OBJECT ORIENTED PROGRAMMING

CREATING CLASSES:-

```
class Dog(objects):
    def __init__(self):
        print('Nice You Have a Dog')

    def speak(self):
        pass
```

⇒ It is not compulsory to put ( ) ⇒ objects in parenthesis.

⇒ We have to define it by giving double underscore & putting self means the constructor will do it.

```
HM = dog()
AM = dog()
```

⇒ HM is known as instant of type dog or class Dog

```
>> Nice You Have a Dog
   Nice You Have a Dog
```

Lets add Attribute to it.

```
class Dog(object):
    def __init__(self, name):
        self.name = name

    def speak(self):
        print("Hi I am", self.name)

HM = Dog('HM')
AM = Dog('AM')

HM.speak()
AM.speak()
```

```
>> Hi I am HM
   Hi I am AM
```

We can create how much method we want to create.

But they has to be self.

* INHERITANCE :-

```
* class Dog (object):
        def __init__(self, name, age):
            self.name = name
            self.age = age
        def speak(self):
            print("Hi I am", self.name, "my age is", self.age, "years old")

class Cat (Dog):
```

now if we create a new class & we want some property of the old class, then we just have to put the class name in parenthesis. And everything will be sorted out.

```
        def __init__(self, name, age, color):
            super().__init__(name, age)
            self.color = color

tim = Cat('tim', 5, 'Blue')
tim.speak()
```

>> Hi I am tim my age is 5 years old.

* How to change something from Dog class that we don't want in Cat Class.

* class Dog(object):

[Same as previous]

```
def talk(self):
    print('Bark')
```

But we don't want "Bark" in Cat class then, what we do is.

```
class Cat(Dog):
```

[Same previous]

```
def talk(self):
    print('Meow')

tim = Cat('Tim', 5, 'Blue')
tim.talk()
```

So, anything we will overwrite in Cat class will some be print. Not from the dog. class.

So, it will overwrite that method.

⇒ Inheritance in simple means putting the attributes of different class.

More of Inheritance

OVERLOADING METHODS :-

```
class Point():
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
        self.coords = (self.x, self.y)

    def move(self, x, y):
        self.x += x
        self.y += y
    def __add__(self, p):
        return Point(self.x + p.x, self.y + p.y)
    def __sub__(self, p):
        return Point(self.x - p.x, self.y - p.y)
    def __mul__(self, p):
        return self.x * p.x + self.y * p.y

    def __str__(self):
        return "(" + str(self.x) + ',' + str(self.y) + ")"

p1 = Point(3,4)
p2 = point(3,2)
p3 = point(1,3)
p4 = point(0,1)

p5 = p1 + p2
p6 = p4 - p1
p7 = p3 * p4
```

```
print(p5, p6, p7)
```

```
>>
(6,6) (-3,-3) 4
```

Comparing Methodes

* STATICS METHODS AND CLASS METHODS :—

```
class Dog:
    def __init__(self, name):
        self.name = name
        self.dog.append(self)

    @classmethod
    def num_dogs(cls):
        return len(cls.dogs)

    @static method
    def bark(n):
        """ barks n times"""
        for _ in range(n):
            print('bark')

Dog.bark(5)
```

these are known as decorators

```
>>
Bark
Bark
Bark
Bark
Bark
```

* PRIVATE AND PUBLIC CLASS:-

If we have anything like "_" before that or dot underscore before, then it means typically they are PRIVATE.

Eg:- _Private:

Now to run this we have created 2 pages one with mod.py & another with tutorial1.py. [Your choice create what name you want to create.]

mod.py

```
class _Private:
    def __init__(self, name):
        self.name = name


class NotPrivate:
    def __init__(self, name):
        self.name = name
        self.priv = _Private(name)

    def _display(self):
        print("Hello")

    def display(self):
        print("Hi")
```

tutorial 1.py

```
import mod
from mod import NotPrivate

test = NotPrivate('Hm')
test.__display()
```

>> Hello

[So, it means we can actually
Show private method as well.
Just the thing is they are private

if we put at the end line

```
test.display()
```

>> Hi