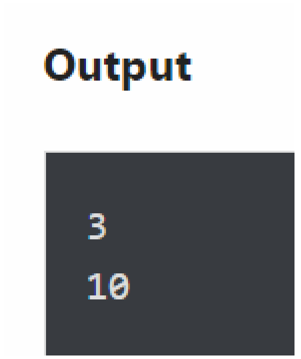
- Polymorphism in Python is the ability of an object to take many forms.
- In simple words, polymorphism allows us to perform the same action in many different ways.
- In polymorphism, a method can process objects differently depending on the class type or data type.
- Polymorphism is achieved by overloading and overriding.
- Python doesn’t support method overloading.

Polymorphism in Built-in function len()

- The built-in function len() calculates the length of an object depending upon its type. If an object is a string, it returns the count of characters, and If an object is a list, it returns the count of items in a list.
- The len() method treats an object as per its class type.



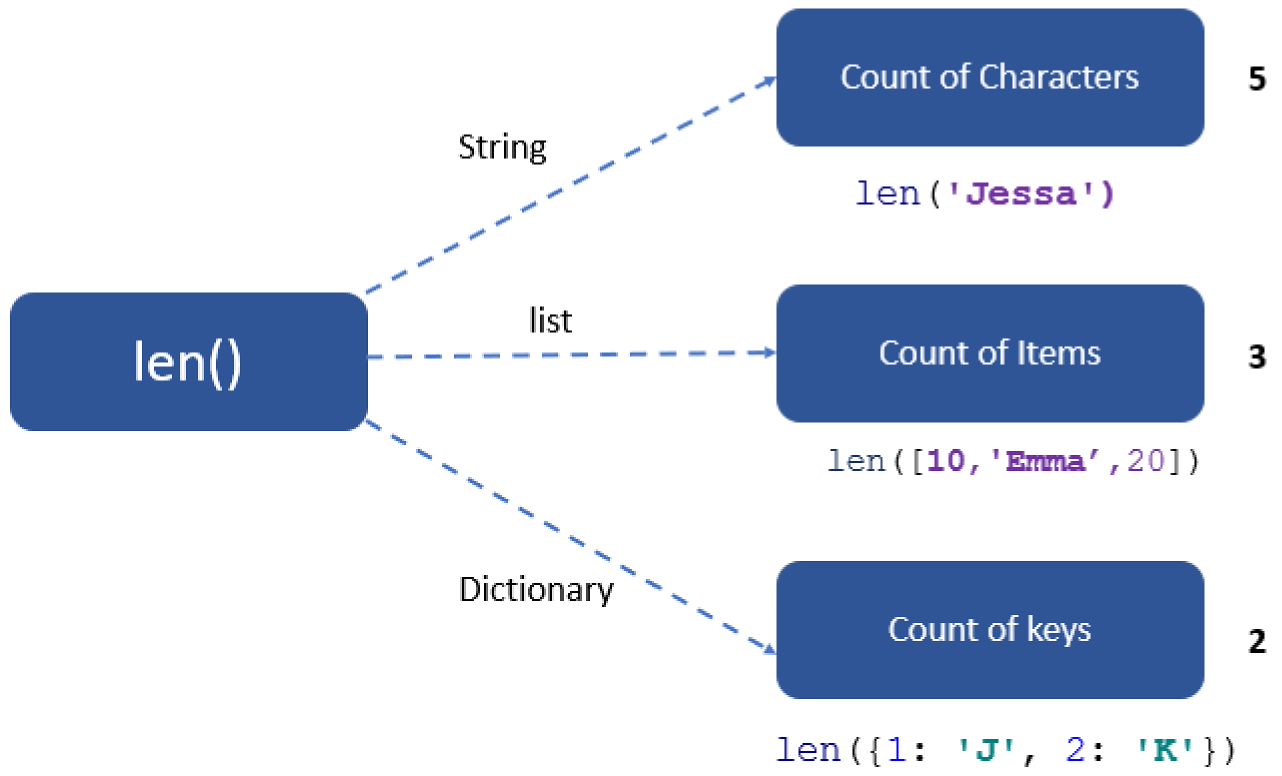
```
students = ['Emma', 'Jessa', 'Kelly']
school = 'ABC School'

# calculate count
print(len(students))
print(len(school))
```



Output

```
3
10
```

Polymorphism in Built-in function len()



Polymorphic len() function

Polymorphism With Inheritance

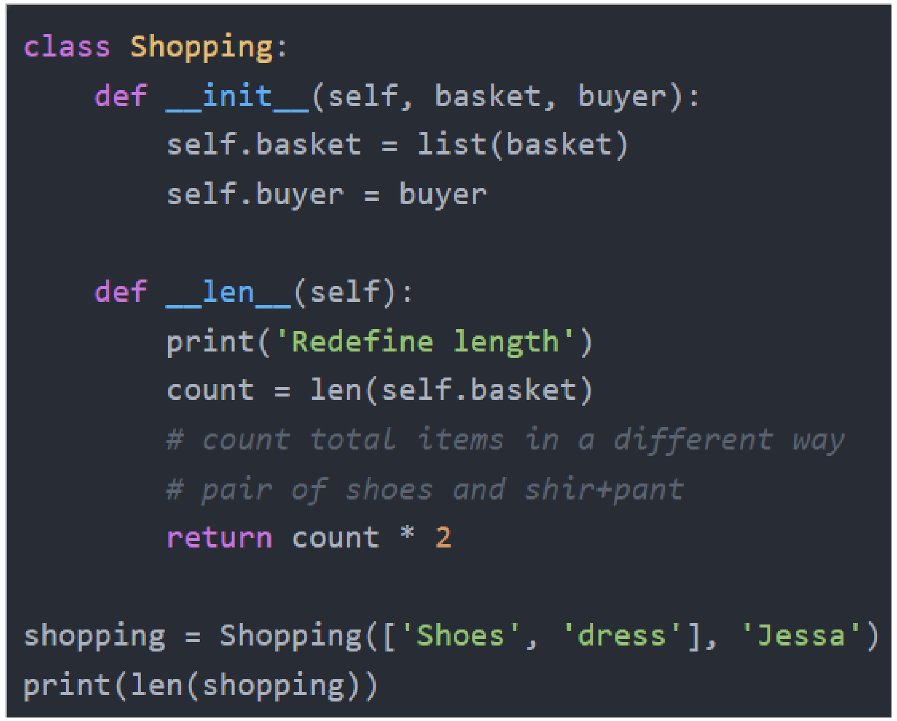
- Polymorphism is mainly used with inheritance. In inheritance, child class inherits the attributes and methods of a parent class. The existing class is called a base class or parent class, and the new class is called a subclass or child class or derived class.

- Using method overriding, polymorphism allows us to defines methods in the child class that have the same name as the methods in the parent class. This process of re-implementing the inherited method in the child class is known as Method Overriding.

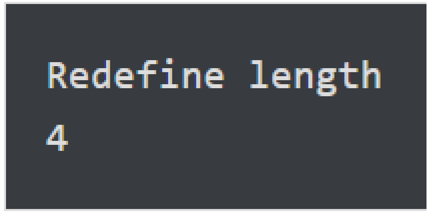
Overrride Built-in Functions

- In Python, we can change the default behavior of the built-in functions.
- For example, we can change or extend the built-in functions such as len(), abs(), or divmod() by redefining them in our class.

Overrride Built-in Functions



```python
class Shopping:
    def __init__(self, basket, buyer):
        self.basket = list(basket)
        self.buyer = buyer

    def __len__(self):
        print('Redefine length')
        count = len(self.basket)
        # count total items in a different way
        # pair of shoes and shir+pant
        return count * 2

shopping = Shopping(['Shoes', 'dress'], 'Jessa')
print(len(shopping))
```



```
Redefine length
4
```

Abstraction

- Abstraction is one of the important principles of object-oriented programming.
- It refers to a programming approach by which only the relevant data about an object is exposed, hiding all the other details.
- This approach helps in reducing the complexity and increasing the efficiency of application development.

Types of Python Abstraction

- There are two types of abstraction.
- One is **data abstraction**, wherein the original data entity is hidden via a data structure that can internally work through the hidden data entities.
- Another type is called **process abstraction**. It refers to hiding the underlying implementation details of a process.

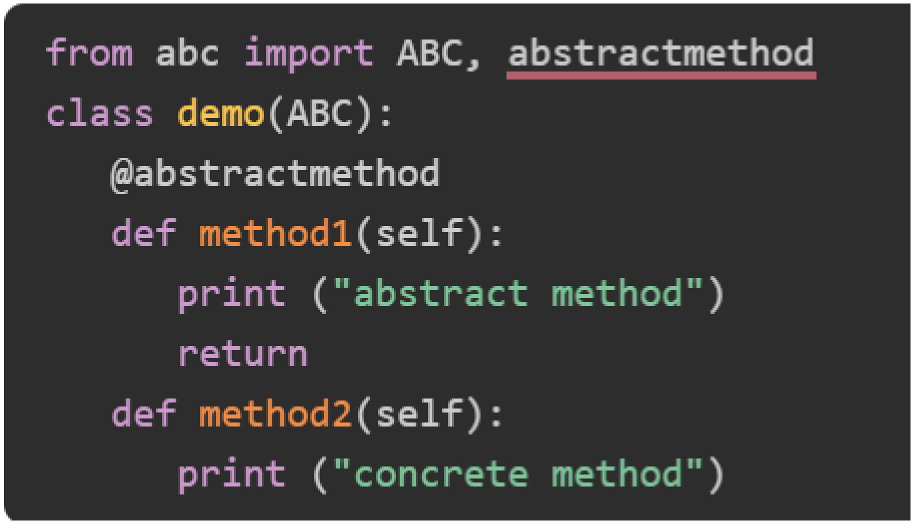
Python Abstract Class

- In object-oriented programming terminology, a class is said to be an **abstract class if it cannot be instantiated**, that is you can’t have an object of an abstract class. You can however use it as a base or parent class for constructing other classes.

Create an Abstract Class

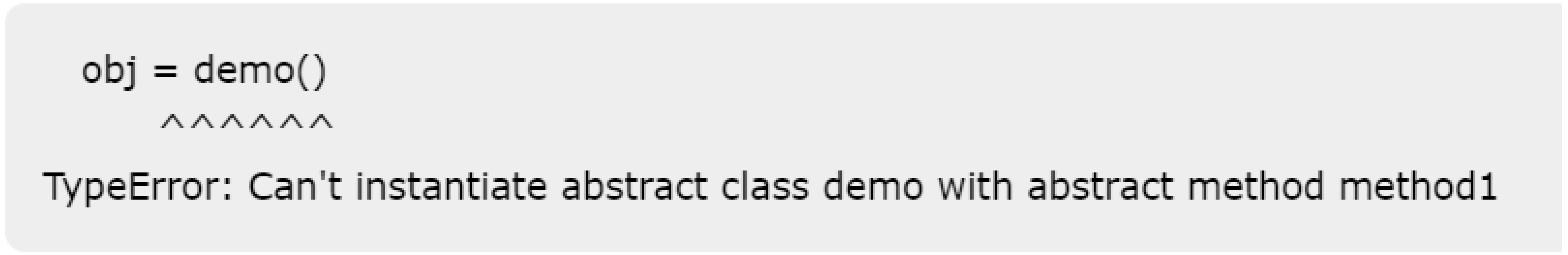
- To create an abstract class in Python, it must inherit the ABC class that is defined in the built-in ABC module.
- This module is available in Python's standard library.
- Moreover, the class must have at least one abstract method.
- Again, an abstract method is the one which cannot be called but can be overridden.
- You need to decorate it with @abstractmethod decorator which is defined in ABC module.

Example: Create an Abstract Class



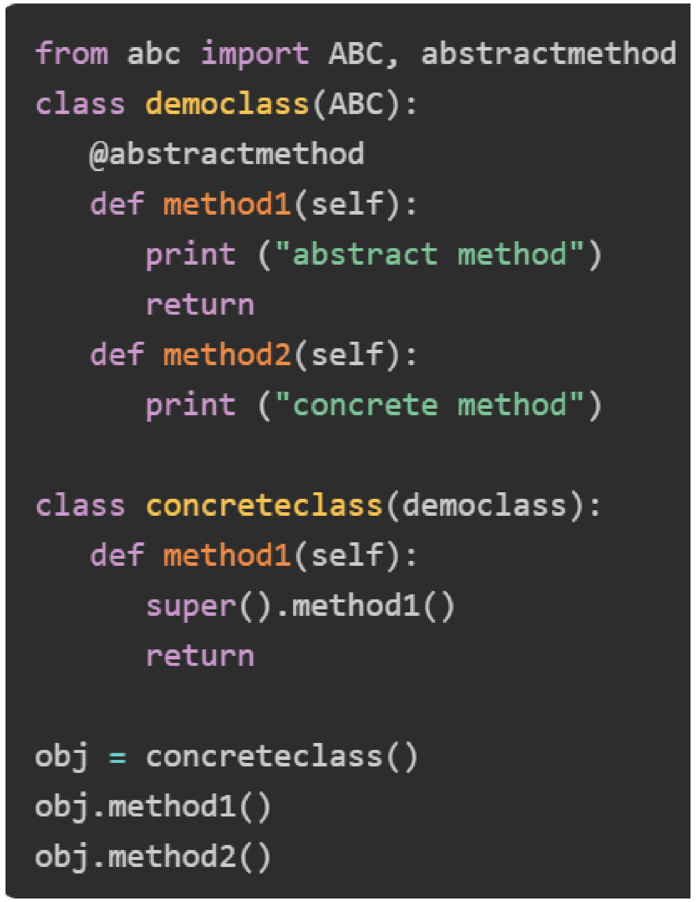
```
from abc import ABC, abstractmethod
class demo(ABC):
   @abstractmethod
   def method1(self):
      print ("abstract method")
      return
   def method2(self):
      print ("concrete method")
```

- The demo class inherits ABC class. There is a method1() which is an abstract method. Note that the class may have other non-abstract (concrete) methods.
- If you try to declare an object of demo class, Python raises TypeError –
- The demo class here may be used as parent for another class. However, the child class must override the abstract method in parent class. If not, Python throws TypeError –

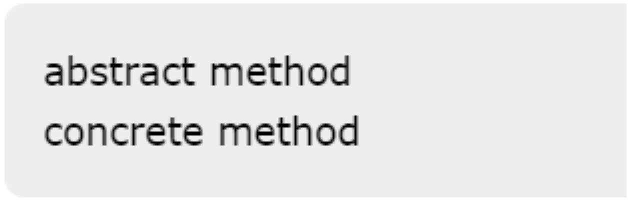


```
   obj = demo()
         ^^^^^^
TypeError: Can't instantiate abstract class demo with abstract method method1
```

Abstract Method Overriding



```python
from abc import ABC, abstractmethod
class democlass(ABC):
   @abstractmethod
   def method1(self):
      print ("abstract method")
      return
   def method2(self):
      print ("concrete method")

class concreteclass(democlass):
   def method1(self):
      super().method1()
      return

obj = concreteclass()
obj.method1()
obj.method2()
```



```
abstract method
concrete method
```