**#Overloading & Overriding**

#**Over-riding/Runtime polymorphism in general (Same method name, Same parameters)**

* **Whenever an object is bound with the functionality at run time, this is known as runtime polymorphism.**
* Method overriding is concept **where even though the method name and parameters passed is similar, the behaviour is different based on the type of object.**
* **methods or functions must have the same name and same signatures.**
* **Method overriding is an example of run time polymorphism.**

**#Implementation of overriding with classes**

**#Note: Two different classes are created**

Method overriding

class A:

    def greet(self):

        print("This is greet of A")

class B(A):

    def greet(self):

        print("This is greet of B")

a1=A()

b1=B()

a1.greet()

b1.greet()

**#Over-Loading** (**Same method name, Different parameters)**

**(By default, not supported by python)**

(When more than **one method with same name is defined in same class**, is known as method overloading)

In python, if method is written such that it can perform more than one task, then it is called over-loading.

**#Operator overloading**

+ sign can be used for both concat and addition

**#Method overloading**

**#Method name should be same, arguments must be different**

#Overloading

#create a class called overload with a method in it to add numbers

class overload:

    def display(self,a=None,b=None):

    print(a,b)

obj=overload()

#calling same method with different arguments

#call function with no parameters

obj.display()

#call function with one parameter

obj.display(10)

#call function with two parameter

obj.display(10,20)

| S.NO | Method Overloading | Method Overriding |
| --- | --- | --- |
| 1. | In the method overloading, methods or functions must have the same name and different signatures. | Whereas in the method overriding, methods or functions must have the same name and same signatures. |
| 2. | Method overloading is a example of compile time polymorphism. | Whereas method overriding is a example of run time polymorphism. |
| 3. | In the method overloading, inheritance may or may not be required. | Whereas in method overriding, inheritance always required. |
| 4. | Method overloading is performed between methods within the class. | Whereas method overriding is done between parent class and child class methods. |
| 5. | It is used in order to add more to the behavior of methods. | Whereas it is used in order to change the behavior of exist methods. |
| 6. | In method overloading, there is no need of more than one class. | Whereas in method overriding, there is need of at least of two classes. |

**Encapsulation**

Encapsulation is used **to hide the values or state of a structured data object** inside a class, preventing unauthorized parties' direct access to them.

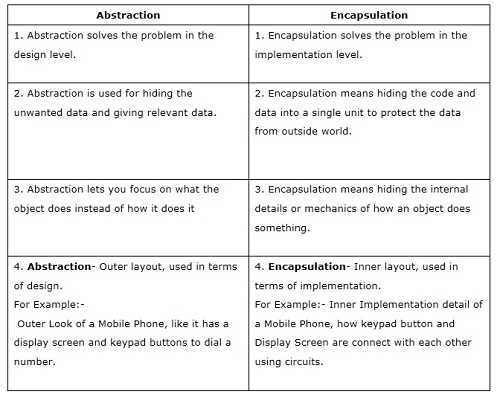
**#Encapsulation vs Abstraction**

Abstraction is the method of hiding the unwanted information.

In abstraction, implementation complexities are hidden using abstract classes and interfaces.

Whereas encapsulation is a method to hide the data in a single entity or unit along with a method to protect information from outside.

While in encapsulation, the data is hidden using methods of getters and setters.



#abstraction

#Abstraction is used to hide the internal functionality of the function from the users.

# The users only interact with the basic implementation of the function,

#  but inner working is hidden. User is familiar with that "what function does"

#  but they don't know "how it does."

#In Python, an abstraction is used to hide the irrelevant data/class

# in order to reduce the complexity.

# It also enhances the application efficiency.

#By defining an abstract base class,

#you can define a common Application Program Interface(API) for a set of subclasses.

# abstract base class work

from abc import ABC, abstractmethod

@abstractmethod

class Car(ABC):

    def mileage(self):

        pass

class Suzuki(Car):

    def mileage(self):

        print("The milage is 25kmph")

class Tesla(Car):

    def mileage(self):

        print("The milage is 26kmph")

c=Car()

s=Suzuki()

t=Tesla()

print(t.mileage())

**CLASS**

#classes , constructor, inhertiance

#OOP CONCEPTS

#4 PILLARS

#1:Inheritance

#2:Polymorphism,overloading,overriding

#3:Encapsulation

#4:Abstraction

'''

A class is a collection of objects.

A class contains the blueprints or the prototype from which the objects are being created.

It is a logical entity that contains some attributes and methods.

'''

'''

#case1: simple class for person

class Person:

    def greet(self):

        print("hi my name is",self.name)

person1=Person()

person1.greet()

'''

#case 2:class with constructor

#The \_\_init\_\_ method is similar to constructors in C++ and Java.

#It is run as soon as an object of a class is instantiated.

#The method is useful to do any initialization you want to do with your object.

'''

class Person:

    def \_\_init\_\_(self,name,age):

        self.name=name

        self.age=age

    #greet method

    def greet(self):

        print("Hi my name is",self.name,"and am",self.age,"years old.")

#create instane/object

person1=Person("shreyank",23)

person1.greet()

'''

#Inheritance

'''

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

It represents real-world relationships well.

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

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

'''

#Simple Inheritance/Single Inheritance

'''

class A:

    def functionA(self):

        print("This is functionA")

class B(A):

    def functionB(self):

        print("This is functionB")

#create object/instance

a1=A()

b1=B()

print(b1.functionA())

'''

#Multiple Inheritance

# When a class can be derived from more than one base class this type of inheritance is called multiple inheritance.

# In multiple inheritance, all the features of the base classes are inherited into the derived class.

'''

class A:

    def functionA(self):

        print("This is functionA")

class B:

    def functionB(self):

        print("This is functionB")

class C(A,B):

    def functionC(self):

        print("This is functionC")

#create object/instance

a1=A()

b1=B()

c1=C()

print(c1.functionC())

'''

#Multi-Level Inheritance

#class B derives from class A & Class C derives from B

'''

class A:

    def functionA(self):

        print("This is functionA")

class B(A):

    def functionB(self):

        print("This is functionB")

class C(B):

    def functionC(self):

        print("This is functionC")

#create object/instance

a1=A()

b1=B()

c1=C()

print(c1.functionC())

'''

#Polymorphism

#It refers to the use of a single type entity (method, operator or object) to represent different types in different scenarios.

#demo with classes

# Same function name in different class but diffrent behaviour

'''

Method overriding

is concept where even though the method name and parameters passed is similar, the behavior is different based on the type of objec

class A:

    def greet(self):

        print("This is greet of A")

class B:

    def greet(self):

        print("This is greet of B")

a1=A()

b1=B()

a1.greet()

b1.greet()

'''

#other example is use of len function in python

'''

len function with string--> returns lenght of string

len function with list ---> returns number of items

len function with dict --> returns number of keys

'''

#Encapsulation

#It describes the idea of wrapping data and the methods that work on data within one unit.

#This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data.

# A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc.

#The idea of information hiding is that if you have an attribute that isn’t visible to the outside, you can control the access to its value to make sure your object is always has a valid state.

#Protected vs Private variable

#\_variable is protected

#\_\_variable is private

#Encapsulation in python can be achieved through protected and private members

#public name variable: can be accessed anywhere in program

'''

class En():

    def \_\_init\_\_(self):

        self.course="python"

        self.name="shreyank"

    #method to join both

    def join(self):

        return self.name+self.course

e=En()

#update the name

e.name="not shreyank"

print(e.join())

'''

#now lets convert the name as protected (\_)

#protected members are accessible within the class or its subclasses but not outside them

class En():

    def \_\_init\_\_(self):

        self.course="python"

        self.\_name="shreyank"

        self.\_\_age=23

    #method to join both

    def join(self):

        return self.\_name+self.course+self.\_\_age

    #we can define a getter to acces age

    def getter(self):

        return self.\_\_age

    #we can use setter to change age

    def setter(self):

        self.\_\_age=45

        return self.\_\_age

e=En()

#access

print(e.\_name)

#age is private,cannot be accessed

#print(e.\_\_age)

#using getter to access the age

print(e.getter())

#using setter to change age value

print(e.setter())