OOP in Python

To work with real scenarios, we started using objects in code.
This is called object oriented programing.

# Creating class
class Student:
    name = "Idrees"

# Creating object (instance of class)
S1 = Student()
Print(S1.name)

__init__ Function (executed when) object creation
Constructor

All classes have a function __init__(), which is always executed when object is being initiated.

# Creating class
class Student:
    def __init__(self, fullname)
        self.name = fullname

argument / parameter

attribute

# Creating object
S1 = Student("Isha")
Print(S1.name)

object

* The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.

There are two type of constructors

1.) Default constructor => In this there are only one parameter that is self. If we don't need to always make this because if we do not made it compiler automatically made. there are some special cases or we can say such as real life scenarios where we made it.

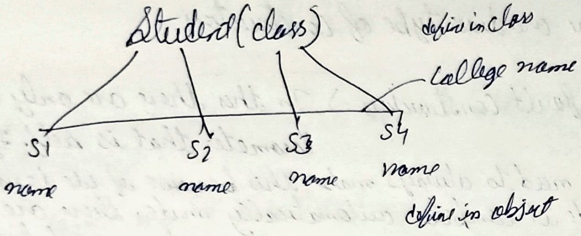
2) Parameterized constructor : It contain more than one parameter.

=> Class & instance Attributes -> data, variable, name

Class.attr — same for all
obj.attr — different for all

instance (obj) attribute : If we creat a Student (class) and add student name S1, S2, S3, S4. For this all the student name is different for in object. This called instance attr. to add this we write self.name in class.

class : For all student college name is common. So we don't need to define many time. We only one a define in class only.



Memory

def Student():

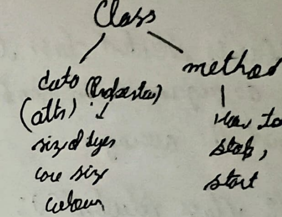
```
    college_name = ABCD     # class attr.

    def __init__(Self, name, marks)
        Self.name = name              -> obj attr:
        Self.marks = marks               To store different value we use
        Print("Adding new Student in Database")

S1 = Student("Karan", 97)
Print(S1.name, S1.marks)   # S1. used to access object name
Print(Student.college_name)  # to access class use student
     ( S1. college_name)
```

Note: If the object values and class values are same then the precedence of object > class

Methods:-



Methods are function that belongs to objects.

```
=> class Student:
       college_name = "ABCD"

       def __init__(Self, name, marks)
           Self.name = name
           Self.marks = marks

       def welcome(Self):                # using method
           Print("Welcome Student", S1.name)

       def get_marks(Self):
           ~~Print~~ return S1.marks

   S1 = Student("Karan", 98)
   ~~Print(S1.name~~
   S1.welcome
   Print(S1.getmarks())
```

Practice Problems

1) Create Student class that take name & marks of 3 subject as argument in constructor. Then create a method to print the average.

Sol:

```
Class Student():
    def __init__(Self, name, marks):
        Self.name = name
        Self.marks = marks
    def avg_marks(Self)
        Sum = 0
        For val in Self.marks:
            Sum += val
        Print("hi", Self.name, "Your avg score is", Sum/3)

S1 = Student("Varun", [98, 99, 97])      # list
S1.avg_marks()

S1.name = "Bhuvan"
S1.avg_marks()      # We can also replace the name.
```

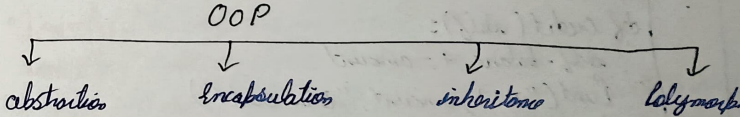
Static Methods

Methods that don't use the Self parameter (work at class level)

```
=> Class Student():
    @staticmethod      # decorator
    def college():
        Print("ABC College")
```

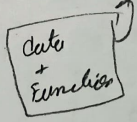
* Decorators allow us to wrap another function in order to extend the behaviour of the wrapped function, without permanently modifying it.

Important (Pillars of OOPs)



Abstraction : Hiding the implementation details of a class and only showing the essential features of the use

Encapsulation : Wrapping data and function into a single uni (object)



Pratice

1) Create Account class with 2 attributes - balance & accountno.
Create methods for debit, credit & printing the balance

Sol:

```
class Account():
    def __init__(self, balance, acc):
        self.balance = bal
        self.accountno = acc
    def debit(self):
        self.balance -= amount
        print("Rs.", amount, "was debited")
        print("Total balance", self.get_balance())

    def credit(self):
        self.balance += amount
        print("Rs.", amount, "was credit")
        print("Total balance", self.get_balance())

    def get_balance(self):
        return self.balance

acc1 = Account(1000, 123456)
acc1.debit(1000)
acc1.credit(500)
```

- Del Keyword

Used to delete object properties or object itself.

```
del S1.name
del S1
```

```
=> class Student:
       def __init__(self, name):
           self.name = name
S1 = Student("Shivam")
print(S1.name)        ---> #output = Shivam
del S1.name
print(S1.name)        ---> #output => Error
```

Public

- Private (like) attributes & methods :-

Conceptual implementation in Python

Private attributes & methods are meant to be used by only within the class and are not accessible from outside the class

Public, Private

To private something just use __ in front of attribute or method.

```
=> class Person:
       # def __init__(self):
           name = "Anonymous"

       def __hello(self):
           print("Hello Person")
       def welcom(self):
           self.__hello()
```

P1 = Person()
Print (P1.welcome())

3) Inheritance

When one class (child/derived) derives the properties & (attr/meth) methods of another class (Parent/base).

```
Class Car:
  ---   -> Start()
  ---   -> Stop()
class toyotacar(car):
   ---
```

```
=> Class Car:
     Colour = "Black"
     @staticmethod
     def start():
        Print("Start car")
     def stop():
        Print("Stop car)
   Class toyotacar(car):
     def __init__(self, name):
        self.name = name
   S1 = toyotacar("fortuner")
   Print(S1.name)
   (S1.start())
```

There are three type of inheritance

- Single inheritance -> Base (Parent) ——> Derived
- Multi-level inheritance -> Base (Par) ——> Derived (Pa) ——> Cu...
- Multiple inheritance

```
Class A:
   VarA = "Welcome to class A"
Class B:
   VarB = "Welcome to class B"
Class C(A,B):
   VarC => "Welcome to class C"
C1 = C()
Print(C1.VarC)
Print(C1.VarB)
Print(C1.VarA)
```

- Super method

Super() method is used to access methods of the parents class.

```
Super().__init__(type)
Super().start()
```

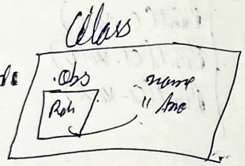
Class Method

A class method is bound to the class & receives the class as an implicit first argument

Note: Static method can't access or modify class & generally for utility.

```
Class Student:
    @classmethod   # decorator
    def college(cls):
        Pass
```

=>
```
Class Person:
    name = "anonyms"          ==> @classmethod
    def Changename(Self cls, name):
        cls Self. name = name     # ① Person.name = name
                                  # __Class__. name     ] two way to change name
P1 = Person()
P1. Changename("Rahul Kumar")
Print(P1. name)       # output => Rahul Kumar
Print(Person. name)   # output => Anonymous
```



① Static Method

② Class method (cls)

③ Instance methods (self)

Property

We use @property decorator on any method in the class to use the method as a property.

=>
```
Class Student:
    def __init__(Self, Phy, chem, math):
        Self. Phy = Phy
        Self. chem = chem
        Self. math = math

    @Property
    def Percent(Self):
        return str((Self. Phy + Self. chem + Self. math) / 3) + "%"

Std1 = Student(98, 99, 99)
Print(Std1. Percent)     # output - 98.66%
Std1. Math = 100
Print (Std1. Percent)    # output => 99.33%
```

-> Why we are using @property decorator?

Ans: Because if we change marks after declared once. ~~at update in class~~. and after this we update marks then the output is also update.

#4) Polymorphism : Operator overloading
many forms

When the same operator is allowed to have different meaning according to the context.

Operation & Dunder function

a + b	# addition	a.__add__(b)
a - b	# substraction	a.__sub__(b)
a * b	# multiplication	a.__mul__(b)
a / b	# division	a.__truediv__(b)
a % b	# addition	a.__mod__(b)

Complex number

2i + 3j
3i + 4j
5i + 7j

Real no
1 + 2
-> 3

In Python complex number को create करने का कोई method नहीं है

So we use own class to add complex number

```
class Complex:
    def __init__(self, real, img):
        self.real = real
        self.img = img

    def shownumber(self):
        print(self.real, "i +", self.img, "j")

    def __add__(self, num2):
        newreal = self.real + num2.real
        newimg = self.img + num2.img
        return Complex(newreal, newimg)

num1 = Complex(1, 3)
num1.shownumber()

num2 = Complex(4, 3)
num2.shownumber()

num3 = num1 + num2
num3.shownumber
```

def __init__(self) - default constructor
def __init__(self, name) - parameterized constructor
argument / parameter

OOPS

Class
Blueprint of an object

Method (Functions) and Attributes (variables)

- Class attributes
- object attributes

object
Real world entity

4 Pillars of oops
1. Inheritance
   - Child class inherits from parents class
2. Abstract
   - Hiding complex logic
   - Show only essentials
3. Encapsulation
   - Binding data and methods
   - Use private / public
4. Polymorphism
   - Many Forms (Same method, different behaviours)
   - Example: 'Speak()' - Dog barks, cat meows

Methods
@staticmethod
@classmethod (cls)
@property (self)