OOPs Document

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

class: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.

Some points on Python class:

Classes are created by keyword class.

Attributes are the variables that belong to a class.

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

syntax:

```
class ClassName:
```

Statement-1

...

Statement-N

object:The object is an entity that has a state and behavior associated with it. It may be any real-world object like a mouse, keyboard, chair, table, pen, etc. Integers, strings, floating-point numbers, even arrays, and dictionaries, are all objects.

self:we can access the attributes and methods of the class(current class only)

In [2]:
```
class James():
    def display(self):
        print('this is class')
obj=James()
obj.display()
```

```
this is class
```

In [3]:
```
class John(): #class keyword class name
    a=10 #data member
    def output(self):# method(self)
        print(self.a)
obj=John()# declaring of object
```

```
obj.output()#using objects we can call the methods
```

```
10
```

In [4]:

```
class sam:
    a=10
    def display(self):
        print(self.a)
ram=sam()
john=sam()
ram.display()
john.display()
```

```
10
10
```

Constructor(**init**): Constructors are generally used for instantiating an object.

```
constructors is to initialize(assign values) to the data members of the class when
an object of the class is created.

The __init__() method is called the constructor and is always called when an object
is created.

Multiple constructor are not allowed
```

syntax:

```
def __init__(self):
# body of the constructor
```

In [5]:

```
class Akhil:
    def __init__(self,a,b,c,d):
        self.a=a
        self.b=b
        self.c=c
        self.d=d
    def display(self):
        print(self.a,self.b,self.d,self.d)
obj=Akhil(10,2.2,'nidhu',True)
obj.display()
```

```
10 2.2 True True
```

In [6]:

```
class Bikes:
    def __init__(self,name,company,cc,mil,price):
        self.a=name
        self.b=company
        self.c=cc
        self.d=mil
        self.e=price
    def bikes_info(self):
        print("Bike Name:",self.a)
        print("Bike Company:",self.b)
        print("Bike CC:",self.c)
        print("Bike mil:",self.d)
        print("Bike price:",self.e)
bn=input("Enter the bike name:")
bc=input("Enter the bike company:")
bcc=int(input("Enter the bike cc:"))
bm=int(input("Enter the bike mil:"))
```

```
bp=float(input("Enter the bike price:"))
bikesobj=Bikes(bn,bc,bcc,bm,bp)
bikesobj.bikes_info()
```

```
Enter the bike name:duke
Enter the bike company:klm
Enter the bike cc:300
Enter the bike mil:35
Enter the bike price:200000
Bike Name: duke
Bike Company: klm
Bike CC: 300
Bike mil: 35
Bike price: 200000.0
```

Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class.

Parent class is the class being inherited from, also called base class.

Child class is the class that inherits from another class, also called derived class.

Single inheritance:

A derived class is derived only from a single parent class and allows the class to derive behaviour and properties from a single base class.

In [7]:
```
class Parent:
    def output(self):
        print('this is parent class')
class Child(Parent):
    def outputChild(self):
        print('this is child class')
i=Child()
i.output()
i.outputChild()
```

```
this is parent class
this is child class
```

Multiple inheritance:

When a class is derived from more than one base class it is called multiple Inheritance. The derived class inherits all the features of the base case.

In [8]:
```
class Father:
    def output(self):
        print('this is parent class')
class Mother:
    def outputM(self):
        print('this is mother class')
class Child(Father,Mother):
    def outputChild(self):
        print('this is child class')
i=Child()
i.output()
```

```
i.outputM()
i.outputChild()
```

```
this is parent class
this is mother class
this is child class
```

Multilevel inheritance:

Features of the base class and the derived class are further inherited into the new derived class

In [9]:

```
class GrandFather:
    def output(self):
        print('this is grandfather class')
class Father(GrandFather):
    def outputF(self):
        print('this is father class')
class Child(Father):
    def outputChild(self):
        print('this is child class')
i=Child()
i.output()
i.outputF()
i.outputChild()
```

```
this is grandfather class
this is father class
this is child class
```

Hierarchical Inheritance:

When more than one derived classes are created from a single base this type of inheritance is called hierarchical inheritance

In [10]:

```
class Father:
    def output(self):
        print('this is father class')
class Child1(Father):
    def outputF(self):
        print('this is child1 class')
class Child2(Father):
    def outputChild(self):
        print('this is child2 class')
i=Child1()
i.output()
i.outputF()
i=Child2()
i.output()
i.outputChild()
```

```
this is father class
this is child1 class
this is father class
this is child2 class
```

Polymorphism

Polymorphism:The word polymorphism means having many forms. In programming, polymorphism means the same function name (but different signatures) being used for different types.

Method overloading: The problem with method overloading in Python is that we may overload the methods but can only use the latest defined method.

In [11]:
```
class Methodoverload:
    def Something(self,a=None,b=None,c=None):
        print(a,b,c)
obj=Methodoverload()
obj.Something(1,2,3)
obj.Something(1,2)
obj.Something(1)
obj.Something()
```

```
1 2 3
1 2 None
1 None None
None None None
```

Method overriding: Method overriding is an ability of any object-oriented programming language that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes

When a method in a subclass has the same name, same parameters or signature and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class.

In [12]:
```
class Methodoverride:
    def display(self):
        print('this is parent class')
class Child(Methodoverride):
    def display(self):
        print('this is child class')
        super().display()
obj=Child()
obj.display()
```

```
this is child class
this is parent class
```

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. To prevent accidental change, an object's variable can only be changed by an object's method. Those types of variables are known as private variables.

In [13]:
```
class GFather:
    def __init__(self,a):
        self._y=a
        print(self._y)
class Father(GFather):
    def display1(self):
        print(self._y)
class Child2(Father):
    def display2(self):
        print('Child2',self._y)
obj=Child2(12)
obj.display2()
obj.display1()
```

```
12
Child2 12
12
```

Abstarction: using an abstract class, a class can derive identity from another class without any object inheritance.

abc class cannot create object

In [14]:

```
from abc import ABC,abstractmethod
class Parent(ABC):
    def done(self):
        pass
    def nidhu(self):
        pass
class Child(Parent):
    def done(self,a):
        print('this is child',a)
    def nidhu(self):
        print('this is another class')
obj=Child()
obj.done(10)
obj.nidhu()
```

```
this is child 10
this is another class
```

In [ ]: