how to access the properties from one class into another class?

we can access the properties from one class into another class in two ways,they are

1).Has-A Relationship

2).Is-A Relationship

Has-A Relationship:

in Has-A Relationship,we can access the properties from one class into another class directly by using class name or reference variable.

ex:

```
class test:
    x=10
    def m1(self):
        self.y=20
        print(test.x)
        print(self.y)
class demo:
    a=1
    def m2(self):
        self.b=2
        print(demo.a)
        print(self.b)
        print(test.x)
        t1=test()
        t1.m1()
        print(t1.y)
d1=demo()
d1.m2()
```

output:

```
1
2
10
10
20
20
```

the Has-A relationship is also known as a Association.

the association can be categorized into two types,they are

```
                Aggrigation
                Composition
```

Aggrigation:
----

the week association/relation between the objects,is known as a Aggrigation.

without container object there's a chance to exist the contained object,is known as a Aggrigation.

Car --> Container object

Engine --> Contained object

ex:

```
class Car:
    def __init__(self, engine):
        self.engine = engine
    def __del__(self):
        print("i am in destructor from car")

class Engine:
    def __init__(self):
        pass
    def __del__(self):
        print("i am in destructor from engine")

engine = Engine()
car = Car(engine)
# If I destroy this Car instance,
#the Engine instance still exists.
del car
print("bye")
```

output:

```
C:\Users\DELL\Desktop>python aggrigation.py
i am in destructor from car
bye
i am in destructor from engine
```

Composition:
----

the strong association/relation between the objects,is known as a Composition.

without container object there's no-chance to exist the contained object,is known as a Composition.

Book --> Container object

Pages --> Contained objects

ex:

```
class Book:
    def __init__(self):
       page1 = Page('This is content for page 1')
       page2 = Page('This is content for page 2')
       self.pages = [page1, page2]
    def __del__(self):
        print("i am in destructor for book")

class Page:
    def __init__(self, content):
        self.content = content
    def __del__(self):
        print("i am in destructor for pages")
book = Book()
# If I destroy this Book instance,
# the Page instances are also destroyed
del book
print("bye")
```

output:

```
C:\Users\DELL\Desktop>python composition.py
i am in destructor for book
i am in destructor for pages
i am in destructor for pages
bye
```

Is-A Relationship:
----

in Is-A Relationship,we can access the properties from super class into sub class directly by using sub class name or subclass reference variable.

if any class which is extended by any another class,that class is called Super/Parent/Base class.

if any class which is extending by any another class,that class is called Sub/Child/Derived class.

Is-A relationship is also known as a Inheritance.

```
        java                    python
        ----                    -------
        class x                 class x:
        {                            stmt_1
          stmt_1;                    ......
          ......                     stmt_n
          stmt_n;
         }                      class y(x):
        class y extends x             stmt_1
         {                            ......
           stmt_1;                    stmt_n
           .....
           stmt_n;
         }
```

ex:

```
class test:
    x=10
    def m1(self):
        self.y=20
        print(test.x)
        print(self.y)
class demo(test):
    a=1
    def m2(self):
        self.b=2
        print(demo.a)
        print(self.b)
        print(demo.x)
        print(self.y)
d1=demo()
d1.m1()
d1.m2()
```

output:

```
10
20
1
2
10
20
```

Type's of Inheritances:

In generally,the Inheritance can be categorized into following types,they are

1).Single-Inheritance

2).Multi-Level Inheritance

3).Multiple Inheritance

4).Hierarichical Inheritance

5).Hybrid Inheritance

6).Cyclic Inheritance

Single-Inheritance:
----

the concept of inherit/access/aquire the properties from only one class,is known as a Single-Inheritance.

ex:

```
class x:
    def m1(self):
        print("i am in m1")
class y(x):
    def m2(self):
        print("i am in m2")
y1=y()
y1.m1()
y1.m2()
```

output:

```
i am in m1
i am in m2
```

Multi-Level Inheritance:
----

the concept of inherit the properties from multiple classes into single class with the concept of one after another,is known as a Multi-Level Inheritance.

ex:

```
class x:
    def m1(self):
        print("i am in m1")
class y(x):
    def m2(self):
        print("i am in m2")
class z(y):
    def m3(self):
        print("i am in m3")
class a(z):
    def m4(self):
        print("i am in m4")
a1=a()
a1.m1()
a1.m2()
a1.m3()
```

```
a1.m4()
```

output:

```
-------
i am in m1
i am in m2
i am in m3
i am in m4
```

Multiple Inheritance:

the concept of inherit the properties from multiple classes into single class with the concept of at a time,is known as a multiple Inheritance.

MRO(Method Resolution Order)?

whenever we are implement the Multiple Inheritance concept,in that case we need to identify which class properties are executed first,which class properties are executed second,....,which class properties are executed last by using MRO concept.

```
        MRO=Current class+Left to Right(BFS)
                                 |
                          Current class+Left to Right(BFS)
                                              |
                                            ......
                                            ......
```

to get the MRO(Method Resolution Order) of any class by using mro() or __mro__

ex:

```
---
class x:
    def m1(self):
        print("i am in m1")
class y:
    def m2(self):
        print("i am in m2")
class z:
    def m3(self):
        print("i am in m3")
class a(x,y,z):
    def m4(self):
        print("i am in m4")
a1=a()
a1.m1()
a1.m2()
a1.m3()
a1.m4()
print(a.mro())
print(a.__mro__)
```

```
output:
-------
i am in m1
i am in m2
i am in m3
i am in m4
[<class '__main__.a'>, <class '__main__.x'>, <class '__main__.y'>, <class
'__main__.z'>, <class 'object'>]
(<class '__main__.a'>, <class '__main__.x'>, <class '__main__.y'>, <class
'__main__.z'>, <class 'object'>)

ex2:
----
class x:
    a=10
class y:
    a=5
class z(x,y):
    a=100
print(z.a)

output:
------
100

ex3:
----
class x:
    a=10
class y:
    a=5
class z(x,y):
    pass
print(z.a)

output:
------
10

ex4:
----
class x:
    a=10
class y:
    a=5
class z(y,x):
    pass
print(z.a)
```

output:

```
5
```

Hierarichical Inheritance:

the concept of inherit the properties from single class into multiple classes,is known as a Hierarichical Inheritance.

ex:

```
class x:
    def m1(self):
        print("i am in m1")
class y(x):
    def m2(self):
        print("i am in m2")
class z(x):
    def m3(self):
        print("i am in m3")
class a(x):
    def m4(self):
        print("i am in m4")
y1=y()
y1.m1()
y1.m2()
z1=z()
z1.m1()
z1.m3()
a1=a()
a1.m1()
a1.m4()
```

output:

```
i am in m1
i am in m2
i am in m1
i am in m3
i am in m1
i am in m4
```

Hybrid Inheritance:

it is combination of morethan two type's of inheritances.

ex:

```
class x:
    def m1(self):
        print("i am in m1")
class y(x):
    def m2(self):
        print("i am in m2")
class z(x):
    def m3(self):
        print("i am in m3")
class a(y,z):
    def m4(self):
        print("i am in m4")
a1=a()
a1.m1()
a1.m2()
a1.m3()
a1.m4()
```

output:

```
i am in m1
i am in m2
i am in m3
i am in m4
```

Cyclic Inheritance:
----

the concept of inherit the properties from super class into sub class and vice-versa,is known as a Cyclic Inheritance.

ex:

```
class x(z):
    def m1(self):
        print("i am in m1")
class y(x):
    def m2(self):
        print("i am in m2")
class z(y):
    def m3(self):
        print("i am in m3")
z1=z()
z1.m1()
z1.m2()
z1.m3()
```

output:

```
Traceback (most recent call last):
  File "C:/Python310/e.py", line 1, in <module>
    class x(z):
NameError: name 'z' is not defined
```

```
note:
----
python dont supporting cyclic inheritance.
```