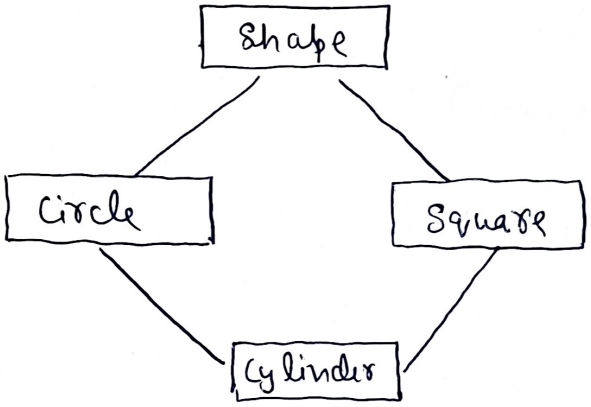
→ "Diamond Problem in Python"

Diamond problem is also known as the "Deadly Diamond of Death". This problem mainly occurs in object-oriented programming in python. It mainly arises when a class inherits from two or more classes that themselves share a common superclass. This situation creates ambiguity in the inheritance hierarchy, leading to confusion about which properties or methods of the shared superclass should be inherited.

Eg :-



Shape → The common superclass representing general properties & behaviours of a shape.

Circle → Intermediate subclass that inherit from shape, each adding specialized attributes or behaviors

Square → Intermediate subclass that inherit from shape, each adding specialized attributes or behaviors

Cylinder → A subclass that inherits from both Circle & Square potentially combining their attributes & behaviors

[P.T.O]

Here is how this hierarchy looks like in Python:

```
class shape:
    def display(self):
        print("class shape")

class circle(shape):
    def display(self):
        print("class circle")

class square(shape):
    def display(self):
        print("class square")

class cylinder(circle, square):
    pass

obj = cylinder()
obj.display()
```

O/p: class circle

Note :→ The given code does not give diamond problem in python becoz the python resolves multiple inheritance dispute using the "Method Resolution order", in short we can call it "MRO"

Problems caused by Diamond Pattern :→

The diamond problem in Python is a problem that can occur when a class inherits from multiple classes that share a common ancestor.

P.T.O

↳ Ambiguity: If cylinder calls a method or accesses an attribute from shape, it is unclear whether to use the version inherited through circle or square.

↳ Duplication: When both subclasses are called, they may both call the base class method, resulting in duplicate calls.

→ Solving Diamond problem in Python:→

Python uses method resolution order (MRO). The MRO provides a clear & consistent sequence in which classes are searched for methods or attributes, ensuring no ambiguity. !

Example :→

```
class Shape:
    def draw(self):
        print("class Shape")
class Circle(Shape):
    def draw(self):
        print("class circle")
class Square(shape):
    def draw(self):
        print("class Square")
class Cylinder(Circle, Square):
    pass

obj = Cylinder()
obj.draw()
print(Cylinder.mro())
```

o/p:→

```
class circle
[<class '__main__.Cylinder'>,
<class '__main__.Circle'>,
<class '__main__.Square'>,
<class '__main__.Shape'>,
<class 'object'>]
```

Using Super():

Python's super() function works with the MRO to ensure that methods in a diamond hierarchy are called in a predictable order. This allows cooperative behaviours among classes.

Eg:-

```
class shape:
    def greet(self):
        print("class shape called")

class circle(shape):
    def greet(self):
        super().greet()
        print("class circle called")

class square(shape):
    def greet(self):
        super().greet()
        print("class square called")

class cylinder(circle, square):
    def greet(self):
        super().greet()
        print("class cylinder called")

d = cylinder()
d.greet()
```

O/P:-

```
class shape called
class square called
class circle called
class cylinder called
```

Note :⇒ The super() function respects the MRO, ensuring the correct order of method calls.

→ "Getter & Setter in Python" :⇒

In python, a getter & setter are methods used to access and update the attributes of a class. These methods provide a way to define controlled access to the attributes of an object. thereby ensuring the integrity of the data. By default, attributes in python can be accessed directly. However, this can pose problems when attributes need validation or transformation before being assigned or retrieved.

↳ Getter : The getter method is used to retrieve the value of a private attribute. It allows controlled access to the attribute.

↳ Setter : The setter method is used to set or modify the value of a private attribute. It allows you to control how the value is updated, enabling validation or modification of the data before it is actually assigned.

There are several ways to implement getter & setter methods :-

→ "Using normal function"

In this approach, getter & setter methods are explicitly defined to get & set the value of a private variable.

The setter allows setting the value & the getter is used to retrieve it.

Eg:-

```
class hello:
    def __init__(self, age = 0):
        self.__age = age

    def get_age(self):          # getter method
        return self.__age

    def set_age(self, x):       # setter method
        self.__age = x

obj = hello()
obj.set_age(21)
print(obj.get_age())
print(obj.__age)
```

O/p. 21
Error

↳ Using property() function :

In this method, the property() function is used to wrap the getter, setter & deleter methods for an attribute, providing a more streamlined approach.

(P.T.O)

```
Class hello:
    def __init__(self):
        self._age = 0
    def get_age(self):
        print("getter method called")
        return self._age
    def set_age(self, a):
        print("setter method called")
        self._age = a
    def del_age(self):
        del self._age

    age = property(get_age, set_age, del_age)

obj = hello()
obj.age = 10
print(obj.age)
```

O/p:- setter method called
getter method called
10

Explanation:-> a hello class is created with an internal _age attribute.
The get_age() function retrieves the value, set_age() sets it & del_age() deletes the attribute.

P.T.O

The property() function binds these methods to the age attribute.

The age attribute is accessed & set using the property function.

→ Using @property decorators:

In this approach, the @property decorator is used for the getter & the @<property_name>.setter decorator is used for the setters. This approach allows a more elegant way to define getter & setter methods.

```
Class hello:
    def __init__(self):
        self._age = 0

    @property
    def age(self):
        print("getter method called")
        return self._age

    @age.setter
    def age(self, a):
        if (a < 18):
            raise ValueError("Sorry your age is below
                eligibility criteria")
        print("setter method called")
        self._age = a
```

(P.T.O)

```
obj = hello()
obj.age = 19
print (obj.age)
```

O/p:-

```
setter method called
getter method called
19
```

Explanation: The hello class uses the @property decorator for the getter method, which returns the age.

The @age.setter decorator is used for the setter method, where a validation (age eligibility) is added.

The setter is used to assign the value, while the getter retrieves it.