In [9]:

```
# to demonstarate the working of class and objects
class Employee:
    id=10
    name="John"
    def display(self):
        print("id=%d \nname=%s"%(self.id,self.name))

emp=Employee()
emp.display()
```

```
id=10
name=John
```

In [11]:

```
#
class Planet:
    def __init__(self,name,number):
        self.name=name
        self.number=number
    def function(self):
        print("I am ",self.name)
        print("I am ",self.number,"planet in the solar system")
obj=Planet("Earth","3")
obj.function()
venus=Planet("venus ","2")
venus.function()
```

```
I am  Earth
I am  3 planet in the solar system
I am  venus
I am  2 planet in the solar system
```

In [12]:

```
class Rect:
    def __init__(self,l,b):
        self.length=l
        self.breadth=b
        print("__init__",self.length,self.breadth)
r1=Rect(10,20)
print(r1.length,r1.breadth)
```

```
__init__ 10 20
10 20
```

In [13]:

```
# when no parameters are passed for invoking a constructor is called non parameterised c
class Person:
    def __init__(self):
        print("without parameters")
p=Person()
```

```
without parameters
```

In [14]:

```
# parameters are passed for invoking a constructor
class Person:
    def __init__(self,name,age):
        self.name=name
        self.age=age
    def display(self):
        print(self.name,self.age)
p=Person("Joey","30")
p.display()
```

```
Joey 30
```

In [15]:

```
class Animal:
    kind='dog'
    def __init__(self,name):
        self.name=name
a=Animal('pinky')
b=Animal('snow')
print(a.kind)#common to both
print(b.kind)#common to both
print(a.name)#unique to a
print(b.name)#unique to b
```

```
dog
dog
pinky
snow
```

In [4]:

```
#classes, objects and functions
class A:
    def __repr__(self):
        return "1"
class B(A):
    def __repr__(self):
        return "2"
class C(B):
    def __repr__(self):
        return "3"
o1 = A()
o2 = B()
o3 = C()
print(o1, o2, o3)
```

```
1 2 3
```

In [ ]:

```
#Inheritance- poweful method in oop
#defines a class that inherits all the function and properties of another class.
# new class is derived class
# one that it inherits from is called base class
#derived class inherits from base class adding new features to it
# promotes reusability
# establishes "is a "relationship between classes
```

In [18]:

```
class Animal:
    name=""
    def eat(self):
        print("I can eat")
class Dog(Animal):
    def display(self):
        print("My name is",self.name)
labrador=Dog()
labrador.name="Rohu"
labrador.eat()# inheritance
labrador.display()
```

```
I can eat
My name is Rohu
```

In [5]:

```
#inheritance
class Demo:
    def check(self):
        return " Demo's check "
    def display(self):
        print(self.check())
class Demo_Derived(Demo):
    def check(self):
        return " Derived's check "
Demo().display()
Demo_Derived().display()
```

```
 Demo's check
 Derived's check
```

In [19]:

```
# If the same method is present in both the classes then the method of the  ubclass over
class Animal:
    name=""
    def eat(self):
        print("i can eat")
class Dog(Animal):
    def eat(self):
        print("I like to eat bones")
obj=Dog()
obj.eat()
```

```
I like to eat bones
```

In [21]:

```
# to access the method of the superclass from the subclass method super() method is used
class Animal:
    name=""
    def eat(self):
        print("i can eat")
class Dog(Animal):
    def eat(self):
        super().eat()
        print("I like to eat bones")
obj=Dog()
obj.eat()
```

```
i can eat
I like to eat bones
```

In [22]:

```
# using of super keyword
class A:
    def test(self):
        print("test of A is called")
class B(A):
    def test(self):
        print("test of B is called ")
        super().test()
class C(A):
    def test(self):
        print("test of C is called")
        super().test()
class D(B,C):
    def test2(self):
        print("test of D is called")
obj=D()
obj.test()
```

```
test of B is called
test of C is called
test of A is called
```

In [23]:

```
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()
```

```
dog barking
Animal Speaking
```

In [7]:

```
#inheritance
class A:
    def one(self):
        return self.two()
    def two(self):
        return 'A'
class B(A):
    def two(self):
        return 'B'
obj1=A()
obj2=B()
print(obj1.two(),obj2.two())
```

```
A B
```

In [24]:

```
# using isinstance method
x=isinstance(5,int)
print(x)
```

```
True
```

In [25]:

```
#using isinstance method
class myObj:
    name="john"
y=myObj()
x=isinstance(y,myObj)
print(x)
```

```
True
```

In [2]:

```
#inheritance
class A:
    def __str__(self):
        return '1'
class B(A):
    def __init__(self):
        super().__init__()
class C(B):
    def __init__(self):
        super().__init__()
def main():
    obj1 = B()
    obj2 = A()
    obj3 = C()
    print(obj1, obj2,obj3)
main()
```

```
1 1 1
```

In [ ]:

```
# binding together of data and functions-encapsulation.
# allows to define a class that inherits all the methods and functions from another clas
# describes a method in the child's class that have all the properties of that in the pa
```

In [27]:

```
class A:
    def __init__(self,name="pqrs"):
        self.__name=name
    # make the getter method
    def getname(self):
        return self.__name
    def setname(self,name):
        self.__name=name
x=A()
print(x.getname())
x.setname("joe")
print(x.getname())
```

```
pqrs
joe
```

In [29]:

```
# acessing public methods to acess private members
class Employee:
    def __init__(self,name,salary):
        self.name=name
        self.__salary=salary
    def show(self):
        print("name",self.name,"salary",self.__salary)
emp=Employee('Jessa ',10000)
emp.show()
```

```
name Jessa  salary 10000
```

In [32]:

```
class Employee:
    #constructor
    def __init__(self,name,salary):
        self.name=name
        self.__salary=salary
emp=Employee('Jessa',100000)
print("name:",emp.name)
print('salary',emp._Employee__salary)
```

```
name: Jessa
salary 100000
```

In [35]:

```
# calling a private member in a public method
class Emp:
    def __init__(self,name,salary):
        self.name=name
        self.__salary=salary
    def show(self):
        print(self.name,self.__salary)
emp=Emp("Maryy","1000000")
emp.show()
```

```
Maryy 1000000
```

In [6]:

```
# accessing private methods in public classes
class A:
    def __init__(self):
        self.__i = 1
        self.j = 5

    def display(self):
        print(self.__i, self.j)
class B(A):
    def __init__(self):
        super().__init__()
        self.__i = 2
        self.j = 7
c = B()
c.display()
```

```
1 7
```

In [36]:

```
#._ is used for name mangling
class Emp:
    def __init__(self,name,age):
        self.name=name
        self.__age=age
e=Emp("abcde",27)
print(e.name,e._Emp__age)
```

```
abcde 27
```

In [1]:

```
# getter and setter method
class Emp:
    def __init__(self,name,age):
        self.name=name
        self.__age=age
    def get_age(self):
        return self.__age
    def set_age(self,age):
        self.__age=age
o=Emp("Jessa",30)
print(o.name,o.get_age())
o.set_age(26)
print(o.name,o.get_age())
```

```
Jessa 30
Jessa 26
```

In [ ]:

```
# Polymorphism - a function can exist in many forms
# Polymorphism means same function names being used for different types.
# promotes function to use entities of different types at different times
```

In [ ]:

```
# Duck typing - a type of the object is less important than the method it defines
# we do not check for types and all instead we check the members and attributes.
# emphasis on what the object really does than what the object is
```

In [38]:

```
class Bird:
    def fly(self):
        print("wings")
class Airplane:
    def fly(self):
        print("fuel")
class Fish:
    def fly(self):
        print("swim")
for obj in Bird(),Airplane(),Fish():
    obj.fly()
```

```
wings
fuel
swim
```

In [ ]:

```
# operator overloading is where the operator behaves differently based on the type of op
#works for builtin classes where the same operator behaves differently for different typ
```

In [39]:

```
class A:
    def display(self):
        print("base class")
class B(A):
    def display(self):
        print("this is derived class")
obj=B()
obj.display()
```

```
this is derived class
```

In [40]:

```
# polymorphism in class methods
class India():
    def capital(self):
        print("Delhi")
    def Language(self):
        print("Hindi and English")
class Usa():
    def capital(self):
        print("Washington D C")
    def Language(self):
        print("English")
obj1=India()
obj2=Usa()
for country in (obj1,obj2):
    country.capital()
    country.Language()
```

```
Delhi
Hindi and English
Washington D C
English
```

In [ ]:

```
# 2 different class types in the same way . here we create a for loop that iterates throu
#which class each method belongs to .
```

In [ ]:

```
# polymorphism with inheritance - method from child class does not fit in the method of p
# process of re-implementing the method of the child class in the parrent class is calle
```

In [ ]:

```
# errors detected during execution are exceptions and are not unconditionally fatal.
#Exceptions should be properly handled or an abrupt termination of program is prevented.
```

In [41]:

```
# try and except blocks
def  Foo(number):
    raise ZeroDivisionError
try:
    Foo(0)
except ArithmeticError:
    print("An error occurred")
print("terminate")
```

```
An error occurred
terminate
```

In [43]:

```
#try except else demonstration
try:
    x=int(input())
    if x>100:
        raise ValueError(x)
except ValueError:
        print(x,"is out of allowed range")
else:
        print(x,"is within allowed range")
```

```
120
120 is out of allowed range
```

In [ ]:

```
# else is an optional block that follows all exception statements and is the part of try
#this should be a part of normal flow and exceptional flow
```

In [44]:

```
#try and except block along with else and finally
def division(n):
    try:
        n = 1 / n
    except ZeroDivisionError:
        print("Division failed")
        n = None
    else:
        print("code is correct")# only is try is correct then the else is executed
    finally:
        print("finally is executed")
        return n
print(division(4))
print(division(0))
```

```
code is correct
finally is executed
0.25
Division failed
finally is executed
None
```

In [54]:

```
#try and except blocks using raise exception.
class MyException(Exception):
    def __init__(self, str):
        self.str = str
    def __str__(self):
        return self.str
n = int(input("enter a number:"))
try:
    if not 1 <= n <= 100 :
        raise MyException("number not in range")
    print("number is fine : ", n)


except MyException as e:
    print(e)
else:
    print("everything is fineee")
print("thats all")
```

```
enter a number:200
number not in range
thats all
```

In [ ]:

```

```

In [ ]:

```

```