Information
Technology
Institute

Python 3

Prepared by

Hesham sayed

Agenda

➢ Object-Oriented Programming (OOPs):

- Class
- Objects
- Polymorphism
- Encapsulation
- Inheritance
- Data Abstraction
- Class Method
- Constructors
- Destructors

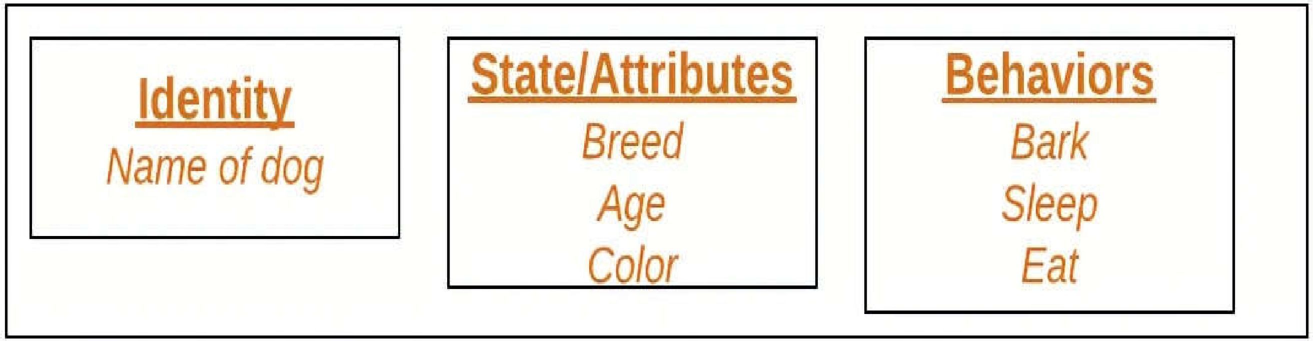
Class Objects

An Object is an instance of a Class. A class is like a blueprint while an instance is a copy of the class with actual values. It’s not an idea anymore, it’s an actual dog, like a dog of breed pug who’s seven years old. You can have many dogs to create many different instances, but without the class as a guide, you would be lost, not knowing what information is required.

An object consists of :

- State: It is represented by the attributes of an object. It also reflects the properties of an object.
- Behaviour: It is represented by the methods of an object. It also reflects the response of an object to other objects.

- Identity: It gives a unique name to an object and enables one object to interact with other objects.

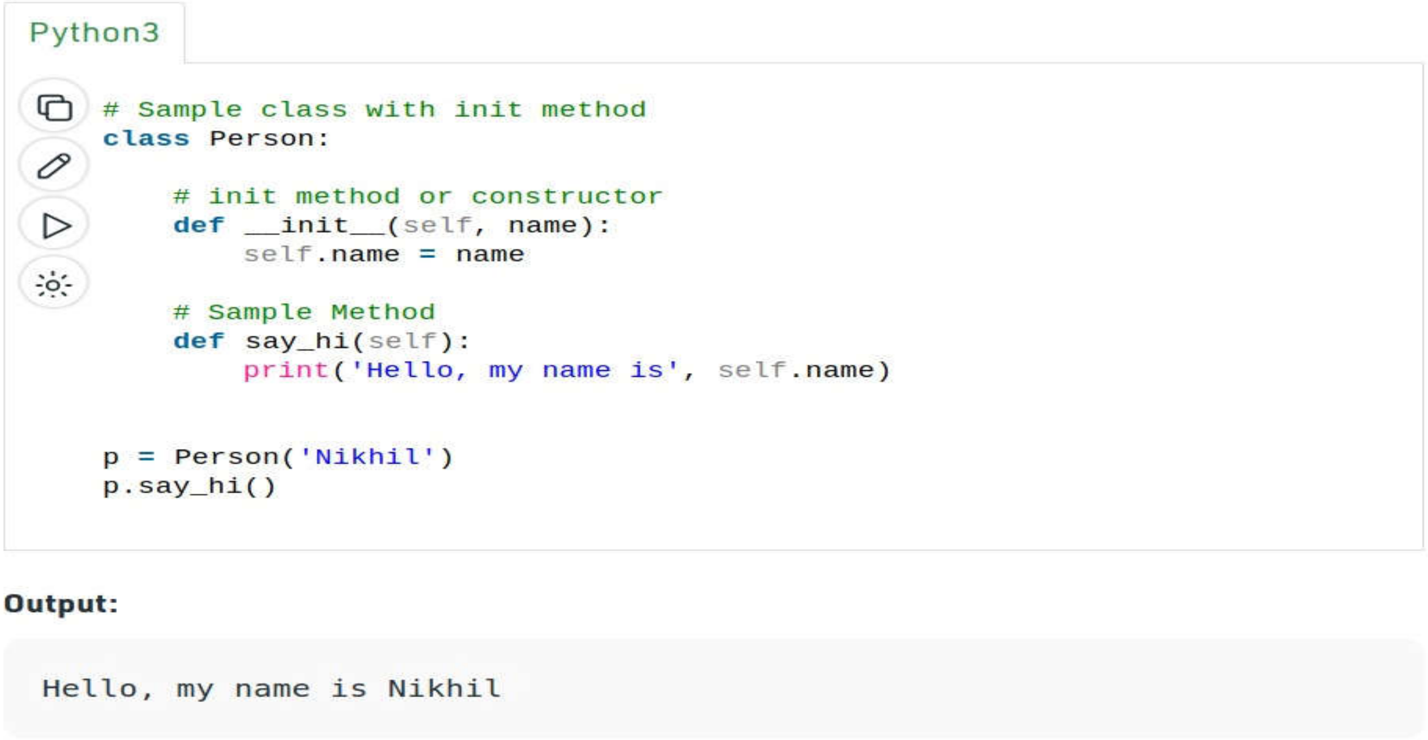


The self

- Class methods must have an extra first parameter in the method definition. We do not give a value for this parameter when we call the method, Python provides it.
- If we have a method that takes no arguments, we still have one argument.
- This is similar to this pointer in C++ and this reference in Java.

init method:

The _init_ method is similar to constructors in C++ and Java. Constructors are used to initialize the object's state. Like methods, a constructor also contains a collection of statements(i.e. instructions) that are executed at the time of Object creation. It runs as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object.



```python
# Sample class with init method
class Person:

    # init method or constructor
    def __init__(self, name):
        self.name = name

    # Sample Method
    def say_hi(self):
        print('Hello, my name is', self.name)


p = Person('Nikhil')
p.say_hi()
```

Output:

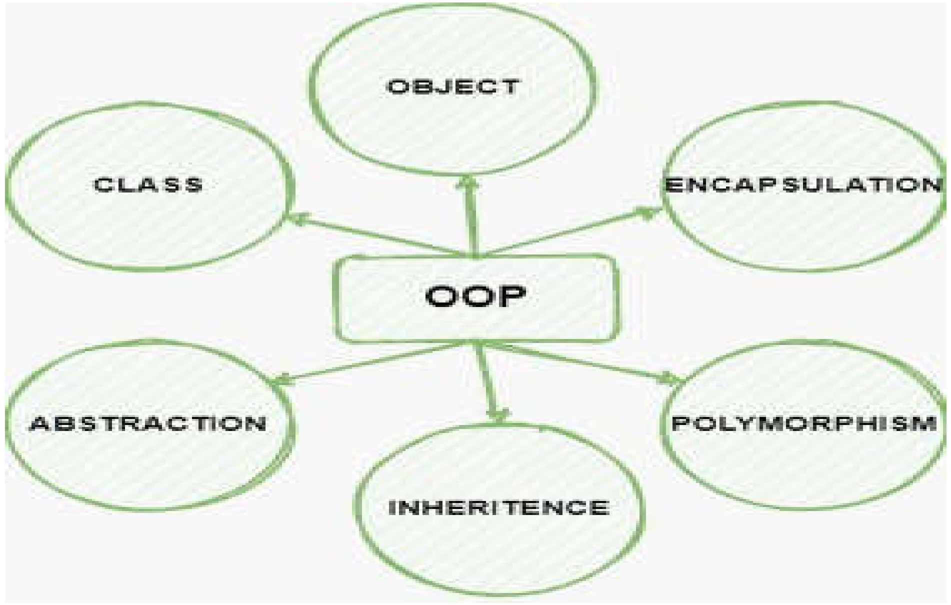
```
Hello, my name is Nikhil
```

Object-Oriented Programming (OOPs)

In Python, object-oriented Programming (OOPs) is a programming paradigm that uses objects and classes in programming. It aims to implement real-world entities like inheritance, polymorphisms, encapsulation, etc. in the programming. The main concept of OOPs is to bind the data and the functions that work on that together as a single unit so that no other part of the code can access this data.

Main Concepts of Object-Oriented Programming (OOPs)

- Class
- Objects
- Polymorphism
- Encapsulation
- Inheritance
- Data Abstraction



- **<u>Class</u>**

A class is a collection of objects. A class contains the blueprints or the prototype from which the objects are being created. It is a logical entity that contains some attributes and methods.

To understand the need for creating a class let’s consider an example, let’s say you wanted to track the number of dogs that may have different attributes like breed, age. If a list is used, the first element could be the dog’s breed while the second element could represent its age. Let’s suppose there are 100 different dogs, then how would you know which element is supposed to be which? What if you wanted to add other properties to these dogs? This lacks organization and it’s the exact need for classes.

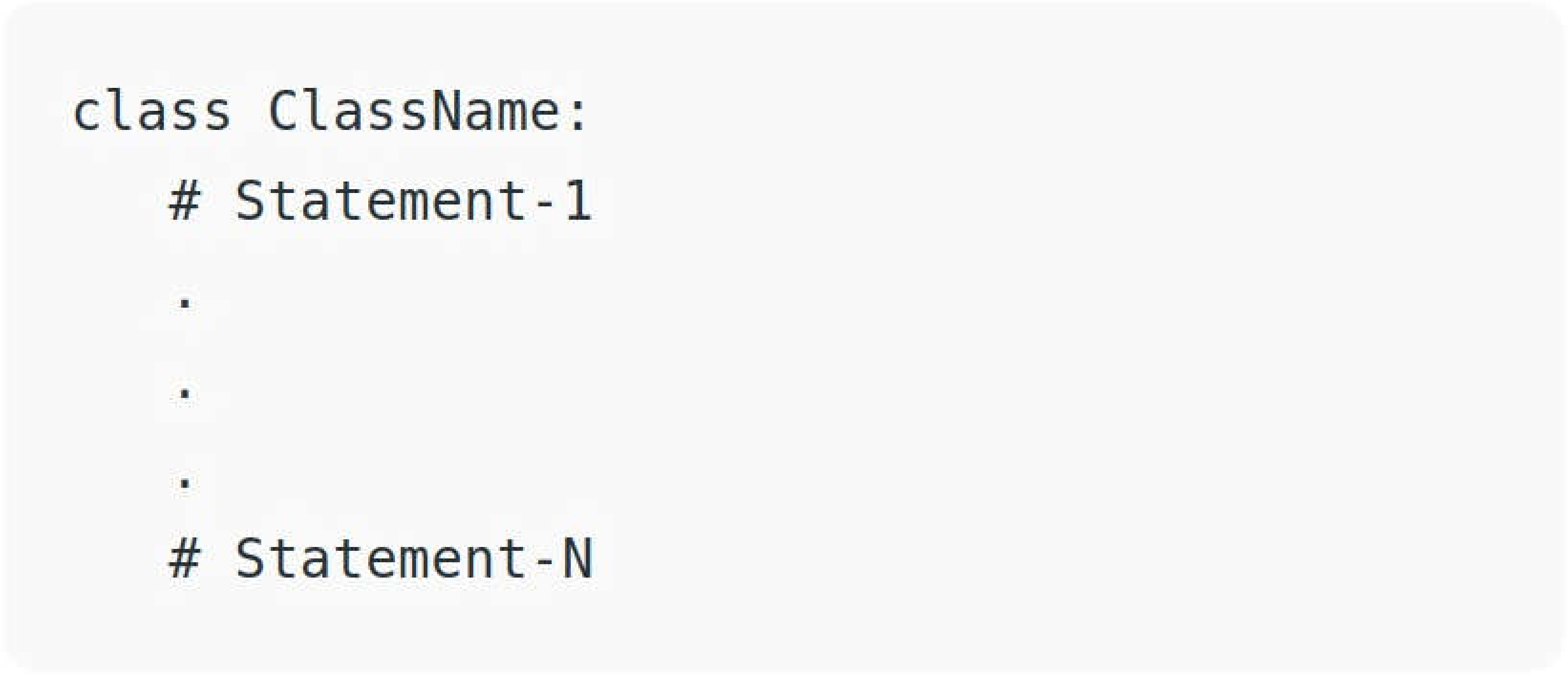
Some points on Python class:

Classes are created by keyword class.

Attributes are the variables that belong to a class.

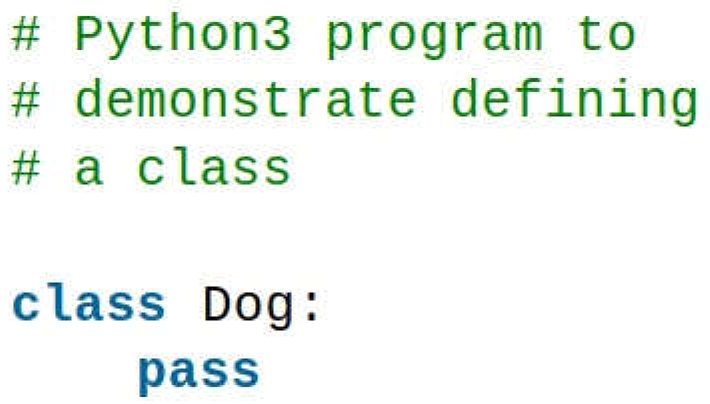
Attributes are always public and can be accessed using the dot (.) operator. Eg.: Myclass.Myattribute

Class Definition Syntax:



```
class ClassName:
    # Statement-1
    .
    .
    .
    # Statement-N
```

Example: Creating an empty Class in Python



```python
# Python3 program to
# demonstrate defining
# a class

class Dog:
    pass
```

In the above example, we have created a class named dog using the class keyword.

- **Objects**

The object is an entity that has a state and behavior associated with it. It may be any real-world object like a mouse, keyboard, chair, table, pen, etc. Integers, strings, floating-point numbers, even arrays, and dictionaries, are all objects. More specifically, any single integer or any single string is an object. The number 12 is an object, the string “Hello, world” is an object, a list is an object that can hold other objects, and so on. You’ve been using objects all along and may not even realize it.

An object consists of :

- **State**:

It is represented by the attributes of an object. It also reflects the properties of an object.

- **Behavior:**

It is represented by the methods of an object. It also reflects the response of an object to other objects.

- **Identity:**

It gives a unique name to an object and enables one object to interact with other objects.

To understand the state, behavior, and identity let us take the example of the class dog (explained above).

- The identity can be considered as the name of the dog.

- State or Attributes can be considered as the breed, age, or color of the dog.

- The behavior can be considered as to whether the dog is eating or sleeping.

Example: Creating an object

obj = Dog()

This will create an object named obj of the class Dog defined above. Before diving deep into objects and class let us understand some basic keywords that will we used while working with objects and classes.

The self

- Class methods must have an extra first parameter in the method definition. We do not give a value for this parameter when we call the method, Python provides it

- If we have a method that takes no arguments, then we still have to have one argument.

- This is similar to this pointer in C++ and this reference in Java.

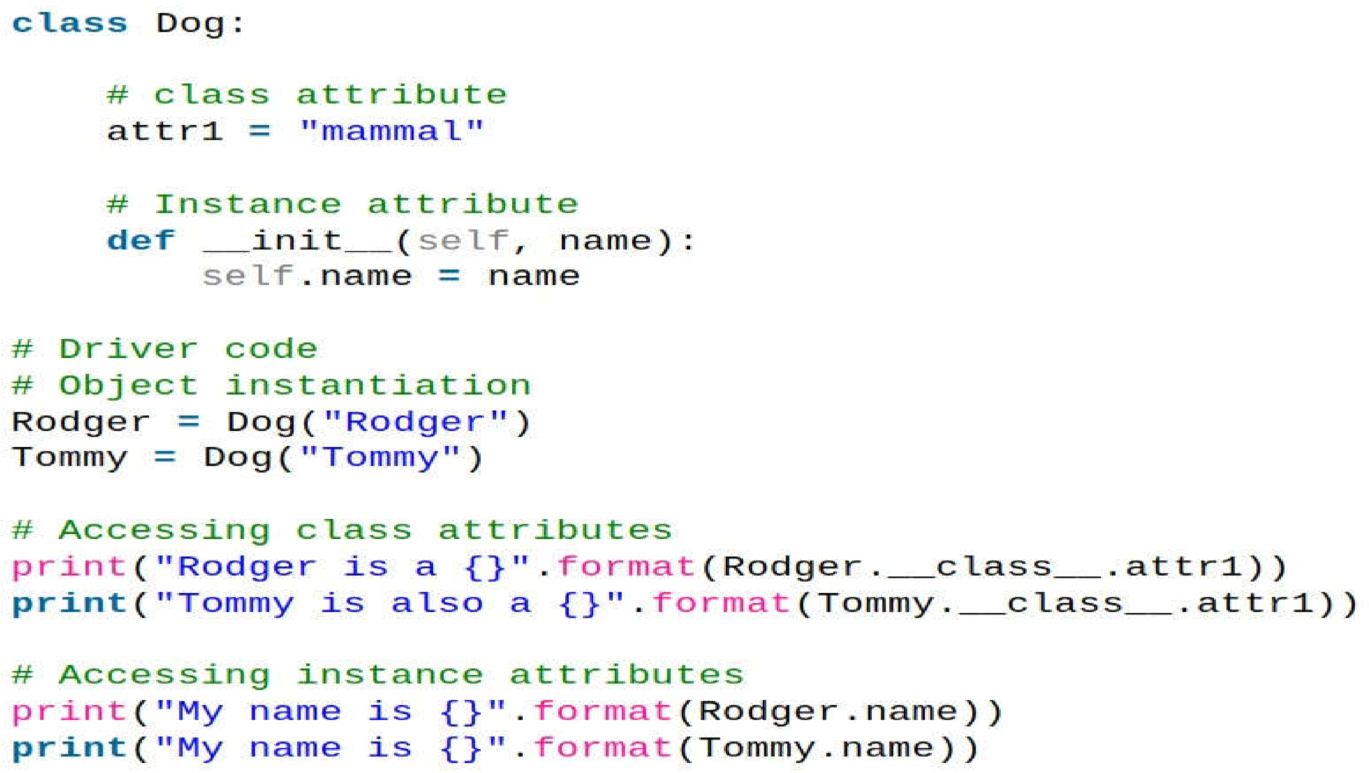
When we call a method of this object as myobject.method(arg1, arg2), this is automatically converted by Python into MyClass.method(myobject, arg1, arg2) – this is all the special self is about.

- **The _init_ method**

The _init_ method is similar to constructors in C++ and Java. It is run as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object.

Now let us define a class and create some objects using the self and _init_ method.

Example 1: Creating a class and object with class and instance attributes



```
class Dog:

    # class attribute
    attr1 = "mammal"

    # Instance attribute
    def __init__(self, name):
        self.name = name

# Driver code
# Object instantiation
Rodger = Dog("Rodger")
Tommy = Dog("Tommy")

# Accessing class attributes
print("Rodger is a {}".format(Rodger.__class__.attr1))
print("Tommy is also a {}".format(Tommy.__class__.attr1))

# Accessing instance attributes
print("My name is {}".format(Rodger.name))
print("My name is {}".format(Tommy.name))
```

Output

```
Rodger is a mammal
Tommy is also a mammal
My name is Rodger
My name is Tommy
```

Example 2: Creating Class and objects with methods

```
class Dog:

    # class attribute
    attr1 = "mammal"

    # Instance attribute
    def __init__(self, name):
        self.name = name

    def speak(self):
        print("My name is {}".format(self.name))

# Driver code
# Object instantiation
Rodger = Dog("Rodger")
Tommy = Dog("Tommy")

# Accessing class methods
Rodger.speak()
Tommy.speak()
```

Output

```
My name is Rodger
My name is Tommy
```

- **Inheritance**

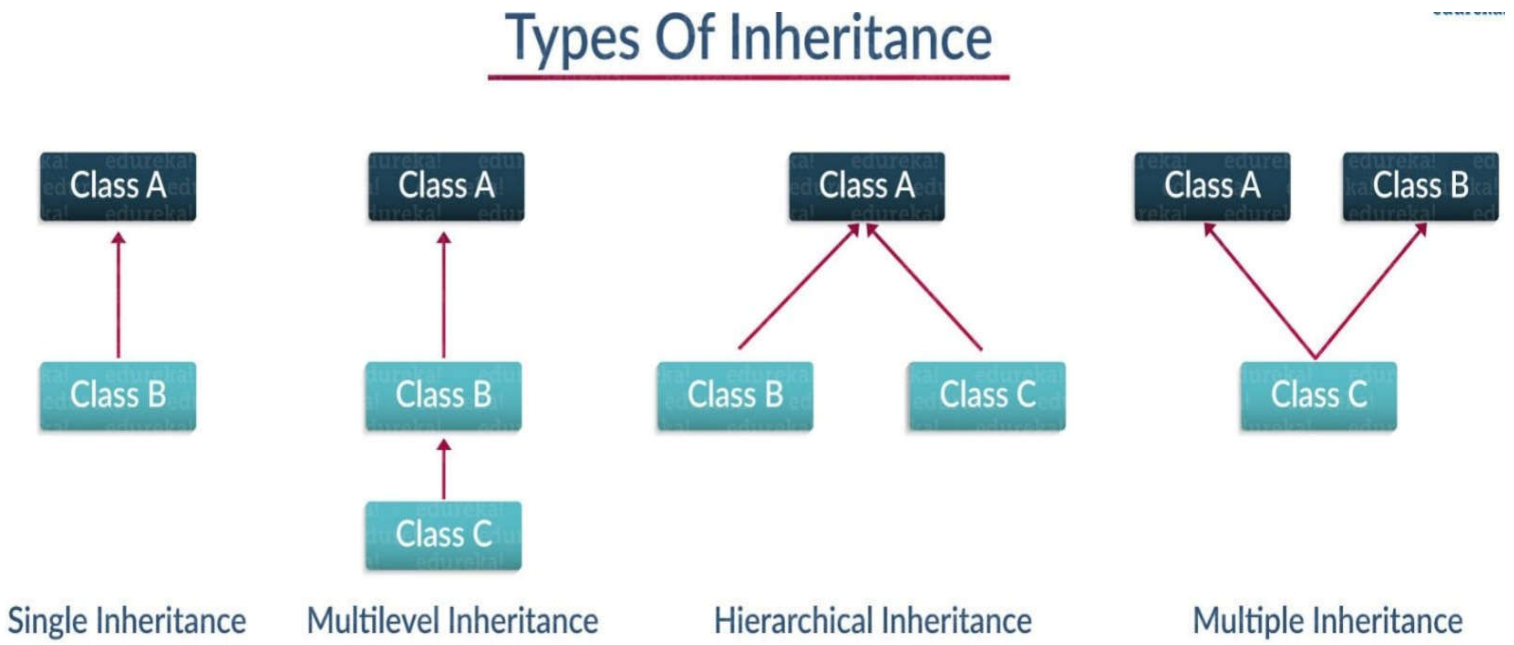
Inheritance is the capability of one class to derive or inherit the properties from another class. The class that derives properties is called the derived class or child class and the class from which the properties are being derived is called the base class or parent class. The benefits of inheritance are:

- It represents real-world relationships well.

- It provides the reusability of a code. We don’t have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.

- It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

Types of Inheritance



Single Inheritance:

Single-level inheritance enables a derived class to inherit characteristics from a single-parent class.

Multilevel Inheritance:

Multi-level inheritance enables a derived class to inherit properties from an immediate parent class which in turn inherits properties from his parent class.

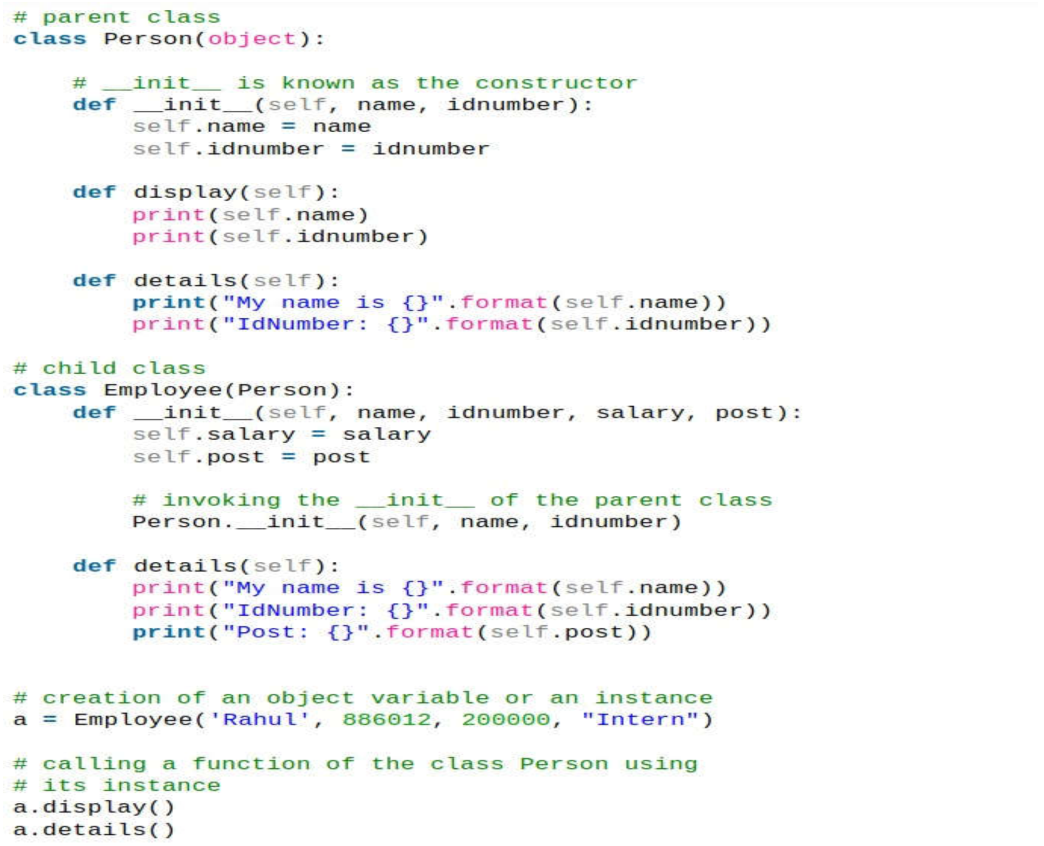
Hierarchical Inheritance:

Hierarchical level inheritance enables more than one derived class to inherit properties from a parent class.

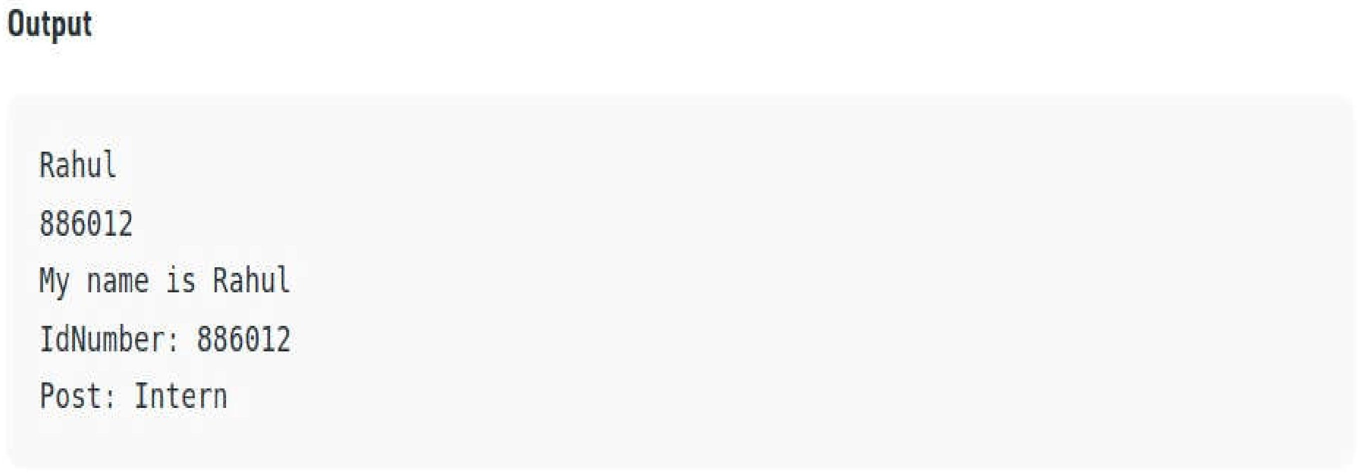
Multiple Inheritance:

Multiple level inheritance enables one derived class to inherit properties from more than one base class.

Example: Inheritance in Python



```python
# parent class
class Person(object):

    # __init__ is known as the constructor
    def __init__(self, name, idnumber):
        self.name = name
        self.idnumber = idnumber

    def display(self):
        print(self.name)
        print(self.idnumber)

    def details(self):
        print("My name is {}".format(self.name))
        print("IdNumber: {}".format(self.idnumber))

# child class
class Employee(Person):
    def __init__(self, name, idnumber, salary, post):
        self.salary = salary
        self.post = post

        # invoking the __init__ of the parent class
        Person.__init__(self, name, idnumber)

    def details(self):
        print("My name is {}".format(self.name))
        print("IdNumber: {}".format(self.idnumber))
        print("Post: {}".format(self.post))


# creation of an object variable or an instance
a = Employee('Rahul', 886012, 200000, "Intern")

# calling a function of the class Person using
# its instance
a.display()
a.details()
```

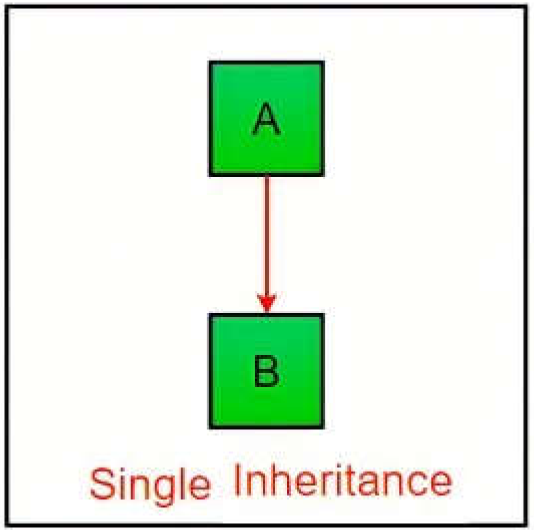


Output

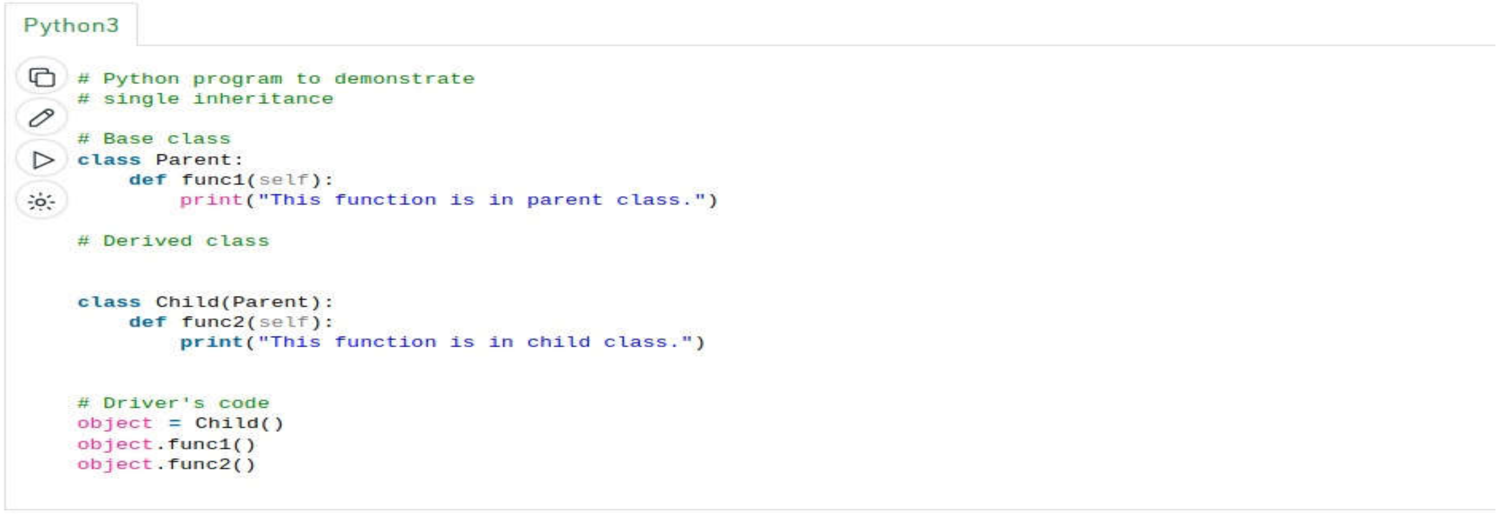
```
Rahul

886012

My name is Rahul

IdNumber: 886012

Post: Intern
```

In the above article, we have created two classes i.e. Person (parent class) and Employee (Child Class). The Employee class inherits from the Person class. We can use the methods of the person class through employee class as seen in the display function in the above code. A child class can also modify the behavior of the parent class as seen through the details() method.

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.



Python3

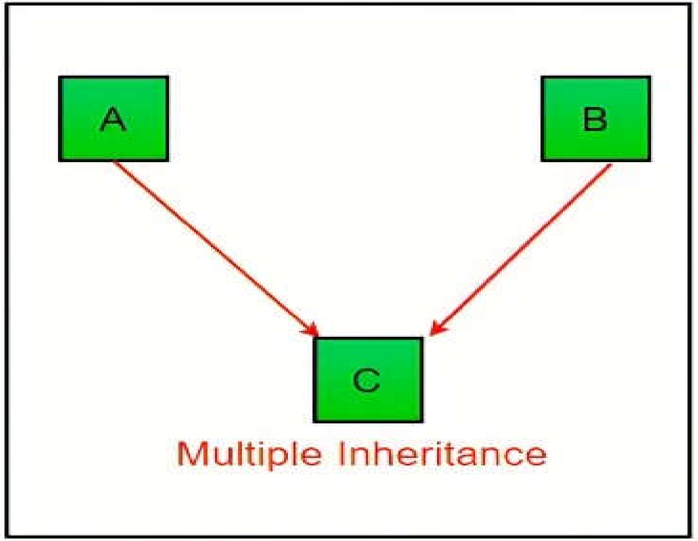
```python
# Python program to demonstrate
# single inheritance

# 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()
object.func2()
```

Output:

```
This function is in parent class.
This function is in child class.
```

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.



Python3

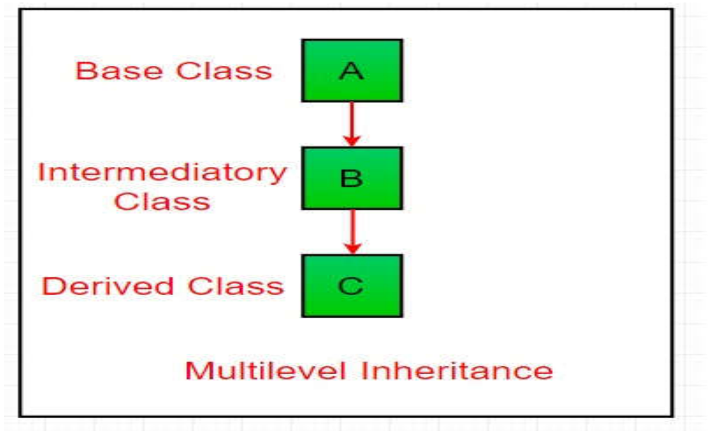
```python
# Python program to demonstrate
# single inheritance

# 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()
object.func2()
```

Output:

```
This function is in parent class.
This function is in child class.
```

Python Multilevel Inheritance:

The property of acquiring all the properties and behaviors of the parent object by an object is termed as Python inheritance. Python facilitates inheritance of a derived class from its base class as well as inheritance of a derived class from another derived class. The inheritance of a derived class from another derived class is called as multilevel inheritance in Python, which is possible to any level.



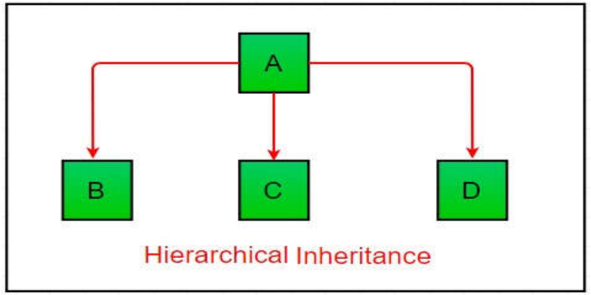
```python
# Python program to demonstrate
# multilevel inheritance

# 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 name :', self.grandfathername)
        print("Father name :", self.fathername)
        print("Son name :", self.sonname)


#  Driver code
s1 = Son('Prince', 'Rampal', 'Lal mani')
print(s1.grandfathername)
s1.print_name()
```

Output:

```
Lal mani
Grandfather name : Lal mani
Father name : Rampal
Son name : Prince
```

Hierarchical Inheritance:

When more than one derived class are 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.



Python3

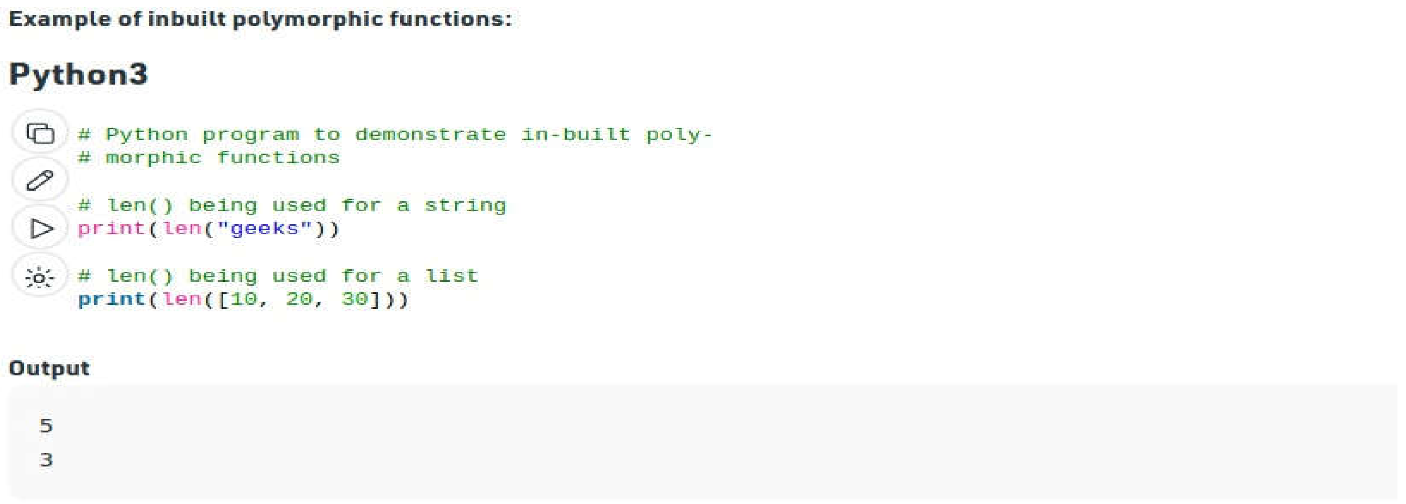
```python
# Python program to demonstrate
# Hierarchical inheritance


# 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.")


# Driver's code
object1 = Child1()
object2 = Child2()
object1.func1()
object1.func2()
object2.func1()
object2.func3()
```

Output:

```
This function is in parent class.
This function is in child 1.
This function is in parent class.
This function is in child 2.
```

What is Polymorphism:

The word polymorphism means having many forms. In programming, polymorphism means the same function name (but different signatures) being used for different types. The key difference is the data types and number of arguments used in function.



Example of inbuilt polymorphic functions:

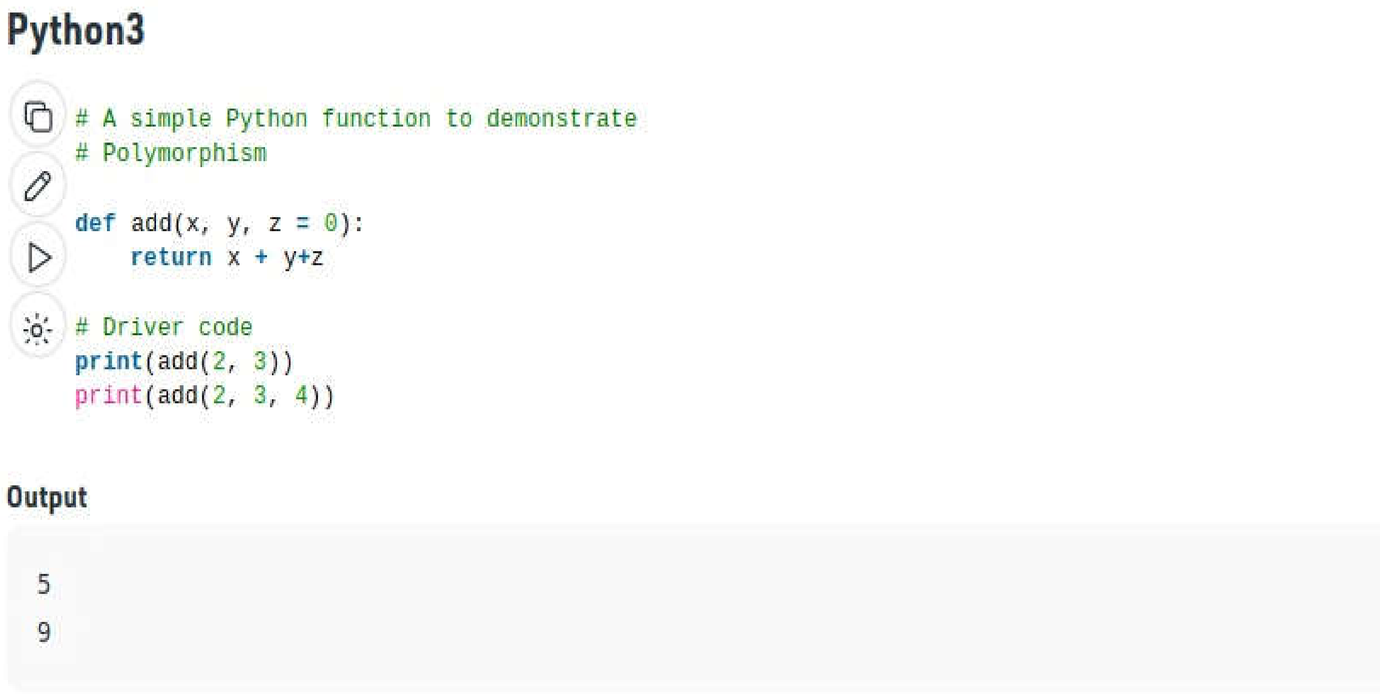
Python3

```python
# Python program to demonstrate in-built poly-
# morphic functions

# len() being used for a string
print(len("geeks"))

# len() being used for a list
print(len([10, 20, 30]))
```

Output

```
5
3
```

Examples of user-defined polymorphic functions:



Python3

```python
# A simple Python function to demonstrate
# Polymorphism

def add(x, y, z = 0):
    return x + y+z

# Driver code
print(add(2, 3))
print(add(2, 3, 4))
```

Output

```
5
9
```

Polymorphism with class methods:

The below code shows how Python can use two different class types, in the same way. We create a for loop that iterates through a tuple of objects. Then call the methods without being concerned about which class type each object is. We assume that these methods actually exist in each class.

Python3

```python
class India():
    def capital(self):
        print("New Delhi is the capital of India.")

    def language(self):
        print("Hindi is the most widely spoken language of India.")

    def type(self):
        print("India is a developing country.")

class USA():
    def capital(self):
        print("Washington, D.C. is the capital of USA.")

    def language(self):
        print("English is the primary language of USA.")

    def type(self):
        print("USA is a developed country.")

obj_ind = India()
obj_usa = USA()
for country in (obj_ind, obj_usa):
    country.capital()
    country.language()
    country.type()
```

Output

```
New Delhi is the capital of India.
Hindi is the most widely spoken language of India.
India is a developing country.
Washington, D.C. is the capital of USA.
English is the primary language of USA.
USA is a developed country.
```

Polymorphism with Inheritance:

In Python, Polymorphism lets us define methods in the child class that have the same name as the methods in the parent class. In inheritance, the child class inherits the methods from the parent class. However, it is possible to modify a method in a child class that it has inherited from the parent class. This is particularly useful in cases where the method inherited from the parent class doesn't quite fit the child class. In such cases, we re-implement the method in the child class. This process of re-implementing a method in the child class is known as Method Overriding.

Python3

```python
class Bird:
  def intro(self):
    print("There are many types of birds.")

  def flight(self):
    print("Most of the birds can fly but some cannot.")

class sparrow(Bird):
  def flight(self):
    print("Sparrows can fly.")

class ostrich(Bird):
  def flight(self):
    print("Ostriches cannot fly.")

obj_bird = Bird()
obj_spr = sparrow()
obj_ost = ostrich()

obj_bird.intro()
obj_bird.flight()

obj_spr.intro()
obj_spr.flight()

obj_ost.intro()
obj_ost.flight()
```

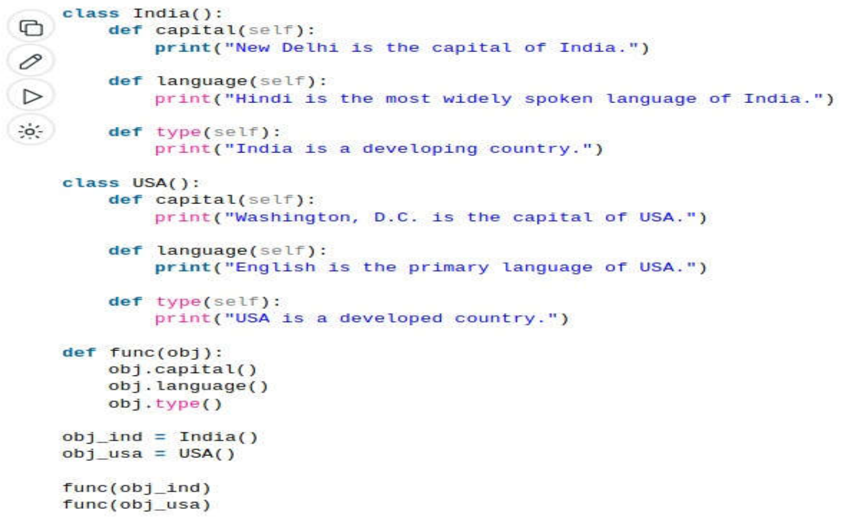
Output

```
There are many types of birds.
Most of the birds can fly but some cannot.
There are many types of birds.
Sparrows can fly.
There are many types of birds.
Ostriches cannot fly.
```

Polymorphism with a Function and objects:

It is also possible to create a function that can take any object, allowing for polymorphism. In this example, let’s create a function called “func()” which will take an object which we will name “obj”. Though we are using the name ‘obj’, any instantiated object will be able to be called into this function. Next, let’s give the function something to do that uses the ‘obj’ object we passed to it. In this case, let’s call the three methods, viz., capital(), language() and type(), each of which is defined in the two classes ‘India’ and ‘USA’. Next, let’s create instantiations of both the ‘India’ and ‘USA’ classes if we don’t have them already. With those, we can call their action using the same func() function:

Code: Implementing Polymorphism with a Function

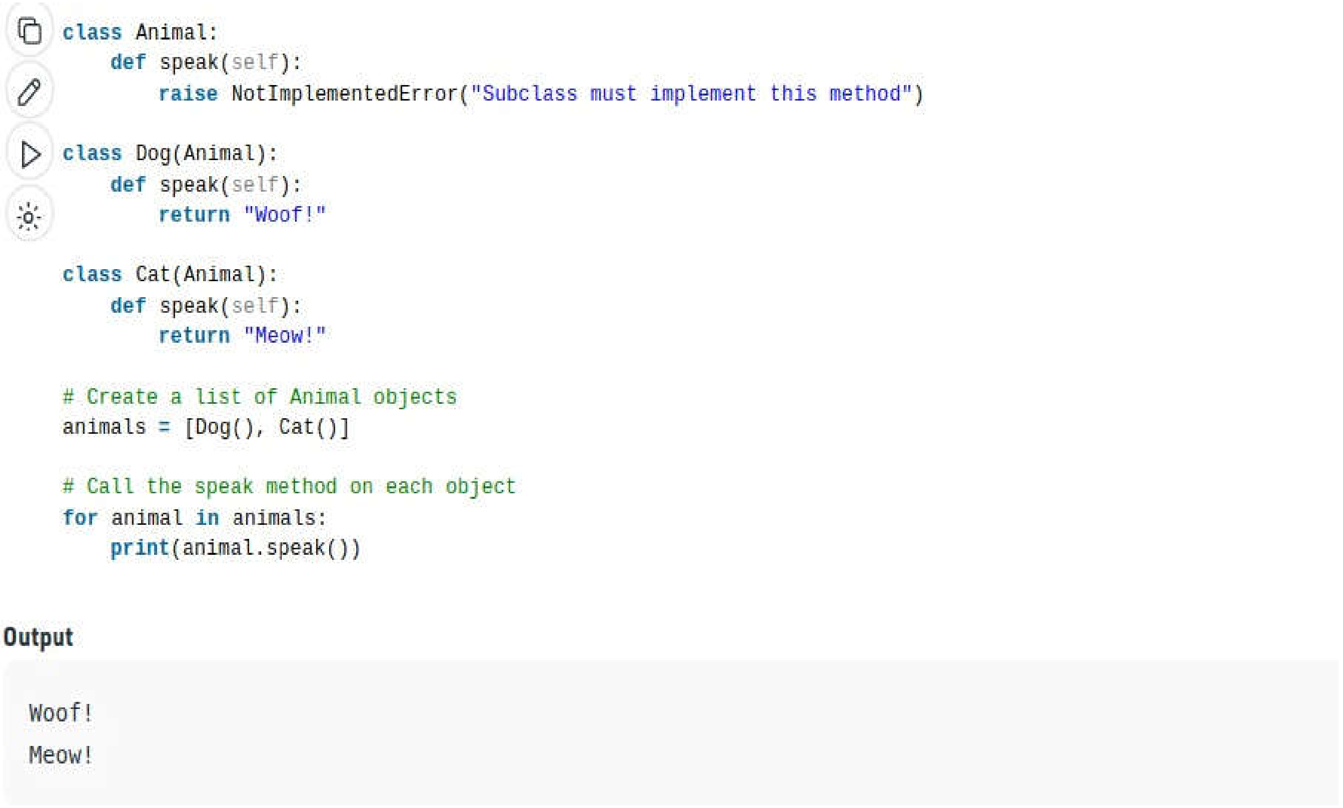


```python
class India():
    def capital(self):
        print("New Delhi is the capital of India.")

    def language(self):
        print("Hindi is the most widely spoken language of India.")

    def type(self):
        print("India is a developing country.")

class USA():
    def capital(self):
        print("Washington, D.C. is the capital of USA.")

    def language(self):
        print("English is the primary language of USA.")

    def type(self):
        print("USA is a developed country.")

def func(obj):
    obj.capital()
    obj.language()
    obj.type()

obj_ind = India()
obj_usa = USA()

func(obj_ind)
func(obj_usa)
```

Output

```
New Delhi is the capital of India.
Hindi is the most widely spoken language of India.
India is a developing country.
Washington, D.C. is the capital of USA.
English is the primary language of USA.
USA is a developed country.
```

polymorphism in Python using inheritance and method overriding:



```
class Animal:
    def speak(self):
        raise NotImplementedError("Subclass must implement this method")

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"

# Create a list of Animal objects
animals = [Dog(), Cat()]

# Call the speak method on each object
for animal in animals:
    print(animal.speak())
```

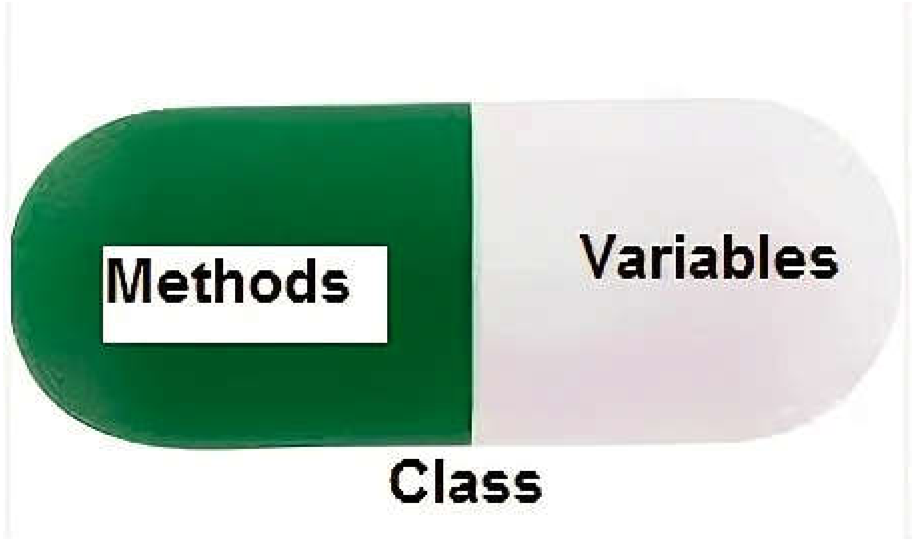
Output

```
Woof!
Meow!
```

Encapsulation in Python:

Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It describes the idea of wrapping data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data. To prevent accidental change, an object's variable can only be changed by an object's method. Those types of variables are known as private variables.

A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc. The goal of information hiding is to ensure that an object's state is always valid by controlling access to attributes that are hidden from the outside world.



Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section etc. The finance section handles all the financial transactions and keeps records of all the data related to finance.

Similarly, the sales section handles all the sales-related activities and keeps records of all the sales. Now there may arise a situation when due to some reason an official from the finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name “sales section”. Using encapsulation also hides the data. In this example, the data of the sections like sales, finance, or accounts are hidden from any other section.

Protected members

Protected members (in C++ and JAVA) are those members of the class that cannot be accessed outside the class but can be accessed from within the class and its subclasses. To accomplish this in Python, just follow the convention by prefixing the name of the member by a single underscore “_”.

Although the protected variable can be accessed out of the class as well as in the derived class (modified too in derived class), it is customary(convention not a rule) to not access the protected out the class body.

Note: The _init_ method is a constructor and runs as soon as an object of a class is instantiated.

Python3

```python
# Python program to
# demonstrate protected members

# Creating a base class
class Base:
    def __init__(self):

        # Protected member
        self._a = 2

# Creating a derived class
class Derived(Base):
    def __init__(self):

        # Calling constructor of
        # Base class
        Base.__init__(self)
        print("Calling protected member of base class: ",
              self._a)

        # Modify the protected variable:
        self._a = 3
        print("Calling modified protected member outside class: ",
              self._a)


obj1 = Derived()

obj2 = Base()

# Calling protected member
# Can be accessed but should not be done due to convention
print("Accessing protected member of obj1: ", obj1._a)

# Accessing the protected variable outside
print("Accessing protected member of obj2: ", obj2._a)
```

Output:

```
Calling protected member of base class:  2
Calling modified protected member outside class:  3
Accessing protected member of obj1:  3
Accessing protected member of obj2:  2
```

Private members

Private members are similar to protected members, the difference is that the class members declared private should neither be accessed outside the class nor by any base class. In Python, there is no existence of Private instance variables that cannot be accessed except inside a class.

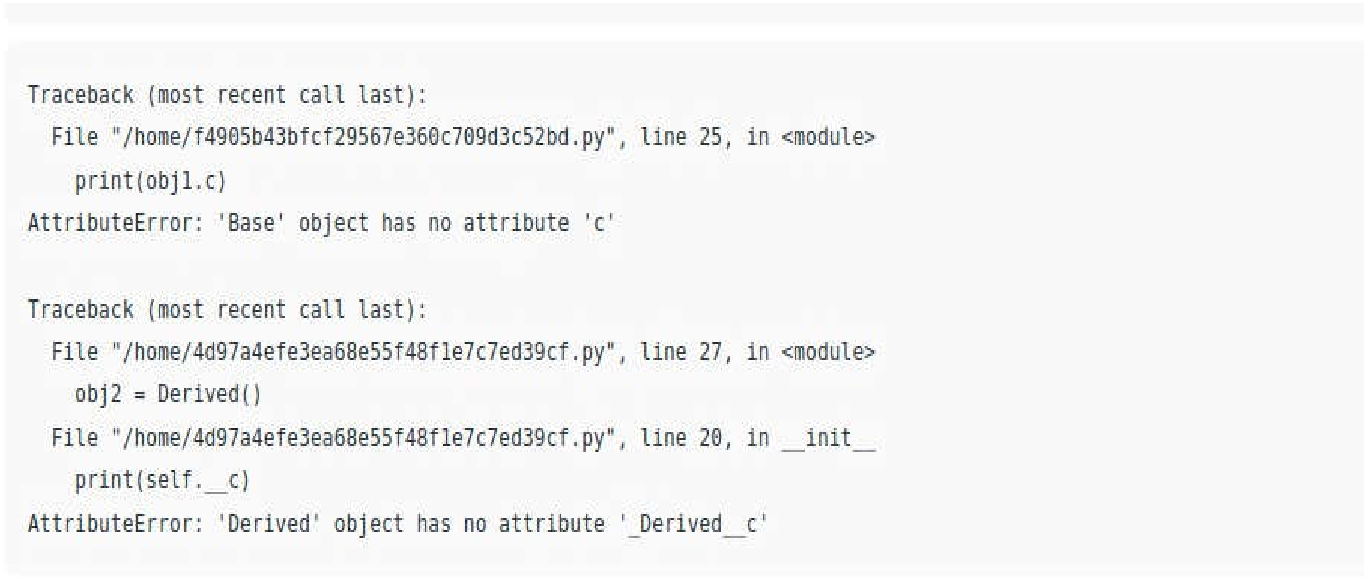
However, to define a private member prefix the member name with double underscore “__”.

Python3

```python
# Python program to
# demonstrate private members

# Creating a Base class


class Base:
    def __init__(self):
        self.a = "GeeksforGeeks"
        self.__c = "GeeksforGeeks"

# Creating a derived class
class Derived(Base):
    def __init__(self):

        # Calling constructor of
        # Base class
        Base.__init__(self)
        print("Calling private member of base class: ")
        print(self.__c)


# Driver code
obj1 = Base()
print(obj1.a)

# Uncommenting print(obj1.c) will
# raise an AttributeError

# Uncommenting obj2 = Derived() will
# also raise an AtrributeError as
# private member of base class
# is called inside derived class
```

Output:

```
GeeksforGeeks
```



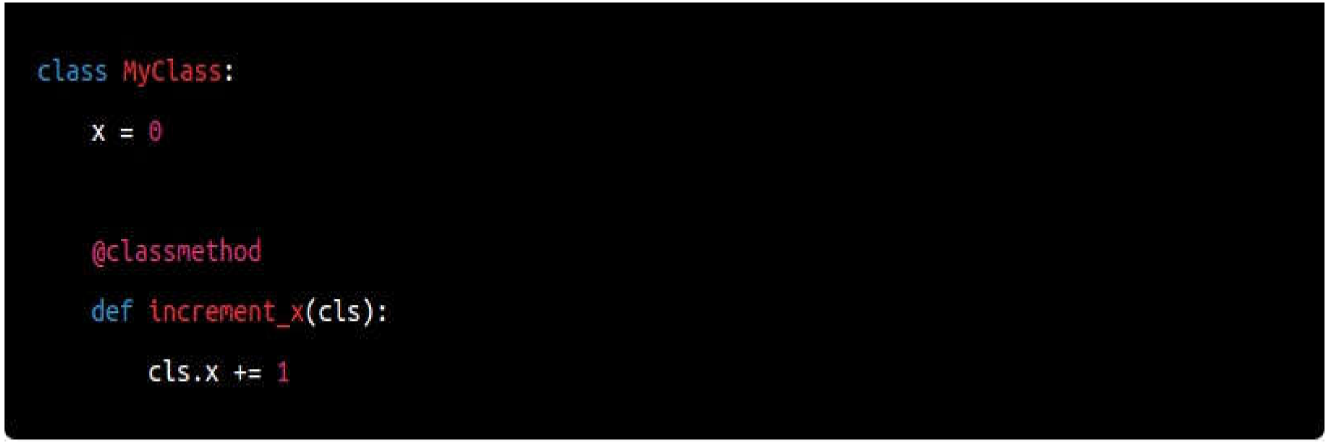
```
Traceback (most recent call last):
  File "/home/f4905b43bfcf29567e360c709d3c52bd.py", line 25, in <module>
    print(obj1.c)
AttributeError: 'Base' object has no attribute 'c'

Traceback (most recent call last):
  File "/home/4d97a4efe3ea68e55f48f1e7c7ed39cf.py", line 27, in <module>
    obj2 = Derived()
  File "/home/4d97a4efe3ea68e55f48f1e7c7ed39cf.py", line 20, in __init__
    print(self.__c)
AttributeError: 'Derived' object has no attribute '_Derived__c'
```

In Python, both class methods and static methods are used to define methods that belong to a class rather than to an instance of a class. However, they have different uses and benefits.

Class Method:

A class method is a method that is bound to the class and not the instance of the class. The first argument of a class method is always the class itself, conventionally called 'cls'. Class methods are typically used to modify class-level attributes and perform operations that do not require access to instance-level attributes

Here's an example of a class method:



```
class MyClass:
    x = 0

    @classmethod
    def increment_x(cls):
        cls.x += 1
```

In the example above, increment_x is a class method. It takes the class itself as the first argument and increments the class-level attribute x.

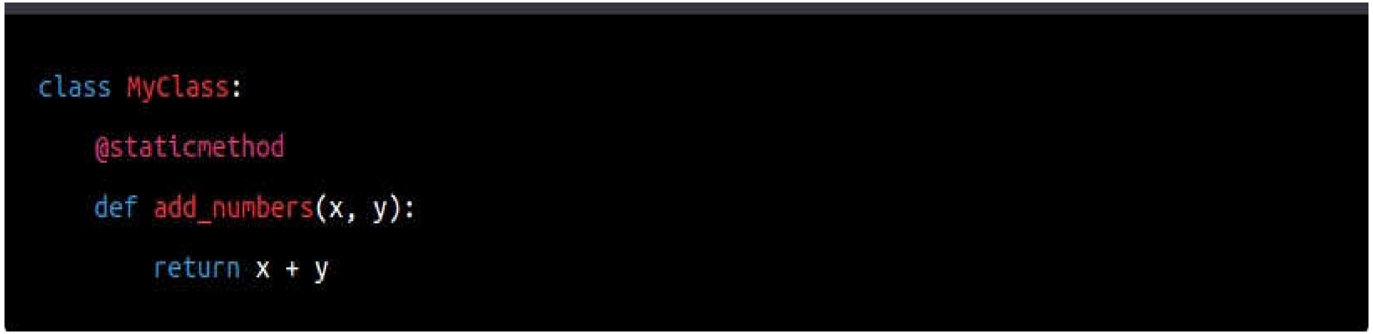
Benefits of class methods:

- Can modify class-level attributes
- Can be used as alternative constructors, allowing for more flexible object creation
- Can be overridden by subclasses

Static Method:

A static method is a method that is bound to the class and not the instance of the class, but does not take the class itself as an argument. Static methods are typically used for utility functions that do not require access to class or instance-level attributes.

Here's an example of a static method:



```
class MyClass:
    @staticmethod
    def add_numbers(x, y):
        return x + y
```

In the example above, add_numbers is a static method. It takes two arguments, x and y, and simply returns their sum.

Benefits of static methods:

- Do not require access to class or instance-level attributes

- Can be called on the class or an instance of the class

In summary, class methods and static methods provide different functionality and have different use cases in Python. Class methods are useful for modifying class-level attributes and creating alternative constructors, while static methods are useful for utility functions that do not require access to class or instance-level attributes.

Lab 3

Question 1:

Create a class Rectangle with two attributes width and height. The class should have a method area() that returns the area of the rectangle.

Question 2:

Create a class Circle with one attribute radius. The class should have a method circumference() that returns the circumference of the circle.

Question 3:

Create a class Employee with three attributes name, age, and salary. The class should have a method raise_salary() that increases the employee's salary by a given percentage.

Question 4:

Create a class Book with two attributes title and author. The class should have a method display() that prints the title and author of the book.

Question 5:

Create a class Car with four attributes make, model, year, and mileage. The class should have a method drive() that increments the mileage of the car by a given amount.

Question 6:

Create a class Person with two attributes name and age. The class should have a constructor that initializes the name and age attributes. Also, implement a destructor for the class that prints a message when the object is destroyed.

Question 7:

Create a class BankAccount with two attributes account_number and balance. The class should have a constructor that initializes the account_number and balance attributes. Also, implement a destructor for the class that prints a message when the object is destroyed.

Question 8:

[Single Inheritance] Create a class Vehicle with an attribute speed. Create a subclass Car that inherits from Vehicle and has an attribute brand. Implement a method in Car that returns the brand and speed of the car.

Question 9:

[Multiple inheritance] Create a class Animal with an attribute name. Create a class Pet with an attribute owner.

Question 10:

Create a subclass Dog that inherits from both Animal and Pet and has an attribute breed. Implement a method in Dog that returns the name, owner, and breed of the dog.

Question 11:

[Multilevel inheritance] Create a class Person with an attribute name. Create a subclass Employee that inherits from Person and has an attribute salary. Create a subclass Manager that inherits from Employee and has an attribute department. Implement a method in Manager that returns the name, salary, and department of the manager.

Question 12:

[Hierarchical Inheritance] Create a class Shape with an attribute color. Create a subclass Rectangle that inherits from Shape and has attributes width and height.

Question 13:

Create a subclass Circle that inherits from Shape and has an attribute radius. Implement a method in each subclass that returns the area of the shape.

Question 14:

[Encapsulation] Create a class BankAccount with a private attribute balance. Implement methods deposit and withdraw to modify the balance, and a method get_balance to retrieve the balance.

Question 15:

[Polymorphism] Create a class Animal with an abstract method speak(). Implement subclasses Dog and Cat that override speak() to output the respective animal sounds.

Question 16:

Create a class Calculator with a class method add that takes two numbers as arguments and returns their sum.

Thank you

Hesham Sayed

E-mail: hesham.sayed636@gmail.com