Object oriented thinking

* Many of today's most popular programming languages are object oriented, but that's not the only way to program. To understand when and why using object oriented languages can be beneficial, it helps to compare it with a common alternative. Procedural programming languages like plain C.
* In procedural code, the program is written as a long series of operations to execute.
* Now, some of that might be organized intonamed functions or sub-routines to make the code modular and maintainable, but the end goal is really just to get from Point A to Point B to complete some task.
* It's a straight forward approach that I like to relate to writing a recipe for a cookbook.
* The program or recipe to say, bake a cake, would list the sequence of steps you need to follow.
* Mix the ingredients together, pour them into a cake pan, and put it in the oven. Just execute those steps in that order and voila, a cake.
* I've found that new programmers have a tendency to write code in this procedural manner because it's easy to think of simple programs in terms of sequential steps. –
* Right, and to approach that same task of baking a cake in an object oriented mannerrather than describing a sequence of steps, I'll describe each of the objects in my kitchen, the pan, the oven, and the mixer, and what each one can do.
* So, instead of writing a single large program, my object oriented code is split apart into several self contained objects.
* The idea here is that we can talk about and use these programmed objects similar to objects in the real world.
* The mixer can mix ingredients together. I can pour the mix into the pan and the oven can bake whatever I give it.
* One of the main advantages of using an object oriented approach is code re-usability. If we want to make something other than a cake,perhaps muffins, I've already created the functionality to mix and bake things in the mixer and the oven. So I can reuse those objects.

Object-oriented Programming, or OOP for short, is a [programming paradigm](http://en.wikipedia.org/wiki/Programming_paradigm) which provides a means of structuring programs so that properties and behaviors are bundled into individual objects.

### **What are Classes and Objects?**

 class is a collection of objects or you can say it is a blueprint of objects defining the common attributes and behavior. Now the question arises, how do you do that?

Well, it logically groups the data in such a way that code reusability becomes easy. I can give you a real-life example- think of an office going ’employee’ as a class and all the attributes related to it like ’emp\_name’, ’emp\_age’, ’emp\_salary’, ’emp\_id’ as the objects in [Python](https://www.edureka.co/blog/python-programming-language). Let us see from the coding perspective that how do you instantiate a class and an object.

Class is defined under a “Class” Keyword.

**Example:**

|  |  |
| --- | --- |
| 1 | class class1(): // class 1 is the name of the class |

### **Objects:**

Objects are an instance of a class. It is an entity that has state and behavior. In a nutshell, it is an instance of a class that can access the data.

**Syntax:**obj = class1()

Here obj is the “object “ of class1.

# **Python Class and Objects**

* A class is a virtual entity and can be seen as a blueprint or template of an object.
* The class came into existence when it instantiated. Let's understand it by an example.
* Suppose a class is a prototype of a building. A building contains all the details about the floor, doors, windows, etc. we can make as many buildings as we want, based on these details. Hence, the building can be seen as a class, and we can create as many objects of this class.
* On the other hand, the object is the instance of a class. The process of creating an object can be called as instantiation.
* Every object has state and behaviour
* State (properties, characteristics, attributes, data, variables)
* Behaviour (action that an object does). Object shows behaviour based on state.
* Beaviour is defined by methods (not functions)
* Explain class and object with particular phone. (Design is class, instance is object)
* Any number of objects can be created from the class.

### **Creating an Object and Class in python:**

**class** employee():  
 **def** \_\_init\_\_(self,name,age,id,salary):   
 self.name = name  
 self.age = age  
 self.salary = salary  
 self.id = id  
   
emp1 = employee(**"harshit"**,22,1000,1234)  
emp2 = employee(**"arjun"**,23,2000,2234)  
print(emp1.\_\_dict\_\_)

**Explanation:**’emp1′ and ’emp2′ are the objects that are instantiated against the class ’employee’.Here, the word (\_\_dict\_\_) is a “dictionary” which prints all the values of object ‘emp1’ against the given parameter (name, age, salary).(\_\_init\_\_) acts like a constructor that is invoked whenever an object is created.

### **Instance Attributes**

class Dog:

# Initializer / Instance Attributes

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

In the case of our Dog() class, each dog has a specific name and age, which is obviously important to know for when you start actually creating different dogs. Remember: the class is just for defining the Dog, not actually creating instances of individual dogs with specific names and ages; we’ll get to that shortly.

Similarly, the self variable is also an instance of the class

### **Class Attributes**

While instance attributes are specific to each object, class attributes are the same for all instances—which in this case is all dogs.

class Dog:

# Class Attribute

species = 'mammal'

# Initializer / Instance Attributes

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

So while each dog has a unique name and age, every dog will be a mammal.

Let’s create some dogs…

## Instantiating Objects

class Dog:

# Class Attribute

species = 'mammal'

# Initializer / Instance Attributes

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

# Instantiate the Dog object

philo = Dog("Philo", 5)

mikey = Dog("Mikey", 6)

# Access the instance attributes

print("{} is {} and {} is {}.".format(

philo.name, philo.age, mikey.name, mikey.age))

# Is Philo a mammal?

if philo.species == "mammal":

print("{0} is a {1}!".format(philo.name, philo.species))

## Instance Methods

### Instance methods are defined inside a class and are used to get the contents of an instance.

class Dog:

# Class Attribute

species = 'mammal'

# Initializer / Instance Attributes

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

# instance method

def description(self):

return "{} is {} years old".format(self.name, self.age)

# instance method

def speak(self, sound):

return "{} says {}".format(self.name, sound)

# Instantiate the Dog object

mikey = Dog("Mikey", 6)

# call our instance methods

print(mikey.description())

print(mikey.speak("Gruff Gruff"))

…………………………………………..

## The 3 Types of Methods in Python

## Instance Methods

Instance methods are the most common type of methods in Python classes. These are so called because they can access unique data of their instance. If you have two objects each created from a car class, then they each may have different properties. They may have different colors, engine sizes, seats, and so on.

Instance methods must have **self** as a parameter

Finally, as instance methods are the most common, there’s no decorator needed.

Any method you create will automatically be created as an instance method, unless you tell Python otherwise.

Here’s an example:

class DecoratorExample:

""" Example Class """

def \_\_init\_\_(self):

""" Example Setup """

print('Hello, World!')

self.name = 'Decorator\_Example'

def example\_function(self):

""" This method is an instance method! """

print('I\'m an instance method!')

print('My name is ' + self.name)

de = DecoratorExample()

de.example\_function()

The **name** variable is accessed through **self**. Notice that when **example\_function** is called, you don’t have to pass self in—Python does this for you.

## Static Methods in Python

Static methods are methods that are related to a class in some way, but don’t need to access any class-specific data. You don’t have to use **self**, and you don’t even need to instantiate an instance, you can simply call your method:

class DecoratorExample:

""" Example Class """

def \_\_init\_\_(self):

""" Example Setup """

print('Hello, World!')

@staticmethod

def example\_function():

""" This method is a static method! """

print('I\'m a static method!')

de = DecoratorExample.example\_function()

The **@staticmethod** decorator was used to tell Python that this method is a static method.

Static methods are great for utility functions, which perform a task in isolation. They don’t need to (and cannot) access class data. They should be completely self-contained, and only work with data passed in as arguments. You may use a static method to add two numbers together, or print a given string.

## Class Methods in Python

Class methods know about their class. They can’t access specific instance data, but they can call other static methods.

Class methods don’t need **self** as an argument, but they do need a parameter called **cls**. This stands for **class**, and like self, gets automatically passed in by Python.

Class methods are created using the **@classmethod** decorator.

class DecoratorExample:

""" Example Class """

def \_\_init\_\_(self):

""" Example Setup """

print('Hello, World!')

@classmethod

def example\_function(cls):

""" This method is a class method! """

print('I\'m a class method!')

cls.some\_other\_function()

@staticmethod

def some\_other\_function():

print('Hello!')

de = DecoratorExample()

de.example\_function()

Class methods are possibly the more confusing method types of the three, but they do have their uses. Class methods can manipulate the class itself, which is useful when you’re working on larger, more complex projects.

…………………………………………………..

### Access Modifiers in Python

…………………………………………………………………………………………………………

* The access modifiers in Python are used to modify the default scope of variables. There are three types of access modifiers in Python: public, private, and protected
* Variables with the public access modifiers can be accessed anywhere inside or outside the class,
* the private variables can only be accessed inside the class,
* while protected variables can be accessed within the same package
* To create a private variable, you need to prefix double underscores with the name of the variable
* To create a protected variable, you need to prefix a single underscore with the variable name
* For public variables, you do not have to add any prefixes at all.

**class** Car:  
 **def** \_\_init\_\_(self):  
 print (**"Engine started"**)  
 self.name = **"corolla"** self.\_\_make = **"toyota"** self.\_model = 1999

* In the script above, we create a simple Car class with a constructor and three variables name, make, and model. The name variable is public while the make and model variables have been declared private and protected, respectively.
* Let's create an object of the Car class and try to access the name variable. Execute the following script:

car\_a = Car() print(car\_a.name)

* Since name is a public variable, therefore we can access it outside the class. In the output, you will see the value for the name printed on the console.
* Now let's try to print the value of the make variable. Execute the following script:

print(car\_a.make)

* In the output, you will see the following error message:

……………………………………………………………..

### OOP Exampe

An example of a class is the class Dog. Don't think of it as a specific dog, or your own dog. We're describing what a dog is and can do, in general. Dogs usually have a name and age; these are instance attributes. Dogs can also bark; this is a method.

When you talk about a specific dog, you would have an object in programming: an object is an instantiation of a class. This is the basic principle on which object-oriented programming is based. So my dog Ozzy, for example, belongs to the class Dog. His attributes are name = 'Ozzy' and age = '2'. A different dog will have different attributes.

### **Object-oriented vs Procedure-oriented Programming languages**

|  |  |  |  |
| --- | --- | --- | --- |
| **Index** | | **Object-oriented Programming** | **Procedural Programming** |
| 1. | Object-oriented programming is the problem-solving approach and used where computation is done by using objects. | | Procedural programming uses a list of  instructions to do computation step by step. |
| 2. | It makes the development and maintenance easier. | | In procedural programming,  It is not easy to maintain the codes  when the project becomes lengthy. |
| 3. | It simulates the real world entity. So real-world problems can be easily solved through oops. | | It doesn't simulate the real world.  It works on step by step instructions  divided into small parts called functions. |
| 4. | It provides data hiding. So it is more secure than procedural languages. You cannot access private data from anywhere. | | Procedural language doesn't provide  any proper way for data binding,  so it is less secure. |
| 5. | Example of object-oriented programming languages is C++, Java, .Net, Python, C#, etc. | | Example of procedural languages are:  C, Fortran, Pascal, VB etc. |

# **Python Constructor**

A constructor is a special type of method (function) which is used to initialize the instance members of the class.

Constructors can be of two types.

1. Parameterized Constructor
2. Non-parameterized Constructor

Constructor definition is executed when we create the object of this class.

## Creating the constructor in python

In python, the method \_\_**init**\_\_ simulates the constructor of the class. This method is called when the class is instantiated. We can pass any number of arguments at the time of creating the class object, depending upon \_\_**init**\_\_ definition. It is mostly used to initialize the class attributes.

Consider the following example to initialize the Employee class attributes.

### **Example**

**class** Employee:   
 **def** \_\_init\_\_(self,name,id):   
 self.id = id;   
 self.name = name;   
 **def** display (self):   
 print(**"ID: %d \nName: %s"**%(self.id,self.name))   
emp1 = Employee(**"John"**,101)   
emp2 = Employee(**"David"**,102)   
   
*#accessing display() method to print employee 1 information*emp1.display();   
   
*#accessing display() method to print employee 2 information*emp2.display()

### **Example: Counting the number of objects of a class**

**class** Student:   
 count = 0   
 **def** \_\_init\_\_(self):   
 Student.count = Student.count + 1   
s1=Student()   
s2=Student()   
s3=Student()   
print(**"The number of students:"**,Student.count)

## Python Non-Parameterized Constructor Example

**class** Student:   
 *# Constructor - non parameterized* **def** \_\_init\_\_(self):   
 print(**"This is non parametrized constructor"**)   
 **def** show(self,name):   
 print(**"Hello"**,name)   
student = Student()   
student.show(**"John"**)

## Python Parameterized Constructor Example

**class** Student:   
 *# Constructor - parameterized* **def** \_\_init\_\_(self, name):   
 print(**"This is parametrized constructor"**)   
 self.name = name   
 **def** show(self):   
 print(**"Hello"**,self.name)   
student = Student(**"John"**)   
student.show()

Python In-built class functions

The in-built functions defined in the class are described in the following table.

|  |  |  |
| --- | --- | --- |
| **SN** | **Function** | **Description** |
| 1 | getattr(obj,name,default) | It is used to access the attribute of the object. |
| 2 | setattr(obj, name,value) | It is used to set a particular value to the  specific attribute of an object. |
| 3 | delattr(obj, name) | It is used to delete a specific attribute. |
| 4 | hasattr(obj, name) | It returns true if the object contains some  specific attribute. |

### **Example**

**class** Student:   
 **def** \_\_init\_\_(self,name,id,age):   
 self.name = name;   
 self.id = id;   
 self.age = age   
   
*#creates the object of the class Student*s = Student(**"John"**,101,22)   
   
*#prints the attribute name of the object s*print(getattr(s,**'name'**))   
   
*# reset the value of attribute age to 23*setattr(s,**"age"**,23)   
   
*# prints the modified value of age*print(getattr(s,**'age'**))   
   
*# prints true if the student contains the attribute with name id*print(hasattr(s,**'id'**))   
*# deletes the attribute age*delattr(s,**'age'**)   
   
*# this will give an error since the attribute age has been deleted*print(s.age)

Built-in class attributes

Along with the other attributes, a python class also contains some built-in class attributes which provide information about the class.

The built-in class attributes are given in the below table.

|  |  |  |
| --- | --- | --- |
| **SN** | **Attribute** | **Description** |
| 1 | \_\_dict\_\_ | It provides the dictionary containing the information  about the class namespace. |
| 2 | \_\_doc\_\_ | It contains a string which has the class documentation |
| 3 | \_\_name\_\_ | It is used to access the class name. |
| 4 | \_\_module\_\_ | It is used to access the module in which, this class is defined. |
| 5 | \_\_bases\_\_ | It contains a tuple including all base classes. |

**class** Student:   
 **def** \_\_init\_\_(self,name,id,age):   
 self.name = name;   
 self.id = id;   
 self.age = age   
 **def** display\_details(self):   
 print(**"Name:%s, ID:%d, age:%d"**%(self.name,self.id))   
s = Student(**"John"**,101,22)   
print(s.\_\_doc\_\_)   
print(s.\_\_dict\_\_)   
print(s.\_\_module\_\_)

…………………………………….

## ****Object-Oriented Programming methodologies:****

Object-Oriented Programming methodologies deal with the following concepts.

* Inheritance
* Polymorphism
* Encapsulation
* Abstraction

**Abstraction:**

* Abstraction is one of the [key concepts](https://stackify.com/oops-concepts-in-java/) of object-oriented programming (OOP) languages. Its main goal is to handle complexity by hiding unnecessary details from the user, showing only the essential features.
* essential features of the object. So in a way, Abstraction means hiding the real implementation and we, as a user, knowing only how to use it.
* Real world example would be a vehicle which we drive with out caring or knowing what all is going underneath.
* A TV set where we enjoy programs with out knowing the inner details of how TV works.

## Encapsulation

* Using OOP in Python, we can restrict access to methods and variables.
* Encapsulation simply refers to data hiding
* As a general principle, in object-oriented programming, one class should not have direct access to the data of the other class. Rather, the access should be controlled via class methods.
* In encapsulation, code and data are wrapped together within a single unit from being modified by accident.
* This prevent data from direct modification which is called encapsulation. In Python, we denote private attribute using underscore as prefix i.e single “ \_ “ or double “ \_\_“.

**class** Computer:  
  
 **def** \_\_init\_\_(self):  
 self.\_\_maxprice = 900  
  
 **def** sell(self):  
 print(**"Selling Price: {}"**.format(self.\_\_maxprice))  
  
 **def** setMaxPrice(self, price):  
 self.\_\_maxprice = price  
  
c = Computer()  
c.sell()  
  
*# change the price*c.\_\_maxprice = 1000  
c.sell()  
  
*# using setter function*c.setMaxPrice(1000)  
c.sell()

When we run this program, the output will be:

Selling Price: 900

Selling Price: 900

Selling Price: 1000

* In the above program, we defined a class Computer. We use \_\_init\_\_() method to store the maximum selling price of computer. We tried to modify the price. However, we can’t change it because Python treats the \_\_maxprice as private attributes. To change the value, we used a setter function i.e setMaxPrice() which takes price as parameter.

### **Encapsulation in Python**

It is the concept of wrapping data such that the outer world has access only to exposed properties. Some properties can be hidden to reduce vulnerability. This is an implementation of data hiding. For example, you want buy a pair of trousers from an online site. The data that you want is its cost and availability. The number of items present and their location is information that you are not bothered about. Hence it is hidden.

In Python this is implemented by creating private, protected and public instance variables and methods.

Private properties have double underscore (\_\_) in the start, while protected properties have single underscore (\_). By default, all other variable and methods are public.

Private properties are accessible from within the class only and are not available for child class(if inherited). Protected properties are accessible from within the class but are available to child class as well. All these restrictions are removed for public properties.

The following code snippets is an example of this concept:

class Person:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

def \_protected\_method(self):

print("protected method")

def \_\_private\_method(self):

print("privated method")

if \_\_name\_\_ == "\_\_main\_\_":

p = Person("mohan", 23)

p.\_protected\_method() # shows a warning

p.\_\_private\_method() # throws Attribute error saying no such method exists

## **What Is Inheritance?**

* Inheritance in object-oriented programming is pretty similar to real-world inheritance where a child inherits some of the characteristics from his parents, in addition to his/her own unique characteristics.
* In object-oriented programming, inheritance signifies an IS-A relation. For instance, a car is a vehicle. Inheritance is one of the most amazing concepts of object-oriented programming as it fosters code re-usability.
* Python Inheritance allows us to define the class that inherits all the methods and properties of another class. **The parent class** is the class being inherited from, also called the base class. **The child class** is the class that inherits from another class, also called derived class.

1. Code reusability- we do not have to write the same code again and again, we can just inherit the properties we need in a child class.
2. It represents a real world relationship between parent class and child class.
3. It is transitive in nature. If a child class inherits properties from a parent class, then all other sub-classes of the child class will also inherit the properties of the parent class.

Let's take a look at a very simple example of inheritance. Execute the following script:

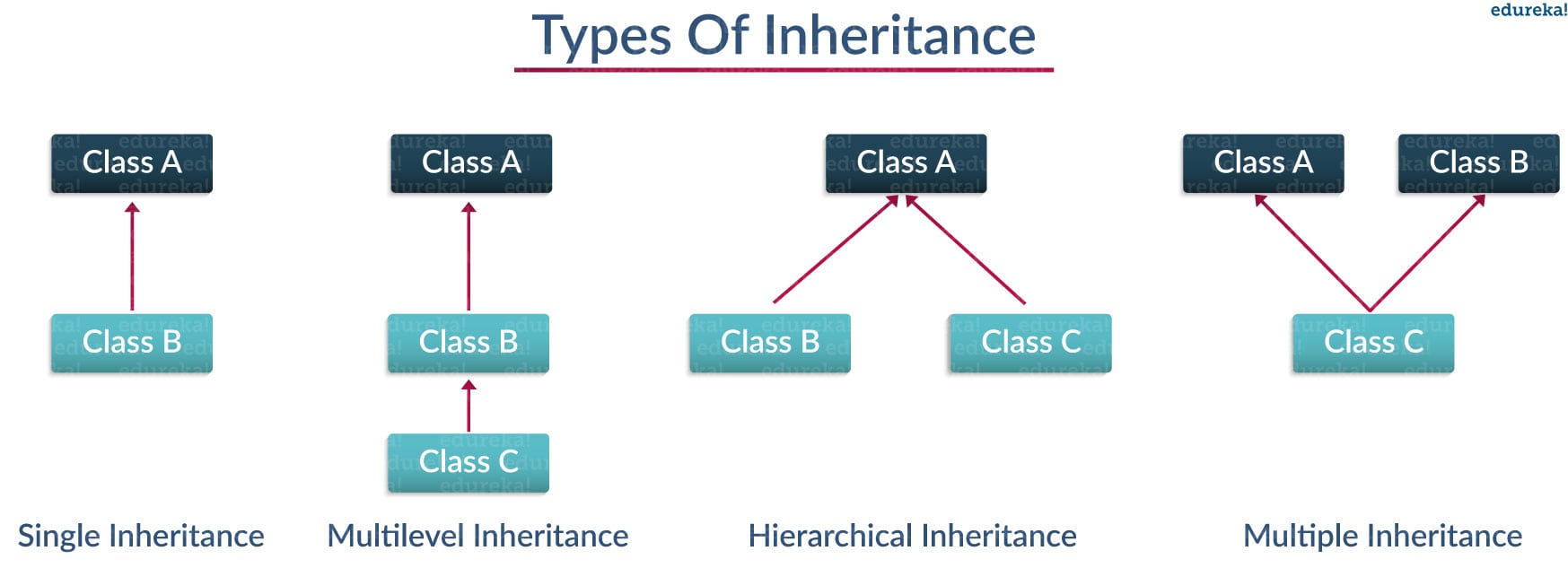
*# Create Class Vehicle***class** Vehicle:  
 **def** vehicle\_method(self):  
 print(**"This is parent Vehicle class method"**)  
  
*# Create Class Car that inherits Vehicle***class** Car(Vehicle):  
 **def** car\_method(self):  
 print(**"This is child Car class method"**)

* In the script above, we create two classes Vehicle class, and the Car class which inherits the Vehicle class. To inherit a class, you simply have to write the parent class name inside the parenthesis that follows the child class name.
* The Vehicle class contains a method vehicle\_method() and the child class contains a method car\_method(). However, since the Carclass inherits the Vehicle class, it will also inherit the vehicle\_method().
* Let's see this in action. Execute the following script:

car\_a = Car()

car\_a.vehicle\_method() *# Calling parent class method*

* In the script above, we create an object of the Car class and call the vehicle\_method() using that Car class object.
* You can see that the Car class doesn't have any vehicle\_method() but since it has inherited the Vehicle class that contains the vehicle\_method(), the car class can also use it. The output looks likes this:



## **Single Inheritance**

## Python Inheritance

When a child class inherits only a single parent class.

### **Syntax**

**class** derived-**class**(base-**class**):

    <**class**-suite>

**class** Parent:  
 **def** func1(self):  
 print(**"this is function one"**)  
**class** Child(Parent):  
 **def** func2(self):  
 print(**" this is function 2 "**)  
ob = Child()  
ob.func1()  
ob.func2()

………………………………………..

**class** Animal:   
 **def** speak(self):   
 print(**"Animal Speaking"**)   
*#child class Dog inherits the base class Animal***class** Dog(Animal):   
 **def** bark(self):   
 print(**"dog barking"**)   
d = Dog()   
d.bark()   
d.speak()

……………………………………………..

## **Multiple Inheritance**

## Python Inheritance

* When a child class inherits from more than one parent class.

### **Syntax**

**class** Base1:

    <**class**-suite>

**class** Base2:

    <**class**-suite>

.

.

.

**class** BaseN:

    <**class**-suite>

**class** Derived(Base1, Base2, ...... BaseN):

    <**class**-suite>

……………………………………………………..

**class** Parent:  
 **def** func1(self):  
 print(**"this is function 1"**)  
**class** Parent2:  
 **def** func2(self):  
 print(**"this is function 2"**)  
**class** Child(Parent , Parent2):  
 **def** func3(self):  
 print(**"this is function 3"**)  
   
ob = Child()  
ob.func1()  
ob.func2()  
ob.func3()  
………………………………………….

**class** Camera:  
 **def** camera\_method(self):  
 print(**"This is parent Camera class method"**)  
  
**class** Radio:  
 **def** radio\_method(self):  
 print(**"This is parent Radio class method"**)  
  
**class** CellPhone(Camera, Radio):  
 **def** cell\_phone\_method(self):  
 print(**"This is child CellPhone class method"**)  
cell\_phone\_a = CellPhone()  
cell\_phone\_a.camera\_method()  
cell\_phone\_a.radio\_method()

……………………………………………

**class** Calculation1:   
 **def** Summation(self,a,b):   
 **return** a+b;   
**class** Calculation2:   
 **def** Multiplication(self,a,b):   
 **return** a\*b;   
**class** Derived(Calculation1,Calculation2):   
 **def** Divide(self,a,b):   
 **return** a/b;   
d = Derived()   
print(d.Summation(10,20))   
print(d.Multiplication(10,20))   
print(d.Divide(10,20))

…………………………………………………….

## **Multilevel Inheritance**

## Python Inheritance

* When a child class becomes a parent class for another child class.
* Multi-level inheritance is archived when a derived class inherits another derived class. There is no limit on the number of levels up to which, the multi-level inheritance is archived in python.

### **Syntax**

**class** class1:

    <**class**-suite>

**class** class2(class1):

    <**class** suite>

**class** class3(class2):

    <**class** suite>

.

.

**class** Parent:  
 **def** func1(self):  
 print(**"this is function 1"**)  
**class** Child(Parent):  
 **def** func2(self):  
 print(**"this is function 2"**)  
**class** Child2(Child):  
 **def** func3(sel):  
 print(**"this is function 3"**)  
ob = Child2()  
ob.func1()  
ob.func2()  
ob.func3()

………………………………………………..

**class** Animal:   
 **def** speak(self):   
 print(**"Animal Speaking"**)   
*#The child class Dog inherits the base class Animal***class** Dog(Animal):   
 **def** bark(self):   
 print(**"dog barking"**)   
*#The child class Dogchild inherits another child class Dog***class** DogChild(Dog):   
 **def** eat(self):   
 print(**"Eating bread..."**)   
d = DogChild()   
d.bark()   
d.speak()   
d.eat()

……………………………………………………..

## **Hierarchical Inheritance**

Hierarchical inheritance involves multiple inheritance from the same base or parent class.

**class** Parent:  
 **def** func1(self):  
 print(**"this is function 1"**)  
**class** Child(Parent):  
 **def** func2(self):  
 print(**"this is function 2"**)  
**class** Child2(Parent):  
 **def** func3(self):  
 print(**"this is function 3"**)  
   
ob = Child()  
ob1 = Child2()  
ob.func1()  
ob.func2()

………………………………………….

*# Create Class Vehicle***class** Vehicle:  
 **def** vehicle\_method(self):  
 print(**"This is parent Vehicle class method"**)  
  
*# Create Class Car that inherits Vehicle***class** Car(Vehicle):  
 **def** car\_method(self):  
 print(**"This is child Car class method"**)  
  
*# Create Class Cycle that inherits Vehicle***class** Cycle(Vehicle):  
 **def** cycleMethod(self):  
 print(**"This is child Cycle class method"**)  
car\_a = Car()  
car\_a.vehicle\_method() *# Calling parent class method*car\_b = Cycle()  
car\_b.vehicle\_method() *# Calling parent class method*

……………………………………………………..

## **Hybrid Inheritance**

Hybrid inheritance involves multiple inheritance taking place in a single program.

……………………………………………….

*# parent class***class** Bird:  
   
 **def** \_\_init\_\_(self):  
 print(**"Bird is ready"**)  
  
 **def** whoisThis(self):  
 print(**"Bird"**)  
  
 **def** swim(self):  
 print(**"Swim faster"**)  
  
*# child class***class** Penguin(Bird):  
  
 **def** \_\_init\_\_(self):  
 *# call super() function* super().\_\_init\_\_()  
 print(**"Penguin is ready"**)  
  
 **def** whoisThis(self):  
 print(**"Penguin"**)  
  
 **def** run(self):  
 print(**"Run faster"**)  
  
peggy = Penguin()  
peggy.whoisThis()  
peggy.swim()  
peggy.run()

Output

Bird is ready

Penguin is ready

Penguin

Swim faster

Run faster

Process finished with exit code 0

……………………………………………………………

## **Python Method Overriding**

**Method Overriding**

You can override a method in python. Look at the example below.

**class** Parent:  
 **def** func1(self):  
 print(**"this is parent function"**)  
   
**class** Child(Parent):  
 **def** func1(self):  
 print(**"this is child function"**)  
   
ob = Child()  
ob.func1()

The functionality of the parent class method is changes by overriding the same method in the child class.

……………………

### Polymorphism

* The term polymorphism literally means having multiple forms. In the context of object-oriented programming, polymorphism refers to the ability of an object to behave in multiple ways.
* Polymorphism in programming is implemented via method-overloading and method overriding.

#### Method Overloading

* Method overloading refers to the property of a method to behave in different ways depending upon the number or types of the parameters. Take a look at a very simple example of method overloading.

*# Creates class Car***class** Car:  
 **def** start(self, a, b=**None**):  
 **if** b **is not None**:  
 print (a + b)  
 **else**:  
 print (a)

* In the script above, if the start() method is called by passing a single argument, the parameter will be printed on the screen. However, if we pass 2 arguments to the start() method, it will add both the arguments and will print the result of the sum.
* Let's try with single argument first:

car\_a = Car()

car\_a.start(10)

* In the output, you will see 10. Now let's try to pass 2 arguments:

car\_a.start(10,20)

* In the output, you will see 30.
* Method overloading
* # class
* class Compute:
* # area method
* def area(self, x = None, y = None):
* if x != None and y != None:
* return x \* y
* elif x != None:
* return x \* x
* else:
* return 0
* # object
* obj = Compute()
* # zero argument
* print("Area:", obj.area())
* # one argument
* print("Area:", obj.area(2))
* # two argument
* print("Area:", obj.area(4, 5))
* The above code will give us the following output.
* Area: 0
* Area: 4
* Area: 20
* In Python we can create a method that can be called in different ways.
* So, we can have a method that has zero, one or more number of parameters and depending on the method definition we can call it with zero, one or more arguments.
* This is method overloading in Python.
* Method overloading in Python is achieved by using one method with different number of arguments.

#### Method Overriding

* We can provide some specific implementation of the parent class method in our child class
* When the parent class method is defined in the child class with some specific implementation, then the concept is called method overriding
* Method overriding refers to having a method with the same name in the child class as in the parent class. The definition of the method differs in parent and child classes but the name remains the same.
* Let's take a simple example method overriding in Python.

……………………….

**class** Animal:   
 **def** speak(self):   
 print(**"speaking"**)   
**class** Dog(Animal):   
 **def** speak(self):   
 print(**"Barking"**)   
d = Dog()   
d.speak()

…………………………………………………

### **Real Life Example of method overriding**

**class** Bank:   
 **def** getroi(self):   
 **return** 10;   
**class** SBI(Bank):   
 **def** getroi(self):   
 **return** 7;   
   
**class** ICICI(Bank):   
 **def** getroi(self):   
 **return** 8;   
b1 = Bank()   
b2 = SBI()   
b3 = ICICI()   
print(**"Bank Rate of interest:"**,b1.getroi());   
print(**"SBI Rate of interest:"**,b2.getroi());   
print(**"ICICI Rate of interest:"**,b3.getroi());

……………………………………………..

*# Create Class Vehicle***class** Vehicle:  
 **def** print\_details(self):  
 print(**"This is parent Vehicle class method"**)  
  
*# Create Class Car that inherits Vehicle***class** Car(Vehicle):  
 **def** print\_details(self):  
 print(**"This is child Car class method"**)  
  
*# Create Class Cycle that inherits Vehicle***class** Cycle(Vehicle):  
 **def** print\_details(self):  
 print(**"This is child Cycle class method"**)

* In the script above the Car and Cycle classes inherit the Vehicle class. The vehicle class has print\_details() method, which is overridden by the child classes.
* Now if you call the print\_details() method, the output will depend upon the object through which the method is being called.

car\_a = Vehicle()

car\_a. print\_details()

car\_b = Car()

car\_b.print\_details()

car\_c = Cycle()

car\_c.print\_details()

**…………………………………………………………………………………………………………………………………………………………………**

EXAMPLES

……………………………………………………………………………………………………………………

WAP to input and print a number

# class definition

class Number:

# \_\_init\_\_ method just like a constructor

def \_\_init\_\_(self, num):

self.num = num;

# method to take input from user

def inputNum(self):

self.num = int(input("Enter an integer number: "))

# method to print the number

def printNum(self):

print "num:", self.num

# main code

# declare object of the class 'Number'

objN = Number(100);

objN.printNum()

# input from user

objN.inputNum()

objN.printNum()

…………………………………………………………………………………………………….

WAP TO demonstrate public variables

*# Python example for public variables***class** person:  
 **def** \_\_init\_\_(self):  
 *# default values* self.name = **"XYZ"** self.age = 0  
  
 **def** printValues(self):  
 print (**"Name: "**,self.name)  
 print (**"Age : "**,self.age)  
  
*# Main code   
# declare object*p = person()  
*# print*p.printValues();  
  
*# since variables are public by default   
# we can access them directly here*p.name = **"Amit"**p.age = 21  
*# print*p.printValues ();

……………………………………………………………………..

WAP to create employee class with some attributes and methods and show data

*# employee class code in Python  
# class definition***class** Employee:  
 \_\_id=0  
 \_\_name=**""** \_\_gender=**""** \_\_city=**""** \_\_salary=0  
   
 *# function to set data* **def** setData(self,id,name,gender,city,salary):  
 self.\_\_id=id  
 self.\_\_name = name  
 self.\_\_gender = gender  
 self.\_\_city = city  
 self.\_\_salary = salary  
   
 *# function to get/print data* **def** showData(self):  
 print(**"Id\t\t:"**,self.\_\_id)  
 print(**"Name\t:"**, self.\_\_name)  
 print(**"Gender\t:"**, self.\_\_gender)  
 print(**"City\t:"**, self.\_\_city)  
 print(**"Salary\t:"**, self.\_\_salary)  
  
  
*#Employee Object*emp=Employee()  
emp.setData(1,**'pankaj'**,**'male'**,**'delhi'**,55000)  
emp.showData()

…………………………………………………………..

WAP to calculate student grade

*# Python code to find student grade***class** Student:  
 **def** \_\_init\_\_(self):  
 self.\_\_roll=0  
 self.\_\_name=**""** self.\_\_marks=[]  
 self.\_\_total=0  
 self.\_\_per=0  
 self.\_\_grade=**""** self.\_\_result=**""  
  
 def** setStudent(self):  
 self.\_\_roll=int(input(**"Enter Roll: "**))  
 self.\_\_name=input(**"Enter Name: "**)  
 print(**"Enter marks of 5 subjects: "**)  
 **for** i **in** range(5):  
 self.\_\_marks.append(int(input(**"Subject "**+str(i+1)+**": "**)))  
  
 **def** calculateTotal(self):  
 **for** x **in** self.\_\_marks:  
 self.\_\_total+=x  
  
 **def** calculatePercentage(self):  
 self.\_\_per=self.\_\_total/5  
  
 **def** calculateGrade(self):  
 **if** self.\_\_per>=85:  
 self.\_\_grade=**"S"  
 elif** self.\_\_per>=75:  
 self.\_\_grade=**"A"  
 elif** self.\_\_per>=65:  
 self.\_\_grade=**"B"  
 elif** self.\_\_per>=55:  
 self.\_\_grade=**"C"  
 elif** self.\_\_per>=50:  
 self.\_\_grade=**"D"  
 else**:  
 self.\_\_grade=**"F"  
  
 def** calculateResult(self):  
 count=0  
 **for** x **in** self.\_\_marks:  
 **if** x>=50:  
 count+=1  
 **if** count==5:  
 self.\_\_result=**"PASS"  
 elif** count>=3:  
 self.\_\_result=**"COMP."  
 else**:  
 self.\_\_result=**"FAIL"  
  
 def** showStudent(self):  
 self.calculateTotal()  
 self.calculatePercentage()  
 self.calculateGrade()  
 self.calculateResult()  
 print(self.\_\_roll,**"\t\t"**,self.\_\_name,**"\t\t"**,self.\_\_total,**"\t\t"**,self.\_\_per,**"\t\t"**,self.\_\_grade,**"\t\t"**,self.\_\_result)  
*#Student object*s=Student()  
s.setStudent()  
s.showStudent()

………………………………………………………

# **Example to implement destructor and constructors using \_\_del\_\_() and \_\_init\_\_()**

**class** Employee:  
 **def** \_\_init\_\_(self): *#Constructor* self.\_\_id = 0  
 self.\_\_name = **""** self.\_\_gender = **""** self.\_\_city = **""** self.\_\_salary = 0  
 print(**"Object Initialized."**)  
 **def** \_\_del\_\_(self): *#Destructor* print(**"Object Destroyed."**)  
 **def** setData(self):  
 self.\_\_id=int(input(**"Enter Id\t:"**))  
 self.\_\_name = input(**"Enter Name\t:"**)  
 self.\_\_gender = input(**"Enter Gender:"**)  
 self.\_\_city = input(**"Enter City\t:"**)  
 self.\_\_salary = int(input(**"Enter Salary:"**))  
 **def** showData(self):  
 print(**"Id\t\t:"**,self.\_\_id)  
 print(**"Name\t:"**, self.\_\_name)  
 print(**"Gender\t:"**, self.\_\_gender)  
 print(**"City\t:"**, self.\_\_city)  
 print(**"Salary\t:"**, self.\_\_salary)  
  
*#Employee Object*emp=Employee()  
*#emp.setData()*emp.showData()

………………………………………………………………….

# **Example to implement Getters and Setters in a class**

In this program, we are implementing **Getters and Setters**. **Getters** are used to access data members so they are also called **accessors** and **Setters** are used to change the data memebers values so they are called **Mutators**.

**class** Employee:  
 **def** \_\_init\_\_(self): *#Constructor* self.\_\_id = 0  
 self.\_\_name = **""** self.\_\_gender = **""** self.\_\_city = **""** self.\_\_salary = 0  
  
 **def** getId(self): *#Accessor/Getters* **return** self.\_\_id  
 **def** setId(self,id): *#Mutators/Setters* self.\_\_id=id  
  
 **def** getName(self):  
 **return** self.\_\_name  
 **def** setName(self,name):  
 self.\_\_name=name  
  
 **def** getGender(self):  
 **return** self.\_\_gender  
 **def** setGender(self,gender):  
 self.\_\_gender=gender  
  
 **def** getCity(self):  
 **return** self.\_\_city  
 **def** setCity(self,city):  
 self.\_\_city=city  
  
 **def** getSalary(self):  
 **return** self.\_\_salary  
 **def** setSalary(self,salary):  
 self.\_\_salary=salary  
  
print(**"Enter Employee Data:"**)  
id = int(input(**"Enter Id\t:"**))  
name = input(**"Enter Name\t:"**)  
gender = input(**"Enter Gender:"**)  
city = input(**"Enter City\t:"**)  
salary = int(input(**"Enter Salary:"**))  
  
e=Employee()  
e.setId(id)  
e.setName(name)  
e.setGender(gender)  
e.setCity(city)  
e.setSalary(salary)  
id2 = e.getId()  
name2 = e.getName()  
gender2 = e.getGender()  
city2 = e.getCity()  
salary2 = e.getSalary()  
  
  
print(**"\nDisplaying Employee Data:"**)  
print(**"Id\t\t:"**, id2)  
print(**"Name\t:"**, name2)  
print(**"Gender\t:"**, gender2)  
print(**"City\t:"**, city2)  
print(**"Salary\t:"**, salary2)

………………………………………….

Single inheritance

*# Python code to demonstrate example of   
# single inheritance***class** Details:  
 **def** \_\_init\_\_(self):  
 self.\_\_id=**"<No Id>"** self.\_\_name=**"<No Name>"** self.\_\_gender=**"<No Gender>"  
 def** setData(self,id,name,gender):  
 self.\_\_id=id  
 self.\_\_name=name  
 self.\_\_gender=gender  
 **def** showData(self):  
 print(**"Id\t\t:"**,self.\_\_id)  
 print(**"Name\t\t:"**, self.\_\_name)  
 print(**"Gender\t\t:"**, self.\_\_gender)  
  
**class** Employee(Details): *#Inheritance* **def** \_\_init\_\_(self):  
 self.\_\_company=**"<No Company>"** self.\_\_dept=**"<No Dept>"  
 def** setEmployee(self,id,name,gender,comp,dept):  
 self.setData(id,name,gender)  
 self.\_\_company=comp  
 self.\_\_dept=dept  
 **def** showEmployee(self):  
 self.showData()  
 print(**"Company\t\t:"**, self.\_\_company)  
 print(**"Department\t:"**, self.\_\_dept)  
  
e=Employee()  
e.setEmployee(101,**"Prem Sharma"**,**"Male"**,**"New Delhi"**,110065)  
e.showEmployee()

…………………………….

*# Python code to demonstrate example of   
# single inheritance with two child classes***class** Details:  
 **def** \_\_init\_\_(self):  
 self.\_\_id=0  
 self.\_\_name=**""** self.\_\_gender=**""  
 def** setDetails(self):  
 self.\_\_id=int(input(**"Enter Id: "**))  
 self.\_\_name=input(**"Enter Name: "**)  
 self.\_\_gender=input(**"Enter gender: "**)  
 **def** showDetails(self):  
 print(**"Id: "**,self.\_\_id)  
 print(**"Name: "**,self.\_\_name)  
 print(**"Gender: "**,self.\_\_gender)  
  
**class** Employee(Details):  
 **def** \_\_init\_\_(self):  
 self.\_\_company=**""** self.\_\_desig=**""  
 def** setEmployee(self):  
 self.setDetails()  
 self.\_\_company=input(**"Enter Compmany Name: "**)  
 self.\_\_desig=input(**"Enter Designation: "**)  
 **def** showEmployee(self):  
 self.showDetails()  
 print(**"Company: "**,self.\_\_company)  
 print(**"Designation: "**,self.\_\_desig)  
  
**class** Doctor(Details):  
 **def** \_\_init\_\_(self):  
 self.\_\_hospital=**""** self.\_\_dept=**""  
 def** setDoctor(self):  
 self.setDetails()  
 self.\_\_hospital=input(**"Enter Hospital Name: "**)  
 self.\_\_dept=input(**"Enter Department: "**)  
 **def** showDoctor(self):  
 self.showDetails()  
 print(**"Hospital: "**,self.\_\_hospital)  
 print(**"Department"**,self.\_\_dept)  
  
**def** main():  
 print(**"Employee Object: "**)  
 e = Employee()  
 e.setEmployee()  
 e.showEmployee()  
 print(**"\nDoctor Object: "**)  
 d=Doctor()  
 d.setDoctor()  
 d.showDoctor()  
  
**if** \_\_name\_\_==**"\_\_main\_\_"**:  
 main()

…………………………………..

# Python code to demonstrate example of

# multiple inheritance

class Personel:

def \_\_init\_\_(self):

self.\_\_id=0

self.\_\_name=""

self.\_\_gender=""

def setPersonel(self):

self.\_\_id=int(input("Enter Id: "))

self.\_\_name = input("Enter Name: ")

self.\_\_gender = input("Enter Gender: ")

def showPersonel(self):

print("Id: ",self.\_\_id)

print("Name: ",self.\_\_name)

print("Gender: ",self.\_\_gender)

class Educational:

def \_\_init\_\_(self):

self.\_\_stream=""

self.\_\_year=""

def setEducational(self):

self.\_\_stream=input("Enter Stream: ")

self.\_\_year = input("Enter Year: ")

def showEducational(self):

print("Stream: ",self.\_\_stream)

print("Year: ",self.\_\_year)

class Student(Personel,Educational):

def \_\_init\_\_(self):

self.\_\_address = ""

self.\_\_contact = ""

def setStudent(self):

self.setPersonel()

self.\_\_address = input("Enter Address: ")

self.\_\_contact = input("Enter Contact: ")

self.setEducational()

def showStudent(self):

self.showPersonel()

print("Address: ",self.\_\_address)

print("Contact: ",self.\_\_contact)

self.showEducational()

def main():

s=Student()

s.setStudent()

s.showStudent()

if \_\_name\_\_=="\_\_main\_\_":main()

………………………………………………………………………………..

# Python code to demonstrate example of

# multilevel inheritance

class Details1:

def \_\_init\_\_(self):

self.\_\_id=0

def setId(self):

self.\_\_id=int(input("Enter Id: "))

def showId(self):

print("Id: ",self.\_\_id)

class Details2(Details1):

def \_\_init\_\_(self):

self.\_\_name=""

def setName(self):

self.setId()

self.\_\_name=input("Enter Name: ")

def showName(self):

self.showId()

print("Name: ",self.\_\_name)

class Details3(Details2):

def \_\_init\_\_(self):

self.\_\_gender=""

def setGender(self):

self.setName()

self.\_\_gender=input("Enter Gender: ")

def showGender(self):

self.showName()

print("Gender: ",self.\_\_gender)

class Employee(Details3):

def \_\_init\_\_(self):

self.\_\_desig=""

self.\_\_dept=""

def setEmployee(self):

self.setGender()

self.\_\_desig=input("Enter Designation: ")

self.\_\_dept= input("Enter Department: ")

def showEmployee(self):

self.showGender()

print("Designation: ",self.\_\_desig)

print("Department: ",self.\_\_dept)

def main():

e = Employee()

e.setEmployee()

e.showEmployee()

if \_\_name\_\_=="\_\_main\_\_":main()

……………………………………

# Python code to demonstrate example of

# hierarchical inheritance

class Details:

def \_\_init\_\_(self):

self.\_\_id="<No Id>"

self.\_\_name="<No Name>"

self.\_\_gender="<No Gender>"

def setData(self,id,name,gender):

self.\_\_id=id

self.\_\_name=name

self.\_\_gender=gender

def showData(self):

print("Id: ",self.\_\_id)

print("Name: ", self.\_\_name)

print("Gender: ", self.\_\_gender)

class Employee(Details): #Inheritance

def \_\_init\_\_(self):

self.\_\_company="<No Company>"

self.\_\_dept="<No Dept>"

def setEmployee(self,id,name,gender,comp,dept):

self.setData(id,name,gender)

self.\_\_company=comp

self.\_\_dept=dept

def showEmployee(self):

self.showData()

print("Company: ", self.\_\_company)

print("Department: ", self.\_\_dept)

class Doctor(Details): #Inheritance

def \_\_init\_\_(self):

self.\_\_hospital="<No Hospital>"

self.\_\_dept="<No Dept>"

def setEmployee(self,id,name,gender,hos,dept):

self.setData(id,name,gender)

self.\_\_hospital=hos

self.\_\_dept=dept

def showEmployee(self):

self.showData()

print("Hospital: ", self.\_\_hospital)

print("Department: ", self.\_\_dept)

def main():

print("Employee Object")

e=Employee()

e.setEmployee(1,"Prem Sharma","Male","gmr","excavation")

e.showEmployee()

print("\nDoctor Object")

d = Doctor()

d.setEmployee(1, "pankaj", "male", "aiims", "eyes")

d.showEmployee()

if \_\_name\_\_=="\_\_main\_\_":

main()

…………………………………………..

# Create a Student class

class Student :

# initialise class variable

counter = 0

# Constructor method

def \_\_init\_\_(self,name,age) :

# instance variable or object attributes

self.name = name

self.age = age

# incrementing the class variable by 1

# whenever new object is created

Student.counter += 1

# Create a method for printing details

def printDetails(self) :

print(self.name,self.age,"years old")

# Create an object of Student class with attributes

student1 = Student('Ankit Rai',22)

student2 = Student('Aishwarya',21)

student3 = Student('Shaurya',21)

# Print the total no. of objects cretaed

print("Total number of objects created: ",Student.counter)

………………………………….

# Define a class for Checking prime number

class Check :

# Constructor

def \_\_init\_\_(self,number) :

self.num = number

# define a method for checking number is prime or not

def isPrime(self) :

for i in range(2, int(num \*\* (1/2)) + 1) :

# if any number is divisible by i

# then number is not prime

# so return False

if num % i == 0 :

return False

# if number is prime then return True

return True

# Main code

if \_\_name\_\_ == "\_\_main\_\_" :

# input number

num = 11

# make an object of Check class

check\_prime = Check(num)

# method calling

print(check\_prime.isPrime())

num = 14

check\_prime = Check(num)

print(check\_prime.isPrime())

…………………………………………

## Using the \_\_del\_\_ method

Below we have a simple code for a class Example, where we have used the \_\_init\_\_method to initialize our object, while we have defined the \_\_del\_\_ method to act as a destructor.

class Example:

def \_\_init\_\_(self):

print "Object created"

# destructor

def \_\_del\_\_(self):

print "Object destroyed"

# creating an object

myObj = Example()

# to delete the object explicitly

del myObj

Object created

Object destroyed