**Ch#1 Python - OOP Concepts**

Python has been an object-oriented language since the time it existed. Due to this, creating and using classes and objects are downright easy. This chapter helps you become an expert in using Python's object-oriented programming support.

If you do not have any previous experience with object-oriented (OO) programming, you may want to consult an introductory course on it or at least a tutorial of some sort so that you have a grasp of the basic concepts. However, here is a small introduction of Object-Oriented Programming (OOP) to help you.

Procedural Oriented Approach

Early programming languages developed in 50s and 60s are recognized as procedural (or procedure oriented) languages.

A computer program describes procedure of performing certain task by writing a series of instructions in a logical order. Logic of a more complex program is broken down into smaller but independent and reusable blocks of statements called functions.

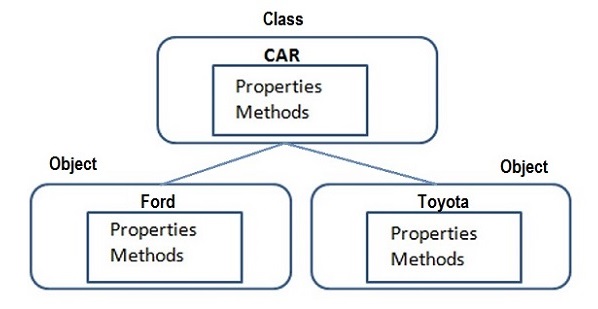
Every function is written in such a way that it can interface with other functions in the program. Data belonging to a function can be easily shared with other in the form of arguments, and called function can return its result back to calling function.

Prominent problems related to procedural approach are as follows −

* Its top-down approach makes the program difficult to maintain.
* It uses a lot of global data items, which is undesired. Too many global data items would increase memory overhead.
* It gives more importance to process and doesn't consider data of same importance and takes it for granted, thereby it moves freely through the program.
* Movement of data across functions is unrestricted. In real-life scenario where there is unambiguous association of a function with data it is expected to process.

Python - OOP Concepts

In the real world, we deal with and process objects, such as student, employee, invoice, car, etc. Objects are not only data and not only functions, but combination of both. Each real-world object has attributes and behavior associated with it.



Attributes

* Name, class, subjects, marks, etc., of student
* Name, designation, department, salary, etc., of employee
* Invoice number, customer, product code and name, price and quantity, etc., in an invoice
* Registration number, owner, company, brand, horsepower, speed, etc., of car

Each attribute will have a value associated with it. Attribute is equivalent to data.

Behavior

Processing attributes associated with an object.

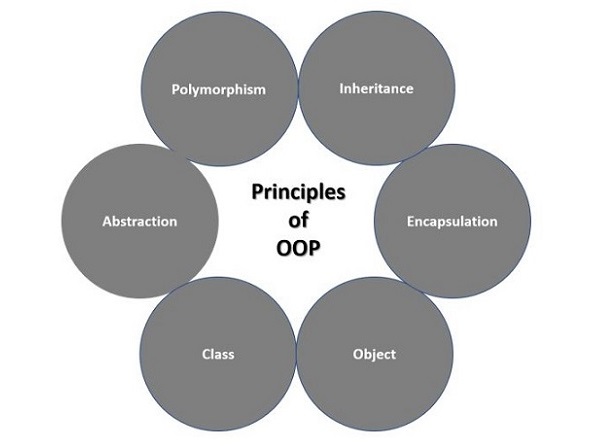
* Compute percentage of student's marks
* Calculate incentives payable to employee
* Apply GST to invoice value
* Measure speed of car

Behavior is equivalent to function. In real life, attributes and behavior are not independent of each other, rather they co-exist.

The most important feature of object-oriented approach is defining attributes and their functionality as a single unit called class. It serves as a blueprint for all objects having similar attributes and behavior.

In OOP, class defines what are the attributes its object has, and how is its behavior. Object, on the other hand, is an instance of the class.

Object-oriented programming paradigm is characterized by the following principles −



Class

A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.

Object

An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle. A unique instance of a data structure that is defined by its class. An object comprises both data members (class variables and instance variables) and methods.

Encapsulation

Data members of class are available for processing to functions defined within the class only. Functions of class on the other hand are accessible from outside class context. So object data is hidden from environment that is external to class. Class function (also called method) encapsulates object data so that unwarranted access to it is prevented.

Inheritance

A software modelling approach of OOP enables extending capability of an existing class to build new class instead of building from scratch. In OOP terminology, existing class is called base or parent class, while new class is called child or sub class.

Child class inherits data definitions and methods from parent class. This facilitates reuse of features already available. Child class can add few more definitions or redefine a base class function.

Polymorphism

Polymorphism is a Greek word meaning having multiple forms. In OOP, polymorphism occurs when each sub class provides its own implementation of an abstract method in base class.

Differences between Procedural and Object Oriented Programming

The following table highlights the major differences between Procedural Programming and Object Oriented Programming −

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Object Oriented Programming** | **Procedural Programming** |
| Definition | Object-oriented Programming is a programming language that uses classes and objects to create models based on the real world environment.  In OOPs, it makes it easy to maintain and modify existing code as new objects are created inheriting characteristics from existing ones. | Procedural Programming is a programming language that follows a step-by-step approach to break down a task into a collection of variables and routines (or subroutines) through a sequence of instructions.  Each step is carried out in order in a systematic manner so that a computer can understand what to do. |
| Approach | In OOPs concept of objects and classes is introduced and hence the program is divided into small chunks called objects which are instances of classes. | In procedural programming, the main program is divided into small parts based on the functions and is treated as separate program for individual smaller program. |
| Access modifiers | In OOPs access modifiers are introduced namely as Private, Public, and Protected. | No such modifiers are introduced in procedural programming. |
| Security | Due to abstraction in OOPs data hiding is possible and hence it is more secure than POP. | Procedural programming is less secure as compare to OOPs. |
| Complexity | OOPs due to modularity in its programs is less complex and hence new data objects can be created easily from existing objects making object-oriented programs easy to modify | There is no simple process to add data in procedural programming, at least not without revising the whole program. |
| Program division | OOP divides a program into small parts and these parts are referred to as objects. | Procedural programming divides a program into small programs and each small program is referred to as a function. |
| Importance | OOP gives importance to data rather than functions or procedures. | Procedural programming does not give importance to data. In POP, functions along with sequence of actions are followed. |
| Inheritance | OOP provides inheritance in three modes i.e. protected, private, and public | Procedural programming does not provide any inheritance. |
| Examples | C++, C#, Java, Python, etc. are the examples of OOP languages. | C, BASIC, COBOL, Pascal, etc. are the examples POP languages. |

**Extra point:**

## The \_\_init\_\_() Function

All classes have a function called \_\_init\_\_(), which is always executed when the class is being initiated.

Use the \_\_init\_\_() function to assign values to object properties, or other operations that are necessary to do when the object is being created:

**Note:** The \_\_init\_\_() function is called automatically every time the class is being used to create a new object.

## The \_\_str\_\_() Function

The \_\_str\_\_() function controls what should be returned when the class object is represented as a string.

If the \_\_str\_\_() function is not set, the string representation of the object is returned:

Example

The string representation of an object WITH the \_\_str\_\_() function:

class Person:  
  def \_\_init\_\_(self, name, age):  
    self.name = name  
    self.age = age  
  
  def \_\_str\_\_(self):  
    return f"{self.name}({self.age})"  
  
p1 = Person("John", 36)  
  
print(p1)

## Object Methods

Objects can also contain methods. Methods in objects are functions that belong to the object.

Instance Methods and Variables:

Instance Variables:

Instance variables are unique to each instance of the class. They are defined inside the constructor (\_\_init\_\_) and are prefixed with self.

class MyClass:

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

self.instance\_variable = value

obj1 = MyClass(10)

obj2 = MyClass(20)

print(obj1.instance\_variable) # Output: 10

print(obj2.instance\_variable) # Output: 20

Instance Methods:

Instance methods are defined with self as the first parameter. They can access and modify instance variables.

class MyClass:

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

self.instance\_variable = value

def update\_variable(self, new\_value):

self.instance\_variable = new\_value

obj = MyClass(10)

print(obj.instance\_variable) # Output: 10

obj.update\_variable(30)

print(obj.instance\_variable) # Output: 30

**Note:** The self parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.

Delete Object Properties

You can delete properties on objects by using the del keyword:

Example

Delete the age property from the p1 object:

del p1.age

[Try it Yourself »](https://www.w3schools.com/python/trypython.asp?filename=demo_class7)

Delete Objects

You can delete objects by using the del keyword:

Example

Delete the p1 object:

del p1

**1.1 Python - Object and Classes**

Python is a highly object-oriented language. In Python, each and every element in a Python program is an object of one or the other class. A number, string, list, dictionary etc. used in a program they are objects of corresponding built-in classes.

#Example 1

num=20

print (type(num))

num1=55.50

print (type(num1))

s="TutorialsPoint"

print (type(s))

dct={'a':1,'b':2,'c':3}

print (type(dct))

def SayHello():

   print ("Hello World")

   return

print (type(SayHello))

"""In Python, the Object class is the base or parent class for all the classes,

built-in as well as user defined.

The class keyword is used to define a new class"""

class Classname:

    "Optional class documentation string"

    class\_suite

"""

The class has a documentation string, which can be accessed via ClassName.\_\_doc\_\_.

The class\_suite consists of all the component statements defining class members, data attributes and functions.

        """

#Example 2

class Emp(obj):

    "Common base class for all employee"

    pass

"""

Any class in Python is a subclass of object class,

hence object is written in parentheses.

However, later versions of Python don't require object to be put in parentheses.

        """

class Emp:

        "Common base class for all employee"

        pass

#To define an object of this class, use the following syntax −

obj=Emp()#create Emp class instance or object

**1.2 Python - Class Attributes**

Every Python class keeps the following built-in attributes and they can be accessed using dot operator like any other attribute −

* **\_\_dict\_\_** − Dictionary containing the class's namespace.
* **\_\_doc\_\_** − Class documentation string or none, if undefined.
* **\_\_name\_\_** − Class name.
* **\_\_module\_\_** − Module name in which the class is defined. This attribute is "\_\_main\_\_" in interactive mode.
* **\_\_bases\_\_** − A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

#Example 1

#Creating class

class Employee:

    #This is emp doc

    "Abasyn employees"

    #Create class constructor(Overload)

    def \_\_init\_\_(self,name="Shary",age=24):

        #Create instance (object) variable

         self.name1=name

         self.age1=age

    #create function

    def displayEmp(self):

        print("Name: ",self.name1,"\tAge: ",self.age1)

#print the output

print("Employee.\_\_doc\_\_:",Employee.\_\_doc\_\_)

print("Employee.\_\_name\_\_(Class\_name):",Employee.\_\_name\_\_)

print("Employee.\_\_module\_\_:",Employee.\_\_module\_\_)

print("Employee.\_\_bases\_\_:",Employee.\_\_bases\_\_)

print("Employee.\_\_dict\_\_:",Employee.\_\_dict\_\_)

"""

name and age are instance variables, as their values may be different for each object.

A class attribute or variable whose value is shared among all the instances of a in this class.

A class attribute represents common attribute of all objects of a class.

Class attributes are not initialized inside \_\_init\_\_() constructor.

They are defined in the class, but outside any method.

"""

#Example 2

"""

Let us add a class variable called empCount in Employee class.

For each object declared, the \_\_init\_\_() method is automatically called.

This method initializes the instance variables as well as increments the empCount by 1."""

class Emp:

    #create class variable

    empCount=0

    #create constructor

    def \_\_init\_\_(self,name,age):

        self.name1=name

        self.age1=age

        Emp.empCount+=1

        print("Name :",self.name1,"\tAge: ",self.age1)

        print("Employee number: ",Emp.empCount)

#Create instances of class

e1=Emp("Shary",24)

e2=Emp("Umair",23)

e3=Emp("Fahad",22)

**1.3 Python - Class Methods**

An instance method accesses the instance variables of the calling object because it takes the reference to the calling object. But it can also access the class variable as it is common to all the objects.

Python has a built-in function classmethod() which transforms an instance method to a class method which can be called with the reference to the class only and not the object.

Syntax

classmethod(instance\_method)

Example

In the Employee class, define a showcount() instance method with the "**self**" argument (reference to calling object). It prints the value of empCount. Next, transform the method to class method counter() that can be accessed through the class reference.

#Example 1

class Employee:

    #  Create class variable initialized to 0

    empCount=0

    #The \_\_init\_\_ method is the constructor that gets called when an object of the class is created.

    def \_\_init\_\_(self,name,age):

        #Instance variables value passed during obj creation

        self.name1=name

        self.age1=age

        #It increments the class variable empCount by 1, indicating the creation of a new employee.

        Employee.empCount+=1

    #Class method

    def showCount(self):

        #This is an instance method named showcount that prints the current value of the class variable empCount

        print("Total Employees: ",self.empCount)

        """

        This line creates a class method named counter using the classmethod decorator.

        The counter class method is associated with the showcount instance method.

        """

    #Class method  using classmethod decorator

    Counter=classmethod(showCount)

#Creates instanses or object

obj1=Employee("Shary",25)

obj2=Employee("Fahad",23)

#Calling instance method

obj1.showCount()#This calls the showcount instance method for the e1 object, which prints the current value of empCount.

#Calling class method

Employee.Counter()#It also prints the current value of empCount because the class method has access to the class variables.

"""

Using @classmethod() decorator is the prescribed way to define

a class method as it is more convenient than first declaring

an instance method and then transforming to a class method.

"""

@classmethod

def showCount1(cls):

        print(cls.empCount)

        return

@classmethod

def newemployee(cls,name,age):

        return cls(name,age)

obj3=Employee("Shary",34)

obj3=Employee("Umair",24)

obj4=Employee.newemployee("Khan",10)

Employee.showCount1()

#important points

"""

Class Methods and Variables:

Class Variables:

Class variables are shared among all instances of a class. They are defined outside any method in the class and are accessed using the class name.

python

Copy code

class MyClass:

    class\_variable = 0

obj1 = MyClass()

obj2 = MyClass()

MyClass.class\_variable = 10

print(obj1.class\_variable)  # Output: 10

print(obj2.class\_variable)  # Output: 10

Class Methods:

Class methods are defined using the @classmethod decorator. They take the class as the first parameter (cls) instead of the instance (self). Class methods can access and modify class variables.

class MyClass:

    class\_variable = 0

    @classmethod

    def update\_class\_variable(cls, new\_value):

        cls.class\_variable = new\_value

obj1 = MyClass()

obj2 = MyClass()

print(obj1.class\_variable)  # Output: 0

MyClass.update\_class\_variable(20)

print(obj1.class\_variable)  # Output: 20

print(obj2.class\_variable)  # Output: 20

    """

**1.4 Python - Static Methods**

is that the static method doesn't have a mandatory argument like reference to the object **− self** or reference to the class **− cls**. Python's standard library fimction staticmethod() returns a static method.

In the Employee class below, a method is converted into a static method. This static method can now be called by its object or reference of class itself.

#Example 1

class Employee:

   empCount = 0

   def \_\_init\_\_(self, name, age):

      self.\_\_name = name

      self.\_\_age = age

      Employee.empCount += 1

    #Create a static method using @static method decorator

   @staticmethod #tecnique 1

   def showCount():

       print("Total Eployees:",Employee.empCount)##It prints the current value of the class variable empCount.

       return

   #Also create from this tecnique 2

   counter=staticmethod(showCount)

#Create object

obj1=Employee("Shary",45)

obj2=Employee("Hamad",14)

#Calling Static Method using an Instance

obj1.counter()

#Calling Static Method using the Class:

Employee.counter()

#Also work

Employee.showCount()

**Class method vs Static Method**

The difference between the Class method and the static method is:

* A class method takes cls as the first parameter while a static method needs no specific parameters.
* A class method can access or modify the class state while a static method can’t access or modify it.
* In general, static methods know nothing about the class state. They are utility-type methods that take some parameters and work upon those parameters. On the other hand class methods must have class as a parameter.
* We use @classmethod decorator in python to create a class method and we use @staticmethod decorator to create a static method in python.

**Decorator:**

In Python, a decorator is a special type of function that is used to modify the behavior of another function or a class method. Decorators allow you to extend or modify the behavior of functions or methods without changing their actual code. Decorators are applied using the **@decorator** syntax before the function or method definition.

**1.5 Python - Constructors**

In object-oriented programming, an object of a class is characterized by one or more instance variables or attributes, whose values are unique to each object. For example, if the Employee class has an instance attribute as name. Each of its objects e1 and e2 may have different value for the name variable.

A constructor is an instance method in a class, that is automatically called whenever a new object of the class is declared. The constructor' role is to assign value to instance variables as soon as the object is declared.

Python uses a special method called \_\_init\_\_() to initialize the instance variables for the object, as soon as it is declared.

The \_\_init\_\_() method acts as a constructor. It needs a mandatory argument self, which the reference to the object.

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

#initialize instance variables

The \_\_init\_\_() method as well as any instance method in a class has a mandatory parameter, **self**. However, you can give any name to the first parameter, not necessarily self.

Let us define the constructor in Employee class to initialize name and age as instance variables. We can then access these attributes of its object.

#Example 1

class Employee:

    "Common base class for all employees"

    #Create Constructors(defult)

    def \_\_init\_\_(self):

        self.name1="Shary"

        self.age1=25

#Create instanse

obj1=Employee()

print(f"Name: {obj1.name1}")

print(f"Age: {obj1.age1}")

#Create 2nd object

obj2=Employee()

print(f"Name: {obj2.name1}")

print(f"Age: {obj2.age1}")

#Note bydefult constructor each object we declare will have same value for its instance variables name and age.

#Example 2

class Employee:

    "Common base class for all employees"

    #Create Parameterized Constructor

    def \_\_init\_\_(self,name,age):

        self.name1=name

        self.age1=age

#Create instances and pass arguments

obj1=Employee("Ali",20)

print(f"Name: {obj1.name1}")

print(f"Age: {obj1.age1}")

#Create 2nd object

obj2=Employee("Fahad",30)

print(f"Name: {obj2.name1}")

print(f"Age: {obj2.age1}")

"""

You can assign defaults to the formal arguments in the constructor

so that the object can be instantiated with or without passing parameters.

        """

#Example 3

class Employee:

    "Common base class for all employees"

    #Create Parameterized Constructor

    def \_\_init\_\_(self,name="Umair",age=40):

        self.name1=name

        self.age1=age

#Create instances and pass arguments

obj1=Employee("Ali",20)

print(f"Name: {obj1.name1}")

print(f"Age: {obj1.age1}")

#Create 2nd object without parameter so its using the above defult values

obj2=Employee()

print(f"Name: {obj2.name1}")

print(f"Age: {obj2.age1}")

#1.5.2  Python - Instance Methods

"""

In addition to the \_\_init\_\_() constructor, there may be one or more instance methods defined in a class.

A method with self as one of the formal arguments is called instance method, as it is called by a specific object."""

#Example 5

"""In the following example a displayEmployee() method has been defined. It returns the name and age attributes of the Employee object that calls the method."""

class Employee:

    "Common base class for all employees"

    #Create Parameterized Constructor

    def \_\_init\_\_(self,name="Umair",age=40):

        self.name1=name

        self.age1=age

    def displayEmp(self):

        print(f"Name: {self.name1} Age:{self.age1}")

#Create objects

obj3=Employee()

obj4=Employee("Ayan",10)

#calling displayEmp using obj

obj3.displayEmp()

obj4.displayEmp()

#You can add, remove, or modify attributes of classes and objects at any time

obj3.salary=700 #Add a salary attribute

print(f"Name: {obj3.name1} Age:{obj3.age1} Salary: {obj3.salary}")

obj3.name1="Umair khan"#Modify name attribute

print(f"Name: {obj3.name1} Age:{obj3.age1} Salary: {obj3.salary}")

del obj3.salary #delete salary attribute

#print(f"Name: {obj3.name1} Age:{obj3.age1} Salary: {obj3.salary}")

#AttributeError: 'Employee' object has no attribute 'salary'

"""

Instead of using the normal statements to access attributes, you can use the following functions −

The getattr(obj, name[, default]) − to access the attribute of object.

The hasattr(obj,name) − to check if an attribute exists or not.

The setattr(obj,name,value) − to set an attribute. If attribute does not exist, then it would be created.

The delattr(obj, name) − to delete an attribute.

        """

print(hasattr(obj3,'salary'))# Returns true if 'salary' attribute exists

setattr(obj3,'salary',8000)# Set attribute 'salary' at 8

print(getattr(obj3,'salary'))# Returns value of name attribute

delattr(obj3,'salary')# Returns value of name attribute

**1.6 Python - Access Modifiers**

The languages such as C++ and Java, use access modifiers to restrict access to class members (i.e., variables and methods). These languages have keywords public, protected, and private to specify the type of access.

A class member is said to be public if it can be accessed from anywhere in the program. Private members are allowed to be accessed from within the class only.

* Usually, methods are defined as public and instance variable are private. This arrangement of private instance variables and public methods ensures implementation of principle of encapsulation.
* **Protected members** are accessible from within the class as well as by classes derived from that class.

Unlike these languages, Python has no provision to specify the type of access that a class member may have. By default, all the variables and methods in a class are public.

Example

Here, we have Employee class with instance variables name and age. An object of this class has these two attributes. They can be directly accessed from outside the class, because they are public.

#Example 1

class Employee:

    "Common base class for all employee"

    #Create overload constructor

    def \_\_init\_\_(self,name="Shary",age=25):

        #Create instance variable

        self.name=name

        self.age=age

#Create instance

obj1=Employee()#no arguments pass so using default parameters

obj2=Employee("Ali",25)#pass args so using this

#Output

#1st obj

print(f"Name: {obj1.name}")#shary

print(f"age: {obj1.age}")#24

#2nd obj

print(f"Name: {obj2.name}")#ali

print(f"age: {obj2.age}")#25

"""

Python doesn't enforce restrictions on accessing any instance variable or method.

However, Python prescribes a convention of prefixing name of variable/method with

single or double underscore to emulate behavior of protected and private access modifiers.

To indicate that an instance variable is private, prefix it with double underscore (such as "\_\_age").

To imply that a certain instance variable is protected, prefix it with single underscore (such as "\_salary")

        """

"""

Let us modify the Employee class. Add another instance variable salary.

Make age private and salary as protected by prefixing double and single underscores respectively.

"""

#Example 2 using Private and protected

class Employee:

    #Create overload constructor

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

        self.name=name #public varaible(anywhere acess with any pakage)

        self.\_\_age=age #private variable private access within class and not access without pakage

        self.\_salary=salary# variable protected no access out of class

    def displayEmployee(self):

        print(f"Name: {self.name} age: {self.\_\_age} salary:{self.\_salary}")

obj3=Employee("Khan",25,1200)

print("\nPublic name",obj3.name)

print("protected salary",obj3.\_salary)#protected

#print(obj3.\_\_age)#private not access out of class

**6.1 Name Mangling**

## Name Mangling

Python doesn't block access to private data, it just leaves for the wisdom of the programmer, not to write any code that access it from outside the class. You can still access the private members by Python's name mangling technique.

Name mangling is the process of changing name of a member with double underscore to the form **object.\_class\_\_variable**. If so required, it can still be accessed from outside the class, but the practice should be refrained.

In our example, the private instance variable "\_\_name" is mangled by changing it to the format

obj.\_class\_\_privatevar

So, to access the value of "\_\_age" instance variable of "e1" object, change it to "e1.\_Employee\_\_age".

"""

#We can access private value

print("(Private) age:",obj3.\_Employee\_\_age)

## 6.2 Python Property Object

Python's standard library has a built-in property() function. It returns a property object. It acts as an interface to the instance variables of a Python class.

The encapsulation principle of object-oriented programming requires that the instance variables should have a restricted private access. Python doesn't have efficient mechanism for the purpose. The property() function provides an alternative.

The property() function uses the getter, setter and delete methods defined in a class to define a property object for the class.

### Syntax

property(fget=None, fset=None, fdel=None, doc=None)

### Parameters

* **fget** − an instance method that retrieves value of an instance variable.
* **fset** − an instance method that assigns value to an instance variable.
* **fdel** − an instance method that removes an instance variable
* **fdoc** − Documentation string for the property.

The function uses getter and setter methods to return the property objec.

#Example The complete program with property objects and their use is given below −

class Employee:

     def \_\_init\_\_(self,name,age):

         self.\_\_name=name

         self.\_\_age=age

     def get\_n(self):

         return self.\_\_name

     def get\_a(self):

         return self.\_\_age

     def set\_n(self,name):

         self.\_\_name=name

         return

     def set\_a(self,age):

         self.\_\_age=age

         return

     name=property(get\_n,set\_n,"name")

     age=property(get\_a,set\_a,"Age")

obj6=Employee("Sufyaan",40)

print("\nName:",obj6.name,"Age",obj6.age)

obj6.name='Changed name'

obj6.age=80

print("Name:",obj6.name,"Age:",obj6.age)

## Getters and Setter Methods

A getter method retrieves the value of an instance variable, usually named as get\_varname, whereas the setter method assigns value to an instance variable − named as set\_varname.

Let us define getter methods get\_name() and get\_age(), and setters set\_name() and set\_age() in the Employee class.

#Example of getter and setter methods

class Employee:

    def \_\_init\_\_(self,name,age):

        self.\_\_name=name

        self.\_\_age=age

    def get\_name(self):

        return self.\_\_name

    def get\_age(self):

        return self.\_\_age

    def set\_name(self,name):

        self.\_\_name=name

        return

    def set\_age(self,age):

        self.\_\_age=age

obj5=Employee("Shary",24)

print("get\_Name:",obj5.get\_name(),"get\_age: ",obj5.get\_age())

obj5.set\_name("Ali")

obj5.set\_age(25)

print("Get\_Name:",obj5.get\_name(),"Get\_Age:",obj5.get\_age())

**7. Python - Inheritance**

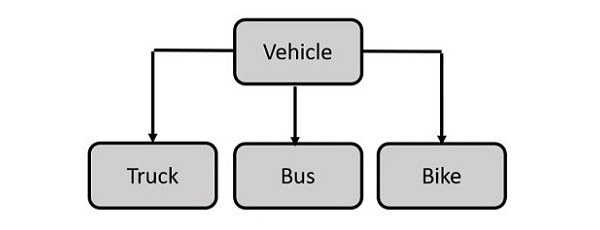
Inheritance is one of the most important features of Object-oriented programming methodology. It is most often used in software development process using many languages such as Java, PHP, Python, etc.

Instead of starting from scratch, you can create a class by deriving it from a pre-existing class by listing the parent class in parentheses after the new class name.

Instead of starting from scratch, you can create a class by deriving it from a pre-existing class by listing the parent class in parentheses after the new class name.

If you have to design a new class whose most of the attributes are already well defined in an existing class, then why redefine them? Inheritance allows capabilities of existing class to be reused and if required extended to design new class.

Inheritance comes into picture when a new class possesses 'IS A' relationship with an existing class. Car IS a vehicle. Bus IS a vehicle; Bike IS also a vehicle. Vehicle here is the parent class, whereas car, bus and bike are the child classes.



Syntax

Derived classes are declared much like their parent class; however, a list of base classes to inherit from is given after the class name −

class SubClassName (ParentClass1[, ParentClass2, ...]):

'Optional class documentation string'

class\_suite

**Examples:**

class Parent:

    def \_\_init\_\_(self):#parent constructor

        self.attr=100

        print("Calling parent constructor")

    def parentMethod(self):

        print("Parent method")

    def set\_attr(self,attr):

        self.attr=attr

    def get\_attr(self):

        print("Parent attr:",self.attr)

class child(Parent):#define child class

    def \_\_init\_\_(self):

        print("Calling child constructor")

    def childMethod(self):

        print("Calling child method")

obj1=child()# instace of child c

obj1.parentMethod()#calling parent method

obj1.childMethod()# calling child method

obj1.set\_attr(200)#caliing parent method

class division:

   def \_\_init\_\_(self, a,b):

      self.n=a

      self.d=b

   def divide(self):

      return self.n/self.d

class modulus:

   def \_\_init\_\_(self, a,b):

      self.n=a

      self.d=b

   def mod\_divide(self):

      return self.n%self.d

class div\_mod(division,modulus):

   def \_\_init\_\_(self, a,b):

      self.n=a

      self.d=b

   def div\_and\_mod(self):

      divval=division.divide(self)

      modval=modulus.mod\_divide(self)

      return (divval, modval)

x=div\_mod(10,3)

print ("division:",x.divide())

print ("mod\_division:",x.mod\_divide())

print ("divmod:",x.div\_and\_mod())

# Example 2

# Define a class 'division' with an initializer (\_\_init\_\_) and a method 'div'

class division:

    def \_\_init\_\_(self, a, b):

        self.n = a

        self.d = b

    def div(self):

        return self.n / self.d

# Define a class 'modules' with an initializer (\_\_init\_\_) and a method 'mdiv'

class modules:

    def \_\_init\_\_(self, a, b):

        self.n = a

        self.d = b

    def mdiv(self):

        return self.n % self.d

# Define a class 'div\_mod' that inherits from both 'division' and 'modules'

class div\_mod(division, modules):

    def \_\_init\_\_(self, a, b):      # When you add the \_\_init\_\_() function, the child class will no longer inherit the parent's \_\_init\_\_() function.

        """

        Note: The child's \_\_init\_\_() function overrides the inheritance of the parent's \_\_init\_\_() function.

        To keep the inheritance of the parent's \_\_init\_\_() function, add a call to the parent's \_\_init\_\_() function: """

        division.\_\_init\_\_(self,a,b)

        """Use the super() Function

        Python also has a super() function that will make the child class

        inherit all the methods and properties from its parent:

        """

        # Initialize the attributes of the base classes

        super().\_\_init\_\_(a, b)

        """

        By using the super() function, you do not have to use the name of the parent element,

        it will automatically inherit the methods and properties from its parent.

        """

    def div\_mod\_f(self):

        # Call the 'div' method from the 'division' class and 'mdiv' method from the 'modules' class

        div\_value = self.div()

        mod\_value = self.mdiv()

        return div\_value, mod\_value

# Create an instance of the 'div\_mod' class with values 10 and 3

obj2 = div\_mod(10, 3)

# Print the results of calling methods on the 'div\_mod' instance

print("Division:", obj2.div())            # Output: 3.3333333333333335

print("Mod Division:", obj2.mdiv())       # Output: 1

print("Div and Mod:", obj2.div\_mod\_f())   # Output: (3.3333333333333335, 1)

**8. Python - Polymorphism**

The term "polymorphism" refers to a function or method taking different form in different contexts. Since Python is a dynamically typed language, Polymorphism in Python is very easily implemented.

If a method in a parent class is overridden with different business logic in its different child classes, the base class method is a polymorphic method.

OR

The word "polymorphism" means "many forms", and in programming it refers to methods/functions/operators with the same name that can be executed on many objects or classes.

## Function Polymorphism

An example of a Python function that can be used on different objects is the len() function.

### String

For strings len() returns the number of characters:

Examples:

Print(len(string))

Print(len(dict))

Print(len(tuple))

## Class Polymorphism

Polymorphism is often used in Class methods, where we can have multiple classes with the same method name.

For example, say we have three classes: Car, Boat, and Plane, and they all have a method called move():

#Example 1  Different classes with the same method:

class Car:

    def \_\_init\_\_(self,brand,model):

        self.brand=brand

        self.model=model

    def move(self):

        print("Drive!")

class Boat:

    def \_\_init\_\_(self,brand,model):

        self.brand=brand

        self.model=model

    def move(self):

        print("Sail!")

class Plane:

    def \_\_init\_\_(self,brand,model):

        self.brand=brand

        self.model=model

    def move(self):

        print("Fly!")

#create Car instance

car1=Car("Ford","Mustang")

#create Boat instance

boat1=Boat("Ibiza","Touring 20")

#Create Plane instance

plane1=Plane("Boeing","747")

for i in (car1,boat1,plane1):

    i.move()

#Output

# Drive!

#Sail!

#Fly!

#Note : Look at the for loop at the end. Because of polymorphism we can execute the same method for all three classes.

# 8.1 Inheritance Class Polymorphism

"""

What about classes with child classes with the same name? Can we use polymorphism there?

Yes. If we use the example above and make a parent class called Vehicle,

and make Car, Boat, Plane child classes of Vehicle, the child classes inherits the Vehicle methods, but can override them:

    """

#Example 2

class Vehicle:

    def \_\_init\_\_(self,brand,model):

        self.brand=brand

        self.model=model

    def move(self):

        print("Move!")

class Car(Vehicle):

    pass

class Boat(Vehicle):

    def move(self):

        print("Sail")

class Plane(Vehicle):

    def move(self):

        print("Fly!")

car2=Car("NE","Mustang")

boat2=Boat("Ibiza2","Touring 30")

plane2=Plane("Boeing2","700")

for j in (car2,boat2,plane2):

    print("\n",i.brand)

    print(j.model)

    j.move()

**Note:**

Child classes inherits the properties and methods from the parent class.

In the example above you can see that the Car class is empty, but it inherits brand, model, and move() from Vehicle.

The Boat and Plane classes also inherit brand, model, and move() from Vehicle, but they both override the move() method.

Because of polymorphism we can execute the same method for all classes.

#Example 3

"""

As an example of polymorphism given below, we have shape which is an abstract class.

It is used as parent by two classes circle and rectangle. Both classes overrideparent's draw() method in different ways."""

# Import the ABC (Abstract Base Class) and abstractmethod from the abc module

from abc import ABC, abstractmethod

# Define an abstract class 'shape' that inherits from ABC (Abstract Base Class)

class shape(ABC):

    # Define an abstract method 'draw'

    @abstractmethod

    def draw(self):

        "Abstract method"

        return

# Define a concrete class 'circle' that inherits from the 'shape' class

class circle(shape):

    # Implement the 'draw' method for the 'circle' class

    def draw(self):

        super().draw()  # Call the 'draw' method of the base class (shape) if it exists

        print("Draw a circle")  # Print a message indicating drawing a circle

        return

# Define a concrete class 'rectangle' that inherits from the 'shape' class

class rectangle(shape):

    # Implement the 'draw' method for the 'rectangle' class

    def draw(self):

        super().draw()  # Call the 'draw' method of the base class (shape) if it exists

        print("Draw a rectangle")  # Print a message indicating drawing a rectangle

        return

# Create a list 'shapes' containing instances of 'circle' and 'rectangle'

shapes = [circle(), rectangle()]

# Iterate through each shape in the 'shapes' list and call the 'draw' method

for shp in shapes:

    shp.draw()

**8.1 Python - Dynamic Binding**

In object-oriented programming, the concept of dynamic binding is closely related to polymorphism. In Python, dynamic binding is the process of resolving a method or attribute at runtime, instead of at compile time.

According to the polymorphism feature, different objects respond differently to the same method call based on their individual implementations. This behavior is achieved through method overriding, where a subclass provides its own implementation of a method defined in its superclass.

The Python interpreter determines which is the appropriate method or attribute to invoke by based on the object's type or class hierarchy at runtime. This means that the specific method or attribute to be called is determined dynamically, based on the actual type of the object.

#Example 1

class shape:

   def draw(self):

      print ("draw method")

      return

class circle(shape):

   def draw(self):

      print ("Draw a circle")

      return

class rectangle(shape):

   def draw(self):

      print ("Draw a rectangle")

      return

shapes = [circle(), rectangle()]

for shp in shapes:

   shp.draw()

**Note:**

As you can see, the draw() method is bound dynamically to the corresponding implementation based on the object's type. This is how dynamic binding is implemented in Python.

## 8.1.1 Python - Duck Typing

## Duck Typing

Another concept closely related to dynamic binding is **duck typing**. Whether an object is suitable for a particular use is determined by the presence of certain methods or attributes, rather than its type. This allows for greater flexibility and code reuse in Python.

Duck typing is an important feature of dynamic typing languages like Python (Perl, Ruby, PHP, Javascript, etc.) that focuses on an object's behavior rather than its specific type. According to the "duck typing" concept, "If it walks like a duck and quacks like a duck, then it must be a duck."

Duck typing allows objects of different types to be used interchangeably as long as they have the required methods or attributes. The goal is to promote flexibility and code reuse. It is a broader concept that emphasizes on object behavior and interface rather than formal types.

Here is an example of duck typing

# Example of Duck Typing in Python

# Define a class 'circle'

class Circle:

    def draw(self):

        print("Draw circle")

        return

# Define a class 'rectangle'

class Rectangle:

    def draw(self):

        print("Draw rectangle")

        return

# Define a class 'area'

class Area:

    def draw(self):

        print("Calculate area")

        return

# Define a function 'duck\_Function' that takes any object with a 'draw' method

def duck\_Function(obj):

    obj.draw()

# Create instances of the classes

objects = [Circle(), Rectangle(), Area()]

# Print a newline for better output readability

print("\n")

# Iterate through the list of objects

for obj in objects:

    # Call the 'duck\_Function' on each object, which calls the 'draw' method of each object

    duck\_Function(obj)

**9. Python - Method Overriding**

You can always override your parent class methods. One reason for overriding parent's methods is that you may want special or different functionality in your subclass.

#Example 1

class Parent:#define parent class

    def mymethod(self):

        print("Calling Parent method")

class Child(Parent):#defin child class

    def mymethod(self):

        print("Calling child method")

c=Child()#instance of child Class

c.mymethod()#Child calls overriden method

# Example 2

"""To understand inheritance in Python, let us take another example.

We use the following Employee class as a parent class."""

# Define a class 'Employee' as the parent class

class Employee:

    # Constructor to initialize name and salary attributes

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

        self.name = name

        self.salary = salary

    # Method to get the name of the employee

    def getname(self):

        return self.name

    # Method to get the salary of the employee

    def getsalary(self):

        return self.salary

# Define a class 'SalesOffice' that inherits from the 'Employee' class

class SalesOffice(Employee):

    # Constructor to initialize name, salary, and incentive attributes

    def \_\_init\_\_(self, name, salary, inc):

        # Call the constructor of the base class ('Employee') using super()

        super().\_\_init\_\_(name, salary)

        # Additionally, the child class has one more instance variable 'incentive'

        self.incentive = inc

    # Override the getsalary() method to add the incentive to the salary

    def getsalary(self):

        return self.salary + self.incentive

# Create an instance 'obj1' of the 'Employee' class

obj1 = Employee("Shary", 9000)

# Print the total salary for 'obj1' using getname() and getsalary() methods

print("Total salary for {} is Rs {}".format(obj1.getname(), obj1.getsalary()))

# Create an instance 'obj2' of the 'SalesOffice' class with an incentive of 500

obj2 = SalesOffice("Hamad", 10000, 500)

# Print the total salary for 'obj2' using getname() and getsalary() methods

print("Total salary for {} is Rs {}".format(obj2.getname(), obj2.getsalary()))

Base Overridable Methods

The following table lists some generic functionality of the object class, which is the parent class for all Python classes. You can override these methods in your own class −

|  |  |
| --- | --- |
| **Sr.No** | **Method, Description & Sample Call** |
| 1 | **\_\_init\_\_ ( self [,args...] )**  Constructor (with any optional arguments)  Sample Call : *obj = className(args)* |
| 2 | **\_\_del\_\_( self )**  Destructor, deletes an object  Sample Call : *del obj* |
| 3 | **\_\_repr\_\_( self )**  Evaluatable string representation  Sample Call : *repr(obj)* |
| 4 | **\_\_str\_\_( self )**  Printable string representation  Sample Call : *str(obj)* |

**10. Python - Method Overloading**

Method overloading is an important feature of object-oriented programming. Java, C++, C# languages support method overloading, but in Python it is not possible to perform method overloading.

When you have a class with method of one name defined more than one but with different argument types and/or return type, it is a case of method overloading. Python doesn't support this mechanism as the following code shows

#Example 1

class Add:

    def add(self,a,b):

        result=a+b

        return result

    def add(self,a,b,c):

        result=a+b+c

        return result

obj1=Add()

print("Sum(three para): ",obj1.add(10,20,30))

#print("Sum(two para): ",obj1.add(10,20)) Show error

"""

Output

The first call to add() method with three arguments is successful.

However, calling add() method with two arguments as defined in the class fails.

Python considers only the latest definition of add() method, discarding the earlier definitions.

"""

#Solve this issue

"""

To simulate method overloading, we can use a workaround by defining default value to method arguments as None,

so that it can be used with one, two or three arguments.

"""

class Add:

    def add(self,a=None,b=None,c=None):

        results=0

        if a!=None and b!=None and c!=None:

            results=a+b+c

        elif (a!=None and b!=None and c==None):

            results=a+b

        return results

obj2=Add()

print("Sum of Three number: ",obj2.add(10,20,30))

print("Sum of Two number: ",obj2.add(10,20))

#With this workaround, we are able to incorporate method overloading in Python class.

"""

Python's standard library doesn't have any other provision for implementing method overloading.

However, we can use dispatch function from a third party module named MultipleDispatch for this purpose.

This module has a @dispatch decorator. It takes the number of arguments to be passed to the method to be overloaded.

Define multiple copies of add() method with @dispatch decorator as below −

    """

#Example 3

from multipledispatch import dispatch

class Add:

    @dispatch(int,int)

    def add(self,a,b):

        result=a+b

        return result

    @dispatch(int,int,int)

    def add(self,a,b,c):

        result=a+b+c

        return result

obj1=Add()

print("\nSum(three para): ",obj1.add(10,20,30))

print("Sum(two para): ",obj1.add(10,20))

**Python - Dynamic Typing**

* One of the standout features of Python language is that it is a dynamically typed language.
* The compiler-based languages C/C++, Java, etc. are statically typed.
* Let us try to understand the difference between static typing and dynamic typing.
* In a statically typed language, each variable and its data type must be declared before assigning it a value.
* Any other type of value is not acceptable to the compiler, and it raises a compile-time error.

**11. Python - 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 in application development.

**There are two types of abstraction**.

1. **Data 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.

1. **Process abstraction**

Other type is called process abstraction. It refers to hiding the underlying implementation details of a process.

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

To form an abstract class in Python, it must inherit ABC class that is defined in the 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

#Example 1

from abc import ABC,abstractmethod

class demo(ABC):

    @abstractmethod

    def methed1(self):

        print("Abstact Method")

        return

    def methed2(self):

        print("Concrete Method")

#obj1=demo()    #TypeError: Can't instantiate abstract class demo with abstract method method1

"""

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 this error −

TypeError: Can't instantiate abstract class concreteclass with abstract method met

"""

#the child class with the abstract method overridden is given in the following example

from abc import ABC,abstractmethod

class demo(ABC):

    @abstractmethod

    def methed1(self):

        print("Abstact Method")

        return

    def methed2(self):

        print("Concrete Method")

class concreteClass(demo):

    def methed1(self):

        super().methed1()

        return

obj2=concreteClass()

obj2.methed1()

obj2.methed2()

**12. Python - Encapsulation**

The principle of Encapsulation is one of the main pillars on which the object-oriented programming paradigm is based. Python takes a different approach towards the implementation of encapsulation.

We know that a class is a user-defined prototype for an object. It defines a set of data members and methods, capable of processing the data. According to principle of data encapsulation, the data members that describe an object are hidden from environment that is external to class. They are available for processing to methods defined within the class only. Methods themselves on the other hand are accessible from outside class context. Hence object data is said to be encapsulated by the methods. The result of such encapsulation is that any unwarranted access to the object data is prevented.

Languages such as C++ and Java use access modifiers to restrict access to class members (i.e., variables and methods). These languages have keywords public, protected, and private to specify the type of access.

A class member is said to be public if it can be accessed from anywhere in the program. Private members are allowed to be accessed from within the class only. Usually, methods are defined as public and instance variable are private. This arrangement of private instance variables and public methods ensures the implementation of encapsulation.

Unlike these languages, Python has no provision to specify the type of access that a class member may have. By default, all the variables and methods in a Python class are public, as is demonstrated by the following example.

**Access modifiers:**

A Class in Python has three types of access modifiers:

* **Public Access Modifier**
* **Protected Access Modifier**
* **Private Access Modifier**

1. **Public Access Modifier:**

The members of a class that are declared public are easily accessible from any part of the program. All data members and member functions of a class are public by default.

## Protected Access Modifier:

The members of a class that are declared protected are only accessible to a class derived from it. Data members of a class are declared protected by adding a single underscore ‘\_’ symbol before the data member of that class.

## Private Access Modifier:

The members of a class that are declared private are accessible within the class only, private access modifier is the most secure access modifier. Data members of a class are declared private by adding a double underscore ‘\_\_’ symbol before the data member of that class.

Example 1

Here, we have an Employee class with instance variables, **name** and **age**. An object of this class has these two attributes. They can be directly accessed from outside the class, because they are public.

#Example 1

class Student:

    def \_\_init\_\_(self,name="Shary",marks=50,age=24):

        self.name=name

        self.\_\_age=age

        self.\_marks=marks

    def studentData(self):

        print("Name: {} age: {} marks: {} ".format(self.name,self.\_\_age,self.\_marks))

obJ1=Student()

obj2=Student("Umair",300,20)

obJ1.studentData()

obj2.studentData()

print(f"\nName: {obj2.name}")#public access in anywhere

print(f"Marks: {obj2.\_marks}")#protected access in out of class but not access out of package

#print(f"Age: {obj2.\_\_age}") not Access out of class

#Private data access

print(obj2.\_Student\_\_age)

**13. Python - Interfaces**

In software engineering, an interface is a software architectural pattern. An interface is like a class but its methods just have prototype signature definition without any body to implement. The recommended functionality needs to be implemented by a concrete class.

In languages like Java, there is interface keyword which makes it easy to define an interface. Python doesn't have it or any similar keyword. Hence the same ABC class and @abstractmethod decorator is used as done in an abstract class.

An abstract class and interface appear similar in Python. The only difference in two is that the abstract class may have some non-abstract methods, while all methods in interface must be abstract, and the implementing class must override all the abstract methods.

#Example 1

from abc import ABC,abstractmethod

class demointerface(ABC):

    @abstractmethod

    def method1(self):

        print("Abstract method1")

        return

    @abstractmethod

    def method2(self):

        print("Abstract method2")

        return

class ConcreteClass(demointerface):

    def method1(self):

        #super().method1()

        print("concrete Method1")

        return

    def method2(self):

        #super().method2()

        print("concrete Method2")

        return

# obj1=demo() TypeError: Can't instantiate abstract class demo with abstract methods method1, method2

obj2=ConcreteClass()

obj2.method1()

obj2.method2()

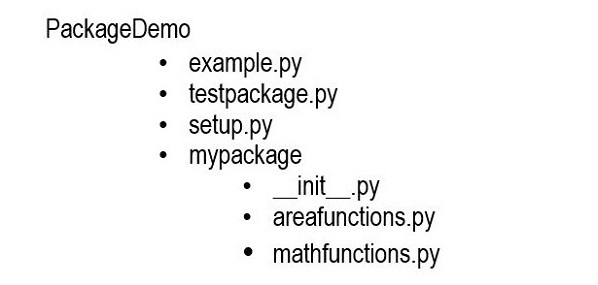
**Python - Packages**

In Python, module is a Python script with .py extension and contains objects such as classes, functions etc. Packages in Python extend the concept of modular approach further. Package is a folder containing one or more module files; additionally a special file "\_\_init\_\_.py" file which may be empty but may contain the package list.

Let us create a Python package with the name mypackage. Follow the steps given below −

* Create an outer folder to hold the contents of mypackage. Let its name be packagedemo.
* Inside it, create another folder mypackage. This will be the Python package we are going to construct.Two Python modules areafunctions.py and mathfunctions.py will be created inside mypackage.
* Create an empty "\_\_.init\_\_.py" file inside mypackage folder.
* Inside the outer folder, we shall later on store a Python script example.py to test our package.

The **file/folder structure** should be as shown below –



**Example.py**

#imoprt rectangle (function) from areafunction(filename) from mypackage fulder

from mypackage.areafunction import rectangle

print("Area: ",rectangle(10,20))

from mypackage.mathfunction import avg

print("Average of two numbers: ",avg(10,20))

**testpackage.py**

from mypackage import pow,circle

print("Area of circle: ",circle(5))

print("10 raised to 2: ",pow(10,2))

**Setup.py**

from setuptools import setup

setup(name='mypackage',

      version='0.1',

      description='Package setup script',

      url='#',

      author='Sharykhan',

      author\_email='shahriyarkhan@gmail.com',

      license='MIT',

      packages=['mypackage'],

      zip\_safe=False)

**inner fulder**

**\_\_Init\_\_.py**

from .areafunction import circle

from .mathfunction import sum,pow

**areafunction.py**

#Areafunction

def rectangle(w,h):

    area=w\*h

    return area

def circle(r):

    import math

    area=math.pi\*math.pow(r,2)

    return area

**mathfunction.py**

#Mathfunction

def sum(x,y):

    val=x+y

    return val

def avg(x,y):

    val=(x+y)/2

    return val

def pow(x,y):

    val=x\*\*y

    return val

Using your favorite code editor, save the following two Python modules in mypackage folder.

## Package Installation

Right now, we are able to access the package resources from a script just above the package folder. To be able to use the package anywhere in the file system, you need to install it using the PIP utility.

First of all, save the following script in the parent folder, at the level of package folder.

#setup.py

from setuptools import setup

setup(name='mypackage',

version='0.1',

description='Package setup script',

url='#',

author='anonymous',

author\_email='test@gmail.com',

license='MIT',

packages=['mypackage'],

zip\_safe=False)

Run the PIP utility from command prompt, while remaining in the parent folder.

**python setup.py install**

**15. Python - Inner Classes**

A class defined inside another class is known as an inner class in Python. Sometimes inner class is also called nested class. If the inner class is instantiated, the object of inner class can also be used by the parent class. Object of inner class becomes one of the attributes of the outer class. Inner class automatically inherits the attributes of the outer class without formally establishing inheritance.

Syntax

class outer:

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

pass

class inner:

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

pass

An inner class lets you group classes. One of the advantages of nesting classes is that it becomes easy to understand which classes are related. The inner class has a local scope. It acts as one of the attributes of the outer class.

Example

In the following code, we have student as the outer class and subjects as the inner class. The \_\_init\_\_() constructor of student initializes name attribute and an instance of subjects class. On the other hand, the constructor of inner subjects class initializes two instance variables sub1, sub2.

A show() method of outer class calls the method of inner class with the object that has been instantiated.

# Define the Student class

class Student:

    # Constructor to initialize the student object with a name

    def \_\_init\_\_(self, name):

        # Assign the provided name to the 'name' attribute

        self.name = name

        # Create an instance of the subjects class and assign it to the 'subs' attribute

        self.subs = self.subjects()

    # Method to display the student's name and subjects

    def show(self):

        # Print the student's name and call the display method of the subjects class

        print(f"Name: {self.name} Subjects: {self.subs.display()}")

    # Inner class for representing subjects

    class subjects:

        # Constructor to initialize subjects (HCI and MAD)

        def \_\_init\_\_(self):

            # Assign the subject names to 'sub1' and 'sub2' attributes

            self.sub1 = "HCI"

            self.sub2 = "MAD"

        # Method to display subjects

        def display(self):

            # Return a formatted string representing the subjects

            return f"{self.sub1}, {self.sub2}"

# Create an instance of the Student class with the name "Shary"

obj1 = Student("Shary")

# Call the show method to display information about the student and subjects

obj1.show()

"""

It is quite possible to declare an object of outer class independently,

and make it call its own display() method.

    """

sub=Student("Umair").subjects().display()

**16.Python - Anonymous Class and Objects**

Python's built-in type() function returns the class that an object belongs to. In Python, a class, both a built-in class or a user-defined class are objects of type class.

#Example

class Employee:

    def \_\_init\_\_(self):

        self.var=10

        return

obj=Employee()

print("class of int",type(int))

print("class of list",type(list))

print("class of dict",type(dict))

print("class of Employee",type(Employee))

The type() has a three argument version as follows −

### Syntax

newclass=type(name, bases, dict)

Using above syntax, a class can be dynamically created. Three arguments of type function are −

* **name** − name of the class which becomes \_\_name\_\_ attribute of new class
* **bases** − tuple consisting of parent classes. Can be blank if not a derived class
* **dict** − dictionary forming namespace of the new class containing attributes and methods and their values.

We can create an anonymous class with the above version of type() function. The name argument is a null string, second argument is a tuple of one class the object class (note that each class in Python is inherited from object class). We add certain instance variables as the third argument dictionary. We keep it empty for now.

#Example

class Employee:

    def \_\_init\_\_(self):

        self.var=10

        return

obj=Employee()

print("class of int",type(int))

print("class of list",type(list))

print("class of dict",type(dict))

print("class of Employee",type(Employee))

#

anon=type('',(object,),{})

#To create an object of this anonymous class −

obj=anon()

#The result shows that the object is of anonymous class

print("types of obj:",type(obj))

#The result shows that the object is of anonymous class

#Example 2

"""

We can also add instance variables and instance methods dynamically. Take a look at this example −

    """

# Define a function getA to be used as an instance method

def getA(self):

    # Return the value of the 'a' attribute

    return self.a

# Use the type function to dynamically create a class

# Arguments: class name (''), base classes (object,), and a dictionary with attributes and methods

# This dynamically created class has attributes 'a', 'b', and 'c', and methods 'getA' and 'getB'

# The 'getA' method is defined using the getA function, and the 'getB' method is defined using a lambda function

obj1 = type('', (object,), {'a': 3, 'b': 5, 'c': 6, 'getA': getA, 'getB': lambda self: self.b})()

# Call the getA and getB methods of the dynamically created class and print the results

print(obj1.getA(), obj1.getB())

In Python, an anonymous function is a function that is defined without a name. Anonymous functions are created using the **lambda** keyword. They are also sometimes referred to as lambda functions.

The general syntax for a lambda function is as follows:

pythonCopy code

lambda arguments: expression

Here, **lambda** is the keyword, **arguments** are the input parameters, and **expression** is the result of the function. Lambda functions are often used for short, simple operations where a full function definition is not necessary.

Here's an example of a lambda function that adds two numbers:

pythonCopy code

add = lambda x, y: x + y

result = add(3, 5)

print(result) # Output: 8

**17\_Python - Singleton Class**

A Singleton class is a class of which only one object can be created. This helps in optimizing memory usage when you perform some heavy operation, like creating a database connection.

# Define a class named 'singleton'

class singleton:

    # Class variable to hold the single instance of the class

    \_instance = None

    # Define a special method '\_\_new\_\_' which is called to create a new instance of the class

    def \_\_new\_\_(cls):

        # Check if the class variable '\_instance' is None (indicating no instance has been created yet)

        if cls.\_instance is None:

            print("Creating the object")

            # Create a new instance of the class using the super() function

            cls.\_instance = super(singleton, cls).\_\_new\_\_(cls)

        # Return the existing instance if it already exists, otherwise, return the new instance

        return cls.\_instance

# Create an instance of the 'singleton' class

obj1 = singleton()

# Print the instance (memory address) of the created object

print(obj1)

# Create another instance of the 'singleton' class

obj2 = singleton()

# Print the instance (memory address) of the second object

print(obj2)

This is how the above code works −

When an instance of a Python class declared, it internally calls the \_\_new\_\_() method. We override the \_\_new\_\_() method that is called internally by Python when you create an object of a class. It checks whether our instance variable is None. If the instance variable is None, it creates a new object and call the super() method and returns the instance variable that contains the object of this class.

If multiple objects are created, it becomes clear that the object is only created the first time; after that, the same object instance is returned.

Creating the object

<\_\_main\_\_.SingletonClass object at 0x000002A5293A6B50>

<\_\_main\_\_.SingletonClass object at 0x000002A5293A6B50>

**18. Python - Enums**

The term 'enumeration' refers to the process of assigning fixed constant values to a set of strings, so that each string can be identified by the value bound to it. Python's standard library offers the **enum** module. The Enum class included in **enum** module is used as the parent class to define enumeration of a set of identifiers − conventionally written in upper case.

#EXample 1

from enum import Enum

class subjects(Enum):

    English=1

    Maths=2

    HCI=3

    MAD=4

"""

In the above code, "subjects" is the enumeration.

It has different enumeration members, e.g., subjects.MATHS. Each member is assigned a value.

Each member is ab object of the enumeration class subjects,

and has name and value attributes

"""

obj1=subjects.Maths

obj2=subjects.HCI

print(type(obj1),obj1.value)

print(type(obj2),obj2.value)

#Example 2 Value bound to the enum member needn't always be an integer, it can be a string as well.

class subjects(Enum):

    English="E"

    Maths="M"

    HCI="H"

    MAD="MD"

obj1=subjects.Maths

obj2=subjects.HCI

print("\n",obj1.name,obj1.value)

print(obj2.name,obj2.value,"\n")

#Example 3  You can iterate through the enum members in the order of their appearance in the definition

for sub in subjects:

    print(sub.name,sub.value)

#Example 4

"""

An enum class cannot have same member appearing twice,

however, more than one members may be assigned same value.

To ensure that each member has a unique value bound to it,

use the @unique decorator."""

from enum import Enum,unique

#@unique

class subjects(Enum):

       ENGLISH = 1

       MATHS = 2

       GEOGRAPHY = 3

       HCI = 2

obj3=subjects.MATHS

print(obj3.name,obj3.value,"\n")

"""

unique

    raise ValueError('duplicate values found in

%r: %s' %

ValueError: duplicate values found in <enum 'subjects'>: HCI -> MATH

    """

#Example 5

"""The Enum class is a callable class, hence you can use the following alternative method of defining enumeration −

"""

from enum import Enum

subjects1 = Enum("subjects", "ENGLISH MATHS SCIENCE SANSKRIT")

for sub in subjects1:

    print(sub.name)

**19. Python - Reflection**

In object-oriented programming, reflection refers to the ability to extract information about any object in use. You can get to know the type of object, is it a subclass of any other class, what are its attributes and much more. Python's standard library has a number of functions that reflect on different properties of an object. Reflection is also sometimes called introspect.

Let us take a review of reflection functions.

The type() Function

We have used this function many times. It tells you which class does an object belong to.

#Example1 Following statements print the respective class of different built-in data type objects

print (type(10))

print (type(2.56))

print (type(2+3j))

print (type("Hello World"))

print (type([1,2,3]))

print (type({1:'one', 2:'two'}))

## The isinstance() Function

This is another built-in function in Python which ascertains if an object is an instance of the given class

### Syntax

isinstance(obj, class)

This function always returns a Boolean value, true if the object is indeed belongs to the given class and false if not.

#EXample 2 Let us verify the type of an object of a user-defined class −

class test:

    pass

obj1=test()

print(type(obj1))

#EXample 3

print(isinstance(10,int))

print (isinstance(2.56,int))

print (isinstance(2+3j,complex))

print('\n')

print (isinstance([1,2,3], tuple))

print (isinstance({1:'one', 2:'two'}, set))

## The issubclass() Function

This function checks whether a class is a subclass of another class. Pertains to classes, not their instances.

As mentioned earlier, all Python classes are subclassed from object class. Hence, output of following print statements is True for all

#Example 3

print("\n")

print (issubclass(int, object))

print (issubclass(str, object))

print (issubclass(test, object))

## The callable() Function

An object is callable if it invokes a certain process. A Python function, which performs a certain process, is a callable object. Hence callable(function) returns True. Any function, built-in, user defined or a method is callable. Objects of built-in data types such as int, str, etc., are not callable.

#Example 4 callable fuction

print("\n")

print(callable("Hello"))

print(callable(abs))

print(callable(test))

A string object is not callable. But abs is a function which is callable. The pop method of list is callable, but clear() is actually call to the function and not a function object, hence not a callable

## The getattr() Function

The getattr() built-in function retrieves the value of the named attribute of object.

#Example gettattr() function

class test:

    def \_\_init\_\_(self):

        self.name="Shary"

obj2=test()

print("Name of the attribute:",getattr(obj2,'name'))

## The setattr() Function

The setattr() built-in function adds a new attribute to the object and assigns it a value. It can also change the value of an existing attribute.

In the example below, the object of test class has a single attribute − name. We use setattr to add age attribute and to modify the value of name attribute.

#Example 6 settattr()

#Create new attribute

setattr(obj2,'Age',20)

print("Name:",obj2.name,"Age:",obj2.Age)

setattr(obj2,'name','sharyKhan')

print("Name:",obj2.name,"Age:",obj2.Age)

## The hasattr() Function

This built-in function returns True if the given attribute is available to the object argument, and false if not. We use the same test class and check if it has a certain attribute or not.

#Example 7

print(hasattr(obj2,'Age'))

print(hasattr(obj2,'name'))

print(hasattr(obj2,'marsk'))

## The dir() Function

If his built in function called without an argument, return the names in the current scope. Fpr any object as argument, it returns a list the attributes of the given object, and of attributes reachable from it.

* **For a module object** − the function returns the module's attributes.
* **For a class object** − the function returns its attributes, and recursively the attributes of its bases.
* **For any other object** − its attributes, its class's attributes, and recursively the attributes of its class's base classes.

#Example 8 dir()

print("\ndir(int)",dir(int))

print("\ndir(dict)",dir(dict))