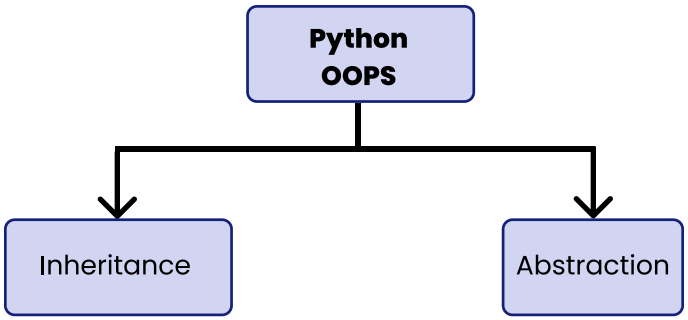
PW SKILLS

Lesson Plan

Inheritance and Abstraction

Topic to covered:

- Inheritance in python
- Abstraction in python



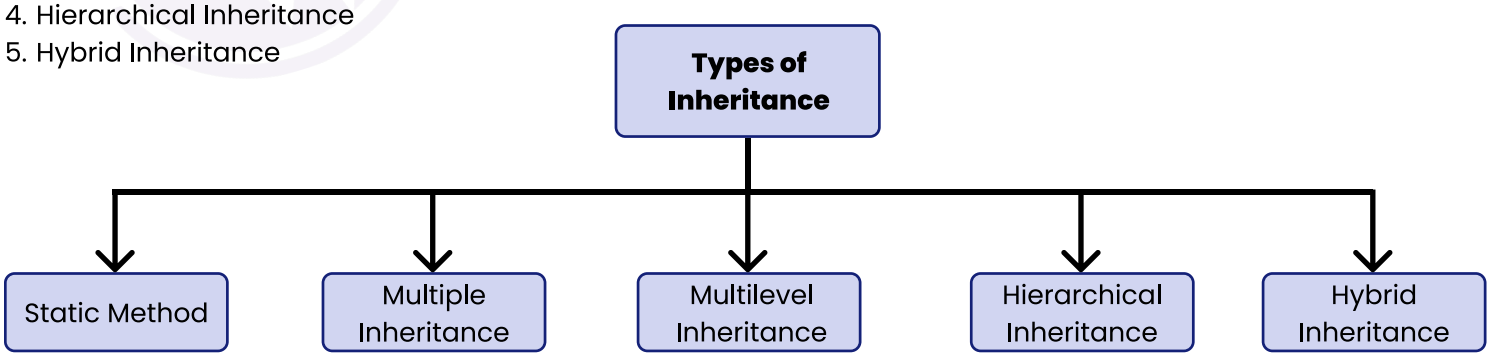
Inheritance : Inheritance plays a significant role in an object-oriented programming language. Inheritance in Python refers to the process of a child class receiving the parent class's properties.
The reuse of code is inheritance's main goal. Instead of starting from scratch when developing a new class, we can use the existing class instead of re-creating it from scratch.

Syntax :

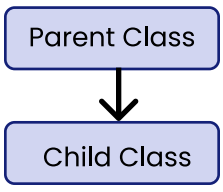
```
class BaseClass:
  # Body of base class
class DerivedClass(BaseClass):
  # Body of derived class
```

Types of Inheritance :

1. Single inheritance
2. Multiple Inheritance
3. Multilevel Inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance



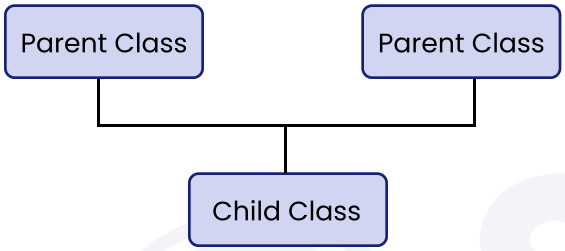
1. Single Inheritance : When a class has only one parent, it is said to have a single inheritance. One class for child and one class for parents are shown here.



```
class Fruit:
  def fruit_info(self):
    print('Inside parent class')

class Apple(Fruit):
  def apple_info(self):
    print("Inside Child class")

obj = Apple()
obj.apple_info()
obj.fruit_info()
```

```
Inside Child class
Inside parent class
```

2. Multiple inheritance : One child class may inherit from several parent classes when there is multiple inheritance




```
class ParentClass1:
  def method1(self):
    print("Method 1 from ParentClass1")
class ParentClass2:
  def method2(self):
    print("Method 2 from ParentClass2")
class ChildClass(ParentClass1, ParentClass2):
  def child_method(self):
    print("Child method")
child = ChildClass()
child.method1()  # Output: Method 1 from ParentClass1
child.method2()  # Output: Method 2 from ParentClass2
child.child_method() # Output: Child method
```

In this example, we have two parent classes, ParentClass1 and ParentClass2, each with their own methods. The ChildClass inherits from both ParentClass1 and ParentClass2.
The ChildClass can now access methods from both parent classes. It can call method1() from ParentClass1, method2() from ParentClass2, as well as its own child_method().
When creating an instance of ChildClass and calling the methods, you will see the respective outputs as mentioned in the comments.
Multiple inheritance allows classes to inherit and combine the behavior of multiple parent classes, providing flexibility in designing complex class hierarchies. However, it's important to carefully consider the design and potential complexities that can arise when using multiple inheritance.

Diamond problem

The diamond problem is a specific issue that can arise in programming languages that support multiple inheritance, including Python. It occurs when a class inherits from two or more classes that have a common base class. This can lead to ambiguity in method resolution, causing conflicts and making it unclear which version of a method should be used.
To mitigate the diamond problem, Python uses a method resolution order (MRO) algorithm called C3 linearization. The MRO determines the order in which the base classes are searched for a method or attribute. It follows a specific set of rules to ensure a consistent and unambiguous order.
Here's an example that demonstrates the diamond problem and how Python resolves it:

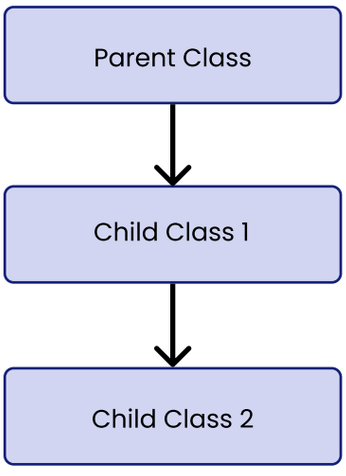
Class A:

```
  def method(self):
    print("Method in A")
class B(A):
  def method(self):
    print("Method in B")
class C(A):
  def method(self):
    print("Method in C")
class D(B, C):
  pass
d = D()
d.method() # Output: Method in B
```

Method in B

In this example, we have four classes: A, B, C, and D. Both B and C inherit from A, and D inherits from both B and C. All classes define a method called method().
When d.method() is called, Python follows the MRO to determine the order in which the base classes are searched for the method. In this case, the MRO is D -> B -> C -> A. So, Python looks for the method() in class D first, then in B, then in C, and finally in A.
As a result, the output is "Method in B". This is because B is the first class in the MRO that defines the method(), so its implementation is used.
Python's MRO algorithm effectively resolves the diamond problem by providing a well-defined order for method resolution. However, it's important to be aware of potential conflicts and understand the MRO to design your class hierarchy and override methods appropriately.

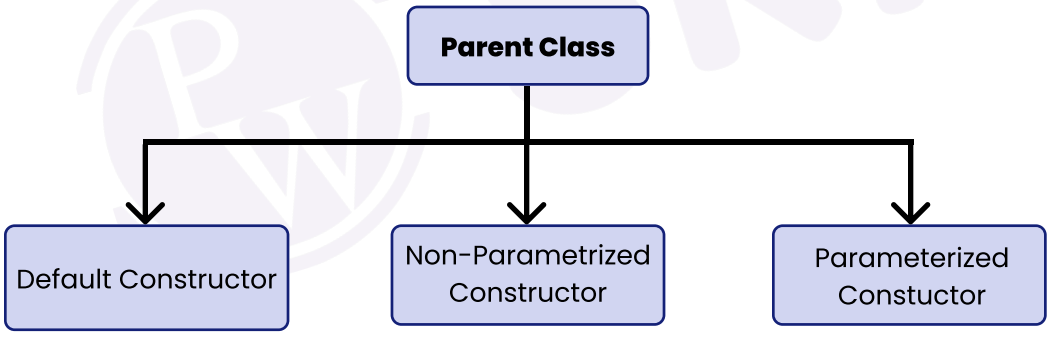
3. Multilevel Inheritance : A class inherits from a child class or derived class under multilevel inheritance. Think of three classes: A, B, and C. Superclass A, child class B, and child class C are all subclasses of A. In other words, multilevel inheritance is the term used to describe a set of classes.



Example-

```
# Base class
class Vehicle:
  def Vehicle_info(self):
    print('Inside Vehicle class')

# Child class
class Car(Vehicle):
  def car_info(self):
    print('Inside Car class')

# Child class
class SportsCar(Car):
  def sports_car_info(self):
    print('Inside SportsCar class')

# Create object of SportsCar
s_car = SportsCar()

# access Vehicle's and Car info using SportsCar object
s_car.Vehicle_info()
s_car.car_info()
s_car.sports_car_info()
```

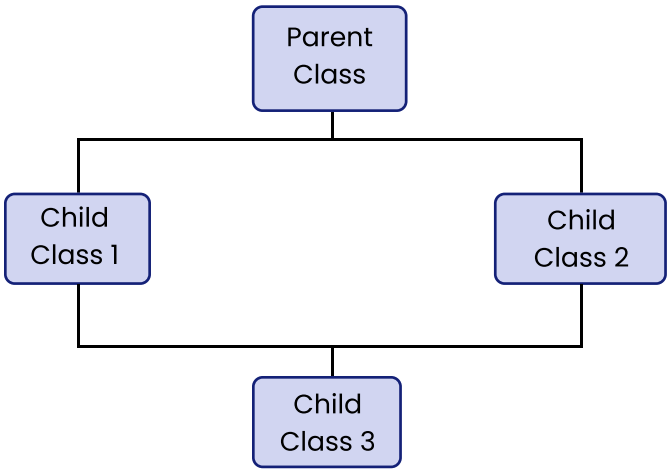
4. Hierarchical Inheritance : A single parent class gives rise to multiple child classes under hierarchical inheritance. To put it another way, we can say that there is one parent class and several child classes.



Example-

```
class Vehicle:
  def info(self):
    print("This is Vehicle")

class Car(Vehicle):
  def car_info(self, name):
    print("Car name is:", name)

class Truck(Vehicle):
  def truck_info(self, name):
    print("Truck name is:", name)

obj1 = Car()
obj1.info()
obj1.car_info('BMW')

obj2 = Truck()
obj2.info()
obj2.truck_info('Ford')
```

5. Hybrid inheritance : When inheritance consists of multiple types or a combination of different inheritance is called hybrid inheritance.




```
class Vehicle:
  def vehicle_info(self):
    print("Inside Vehicle class")

class Car(Vehicle):
  def car_info(self):
    print("Inside Car class")

class Truck(Vehicle):
  def truck_info(self):
    print("Inside Truck class")

# Sports Car can inherits properties of Vehicle and Car
class SportsCar(Car, Vehicle):
  def sports_car_info(self):
    print("Inside SportsCar class")

# create object
s_car = SportsCar()

s_car.vehicle_info()
s_car.car_info()
s_car.sports_car_info()
```

Abstraction in python : In Python, abstraction is a key idea in object-oriented programming (OOP) that allows you to describe complicated systems by hiding unneeded details and exposing just essential aspects. It aids in the management of programme complexity by breaking it down into smaller, more manageable sections.

Classes and objects are used to accomplish abstraction. A class is a blueprint for constructing objects, whereas an object is a class instance.

The class defines the properties (attributes) and behaviours (methods) of the class's objects.Abstraction allows you to construct abstract classes and methods that give a high-level interface without providing the implementation specifics.

Abstract classes cannot be instantiated and must be subclassed. They may contain abstract methods that are declared but not implemented in the abstract class itself. Subclasses are in charge of implementing these abstract methods.

Abstract Class :

To declare an Abstract class, we firstly need to import the abc module. Let us look at an example.

```
from abc import ABC
class abs_class(ABC):
    #abstract methods
```

Here, abs_class is the abstract class inside which abstract methods or any other sort of methods can be defined.
As a property, abstract classes can have any number of abstract methods coexisting with any number of other methods. For example we can see below.

```
from abc import ABC, abstractmethod
class abs_class(ABC):
    #normal method
    def method(self):
        #method definition
    @abstractmethod
    def Abs_method(self):
        #Abs_method definition
```

Here, method() is a normal method whereas Abs_method() is an abstract method implementing @abstractmethod from the abc module.

Abstraction in Python

```
from abc import ABC, abstractmethod
class Absclass(ABC):
  def print(self,x):
    print("Passed value: ", x)
  @abstractmethod
  def task(self):
    print("We are inside Absclass task")

class test_class(Absclass):
  def task(self):
    print("We are inside test_class task")

class example_class(Absclass):
  def task(self):
    print("We are inside example_class task")

#object of test_class created
test_obj = test_class()
test_obj.task()
test_obj.print(100)

#object of example_class created
example_obj = example_class()
example_obj.task()
example_obj.print(200)

print("test_obj is instance of Absclass? ", isinstance(test_obj,
Absclass))
print("example_obj is instance of Absclass? ",
isinstance(example_obj, Absclass))
```

Here,Absclass is the abstract class that inherits from the ABC class from the abc module. It contains an abstract method task() and a print() method which are visible by the user. Two other classes inheriting from this abstract class are test_class and example_class. Both of them have their own task() method (extension of the abstract method).
After the user creates objects from both the test_class and example_class classes and invoke the task() method for both of them, the hidden definitions for task() methods inside both the classes come into play. These definitions are hidden from the user. The abstract method task() from the abstract class Absclass is actually never invoked.
But when the print() method is called for both the test_obj and example_obj, the Absclass's print() method is invoked since it is not an abstract method.

PW SKILLS

THANK YOU !