"Oops" stands for Object-Oriented Programming System

In [1]:

```
class test:
```

```
  File "C:\Users\Euphor\AppData\Local\Temp\ipykernel_4976\3527747674.py",
line 1
    class test:
               ^
IndentationError: expected an indented block
```

In [6]:

```
#also
x= 1
print(type(x))
#here "x" is an object of the class integer("int")
```

```
<class 'int'>
```

In [4]:

```
#Creating a class Test
class test():
    pass

print(type(test()))
```

```
<class '__main__.test'>
```

In [5]:

```
#Object/variable/instence of a Class
a = test()
print(type(a)) #so "a" is an object of a class "test"
```

```
<class '__main__.test'>
```

In [11]:

```
class class2:

    def welcome_msg(self):
        print("welcome to pwskills")

rohan = class2()
rohan.welcome_msg()

"""here we created a class@() is created and
inside class we can define no of functions
that an object can access
"""
```

```
welcome to pwskills
```

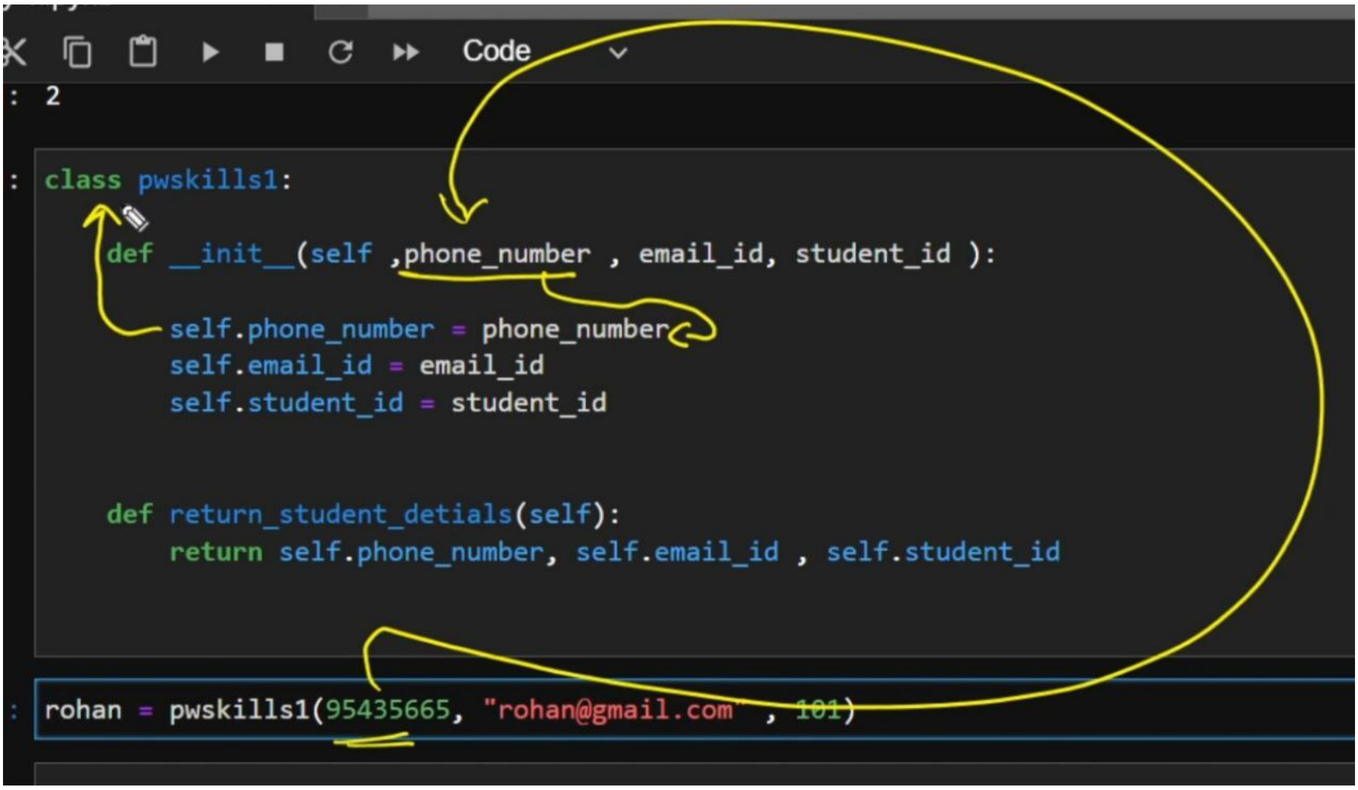
Out[11]:

```
'here we created a class@() is created and \ninside class we can define no
of functions \nthat an object can access\n'
```

In [25]:

```
#creating a class( that takes student details and print them when asked


class class3:

    def __init__(self ,phone_number , email_id, student_id ):
        """__init__ function"""
        #self is just a reference that helps class understand all the variables and funct
        self.phone_number = phone_number
        self.email_id = email_id
        self.student_id = student_id
        #usning self the class knows about the args only for perticular object


    def return_student_detials(self):
        return self.phone_number, self.email_id , self.student_id

sohan = pwskills1(999679869 , "sohan@gmail.com" , 102) #data taken from here and passed t
```

init is a **constructor**. Aconstructor will also try to take a data while creating an object and it will pass this data to the class.



```
: 2

: class pwskills1:

      def __init__(self ,phone_number , email_id, student_id ):

          self.phone_number = phone_number
          self.email_id = email_id
          self.student_id = student_id


      def return_student_detials(self):
          return self.phone_number, self.email_id , self.student_id

: rohan = pwskills1(95435665, "rohan@gmail.com" , 101)
```

In [26]:

```
sohan = class3() #class is asking for 3 arguments
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
~\AppData\Local\Temp\ipykernel_4976\3607199485.py in <module>
----> 1 sohan = class3() #class is asking for 3 arguments

TypeError: __init__() missing 3 required positional arguments: 'phone_numb
er', 'email_id', and 'student_id'
```

In [27]:

```
sohan = class3(9999019952, "sohan@gmail.com",324 ) #class is asking for 3 arguments
```

In [28]:

```
sohan.return_student_detials()
```

Out[28]:

```
(9999019952, 'sohan@gmail.com', 324)
```

In [29]:

```
sohan.phone_number
```

Out[29]:

```
9999019952
```

In [30]:

```
sohan.email_id
```

Out[30]:

```
'sohan@gmail.com'
```

In [31]:

```
sohan.student_id
```

Out[31]:

```
324
```

In [34]:

```
#self is not a keyword
#its just a convention we made of using self
#but we can use anythin inplace of it like(x/y/"string")
#example
class class4:

    def __init__(x ,phone_number , email_id, student_id ):

        x.phone_number = phone_number
        x.email_id = email_id
        x.student_id = student_id


    def return_student_detials(x):
        return sudh.phone_number, sudh.email_id , sudh.student_id
```

In [35]:

```
mohan = class4(99699579567, "mohan@gmail.com" , 234)
```

In [36]:

```
mohan.phone_number
```

Out[36]:

```
99699579567
```

In [40]:

```
class class5:

    def __init__( x,phone_number , email_id, student_id ):

        x.phone_number1 = phone_number
        x.email_id = email_id
        x.student_id = student_id


    def return_student_detials(x):
        return x.phone_number1, x.email_id , x.student_id
```

In [44]:

```
asit = class5(999735,"asit@gmail.com" , 123)
asit.phone_number
#shows that class does not understand the right side of the variable
```

```
--------------------------------------------------------------------------
-
AttributeError                            Traceback (most recent call las
t)
~\AppData\Local\Temp\ipykernel_4976\2668563002.py in <module>
      1 asit = class5(999735,"asit@gmail.com" , 123)
----> 2 asit.phone_number
      3 #shows that class only understands the left side of the variable

AttributeError: 'class5' object has no attribute 'phone_number'
```

In [49]:

```
asit.phone_number1
```

Out[49]:

```
999735
```

In [50]:

```
asit.return_student_detials()
```

Out[50]:

```
(999735, 'asit@gmail.com', 123)
```

8.2) Polymorphism

In [51]:

```
def test(a,b) :
    return a+b

#functions performing addition operation
print(test(4,5))

#performing concatination operation
print(test("sudh " , "kumar"))

#performing list append operation
print(test([2,3,4,5,5] , [4,5,6,7]))
```

```
9
sudh kumar
[2, 3, 4, 5, 5, 4, 5, 6, 7]
```

Above we can see that the function can perform different functions for different data types this concept is called **polymorphism**.

How to apply Polimorphism in classes?

In [52]:

```
#1 Creating a class1
class class1:
    def syllabus(self) :
        print("this is my method for data class1 " )

#2 creating class2
class class2 :
    def syllabus(self) :
        print("this is my method for class1" )

#3 creating an outside object
def class_parcer(class_obj) :
    for i in class_obj :
        i.syllabus()
```

In [54]:

```
#creating objects for both classes
obj_class1 = class1()
obj_class2 = class2()

#storing both these objects inside a variable as a list
class_ojb = [obj_class1 , obj_class2]
```

In [55]:

```
#now passing the above class_obj variable through class_parcer
class_parcer(class_ojb)
```

```
this is my method for data class1
this is my method for class1
```

8.3) Encapsulation

> Encapsulation is used to restrict access to methods and variables. In encapsulation, code and data are wrapped together within a single unit from being modified by accident.

> Encapsulation is an idea which allows us to prevent any kind of access or modification of a data inside the variable in opps.

> Use **"__"** double uderscore before the self variable to make it private

Type *Markdown* and LaTeX: α^2

In [59]:

```
#Class with public varibles
class test :
    def __init__(self , a,b ) :
        self.a = a
        self.b = b
```

In [65]:

```
t = test(45,67)
print(t.a)
print(t.b)
#both the variable can be seen by anyone so to stop this us double under score
```

```
45
67
```

In [97]:

```
#user manipulating the variable
t.a=30
print(t.a)
```

```
30
```

In [ ]:

```
```

The private Access Modifier

> The private member is accessible only inside class. Define a private member by prefixing the member name with two underscores, for example −

In [75]:

```
#Using encapsulation
class car:

    def __init__(self , year , make , model ,speed ) :
        self.__year = year
        self.__make = make
        self.__model = model
        self.__speed = 0

    # below is a public method for the user to modify the variables
    def set_speed(self , speed) :
        self.__speed = 0 if speed < 0 else speed

    def get_speed(self) :
        return self.__speed
```

In [76]:

```
obj_car = car(2021 , "toyota" , "innova" , 12)
```

In [80]:

```
print(obj_car._car__year)

"""can only be accesed privately inside class
by cocder i.e who knows about the variables"""
```

```
2021
```

Out[80]:

```
'can only be accesed privately inside class \nby cocder i.e who knows abou
t the variables'
```

In [81]:

```
#using public methods in above class.
obj_car.get_speed()
```

Out[81]:

```
0
```

In [82]:

```
obj_car.set_speed(456)
```

In [83]:

```
obj_car.get_speed() #speed set by user using public method
```

Out[83]:

```
456
```

In [ ]:

```

```

In [ ]:

```

```

In [102]:

```
class bank_acount:

    def __init__(self , balance ):
        self.__balance = balance  #this balance variable is hidden from user and can't be

    def deposit(self , amount ) :
        self.__balance = self.__balance + amount

    def withdraw(self , amount) :
        if self.__balance >= amount : #its a check to prevent whdthrawal more then the ba
            self.__balance = self.__balance -amount
            return True
        else :
            return False

    def get_balance(self) :
        return self.__balance

# now suppose if the balance variable is not private
# then the user can manipulate the balance variable
# and make it higher.
```

In [103]:

```
obj_bank = bank_acount(30000)
```

In [104]:

```
#getting balance without getting manipulating the variable
obj_bank.get_balance()
```

Out[104]:

```
30000
```

In [105]:

```
obj_bank.deposit(6000)
obj_bank.withdraw(5600)
```

Out[105]:

```
True
```

In [106]:

```
obj_bank.get_balance()
```

Out[106]:

```
30400
```

The protected Access Modifier

> The protected member is accessible. from inside the class and its sub-class. Define a protected member by prefixing the member name with an underscore, for example −

> _points

8.4) 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.

8.4.1) Single Inheritance

In [107]:

```
class parent:

    def test_parent(self) :
        print("this is my parent class ")

class child(parent):
    pass
```

In [109]:

```
child_obj = child()
```

In [110]:

```
child_obj.test_parent()
#So even thought the child is a different
#class it can inherit or use the methonds of parent class
```

```
this is my parent class
```

8.4.2) Multi-level Inheritance

In [111]:

```
class class1 :
    def test_class1(self) :
        print("this is my class1 " )

class class2(class1) :
    def test_class2(self) :
        print("this is my class2" )

class class3(class2) :
    def test_class3(self) :
        print("this is my class3 ")
```

In [112]:

```
#creating object of class 3
obj_class3  = class3()
```

In [114]:

```
#Accesing calss1 using class3 object
obj_class3.test_class1()
#shows that class 3 consist all the methods of class 3
```

```
this is my class1
```

In [116]:

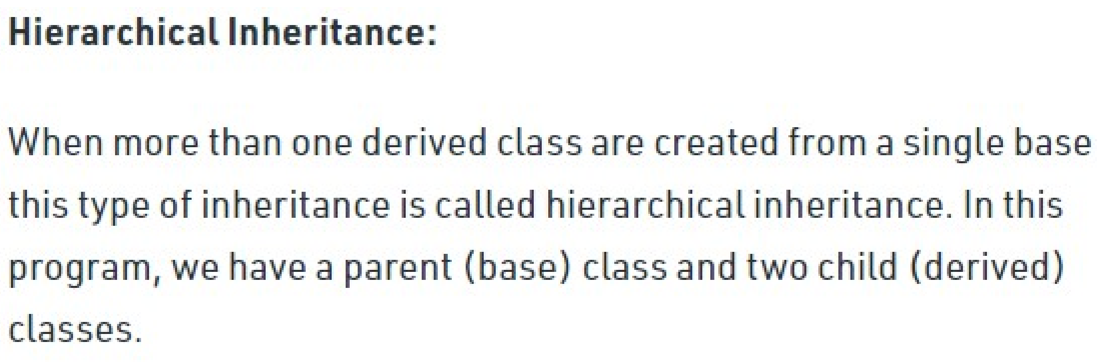
```
#Accesing calss1 using class2 object
obj_class3.test_class2()
#shows that class 3 consist all the methods of class 2
```

```
this is my class2
```

8.4.2) Multiple inheritance

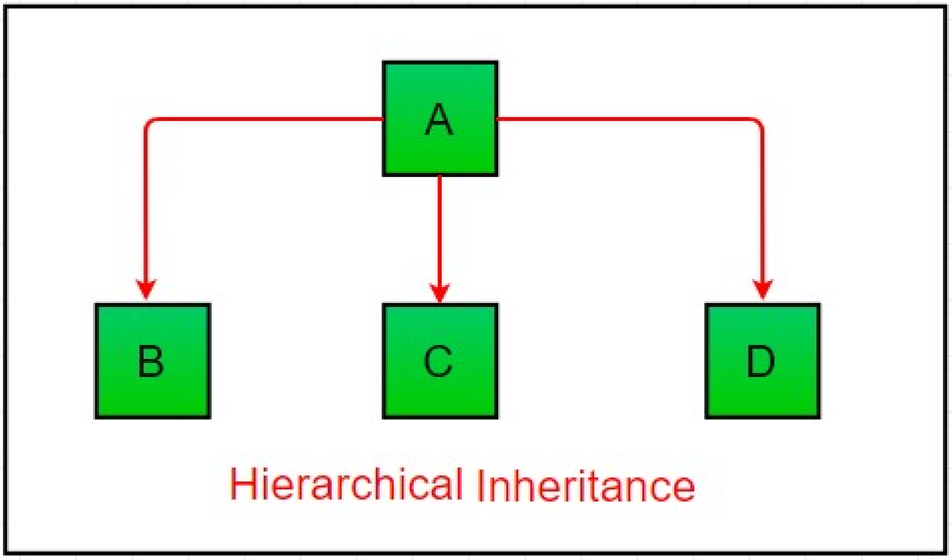
```
class class1:
    def test_class1(self) :
        print("this is my class 1" )

class class2 :
    def test_class2(self) :
        print("this is my class 2")

class class3 (class1 , class2) : #here Class 3 is performing multiple inheritance
    pass
```

In [ ]:



Hierarchical Inheritance:

When more than one derived class are created from a single base this type of inheritance is called hierarchical inheritance. In this program, we have a parent (base) class and two child (derived) classes.



In [118]:

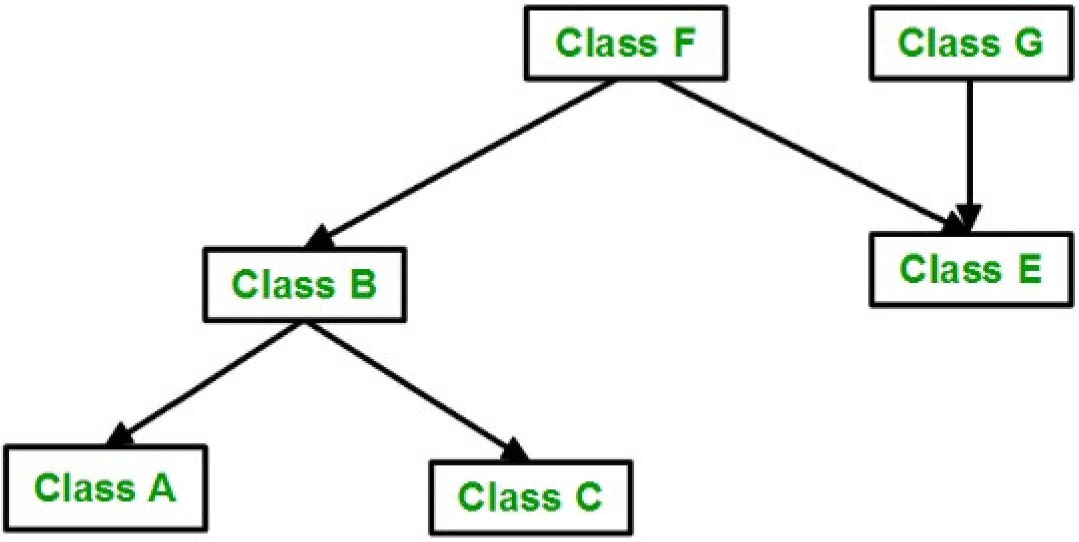
```
# Python program to demonstrate
# Hierarchical inheritance


# Base class
class Parent:
    def func1(self):
        print("This function is in parent class.")

# Derived class1


class Child1(Parent):
    def func2(self):
        print("This function is in child 1.")

# Derivied class2


class Child2(Parent):
    def func3(self):
        print("This function is in child 2.")


# Driver's code
object1 = Child1()
object2 = Child2()
object1.func1()
object1.func2()
object2.func1()
object2.func3()
```

```
This function is in parent class.
This function is in child 1.
This function is in parent class.
This function is in child 2.
```

In [ ]:

Hybrid Inheritance:

Inheritance consisting of multiple types of inheritance is called hybrid inheritance.

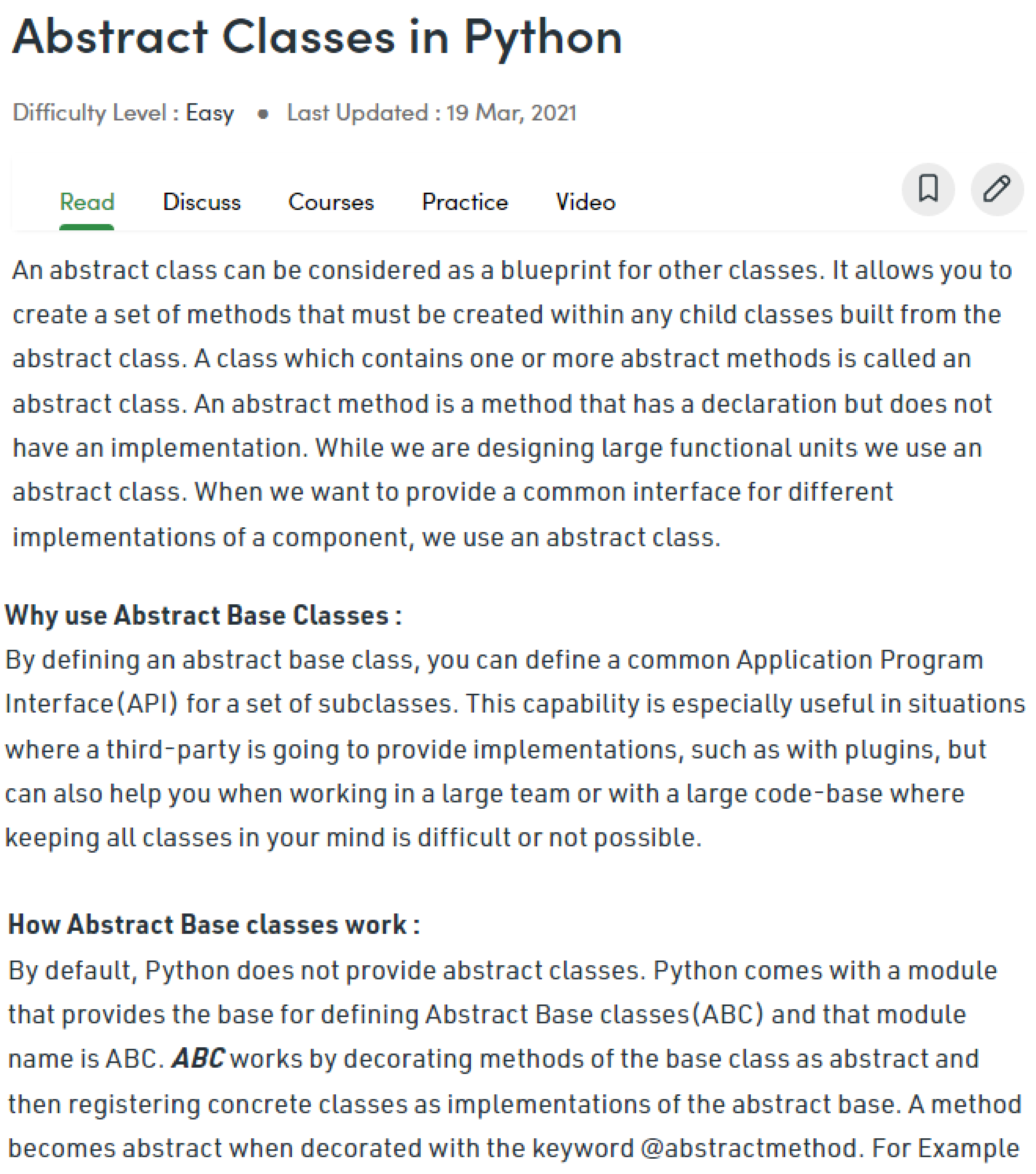


In [119]:

```
# Python program to demonstrate
# hybrid inheritance


class School:
    def func1(self):
        print("This function is in school.")


class Student1(School):
    def func2(self):
        print("This function is in student 1. ")


class Student2(School):
    def func3(self):
        print("This function is in student 2.")


class Student3(Student1, School):
    def func4(self):
        print("This function is in student 3.")


# Driver's code
object = Student3()
object.func1()
object.func2()
```

```
This function is in school.
This function is in student 1.
```

8.5) Abstract Class



Abstract Classes in Python

Difficulty Level : Easy • Last Updated : 19 Mar, 2021

An abstract class can be considered as a blueprint for other classes. It allows you to create a set of methods that must be created within any child classes built from the abstract class. A class which contains one or more abstract methods is called an abstract class. An abstract method is a method that has a declaration but does not have an implementation. While we are designing large functional units we use an abstract class. When we want to provide a common interface for different implementations of a component, we use an abstract class.

Why use Abstract Base Classes :
By defining an abstract base class, you can define a common Application Program Interface(API) for a set of subclasses. This capability is especially useful in situations where a third-party is going to provide implementations, such as with plugins, but can also help you when working in a large team or with a large code-base where keeping all classes in your mind is difficult or not possible.

How Abstract Base classes work :
By default, Python does not provide abstract classes. Python comes with a module that provides the base for defining Abstract Base classes(ABC) and that module name is ABC. ***ABC*** works by decorating methods of the base class as abstract and then registering concrete classes as implementations of the abstract base. A method becomes abstract when decorated with the keyword @abstractmethod. For Example

In [1]:

```
import abc

class pwskills :

    @abc.abstractmethod
    def student_details(self):   #method
        pass

    @abc.abstractmethod
    def student_assignment(self):   #method
        pass


    @abc.abstractmethod
    def student_marks(self):    #method
        pass

#This is a blueprint Class that can be added as an argument in another class so that
#Whatever common methods can be kept inside this Abstract class
```

In [2]:

```
class data_science(pwskills):  #Chlid Class #passing  above class as a blueprint

    def student_details(self):  #you can already access this class from pwskills
        return "it will try to return a details of data science masters "

    def student_assignment(self):
        return "it will return a details of student assignemnt for data science masters "


#as
```

In [6]:

```
class web_dev(pwskills):  #Child Classs
     def student_details(self):
            return "this will retrun a detils of web dev "


     def student_marks(self):
             return "this will return a makrs of web dev class"
```

In [7]:

```
ds = data_science()
ds.student_details()
```

Out[7]:

```
'it will try to return a details of data science masters '
```

In [8]:

```
wb = web_dev()
wb.student_details()
```

Out[8]:

```
'this will retrun a detils of web dev '
```

So

Abstract Class is like a Blueprint for other classes. That allows us to create a set of methohds that must be created inside the child class

In [ ]:

```

```

In [ ]:

```

```