Inner Classes or Nested Classes:
We can declare a class inside another class, such type of classes are called inner classes. Without existing one type of object, if there is no chance of existing another type of object,then we should go for inner classes.

Example:
Without existing Fan object there is no chance of existing Rotator object. Hence Rotator class should be part of Fan class.

```
class Fan():
.........................
.........................
   class Rotator():
   .........................
   .........................
```

Note:
Without existing outer class object there is no chance of existing inner class object.

EXAMPLE:
```
class Outer_Class():
    def __init__(self):
        print("Hey Outer Class")
    class Inner_Class():
        def __init__(self):
            print("Bye Inner Class")
        def Inner_Method(self):
            print("InnerClassMethod")
OO=Outer_Class()
II=OO.Inner_Class()
II.Inner_Method()
```

Class Method in PYTHON:
It returns a class method for the given function. It is a method that is bound to a class rather than its object. It doesn't require creation of a class instance. We can declare class method explicitly by using @classmethod decorator. For class method we should provide 'cls' variable at the time of declaration. A class method receives the class as implicit first argument.

Syntax:
```
@classmethod
def func(cls, args...)
```

Example:
```
class Banker():
    Cust=10000
    @classmethod
    def service(cls,name):
        print('{} has {} Customers...'.format(name,cls.Cust))
#Calling with class name
Banker.service('SBI')
Banker.service('ICICI')
```

When to use?
Class method to create factory methods. Factory methods return class object (Similar to a constructor)

Static Method in PYTHON:
1 Simple functions with no 'self' argument
2 Nested inside a class
3 These methods are general utility methods.
4 Inside these methods we won't use any instance or class variables.
5 We can declare static method explicitly by using @staticmethod decorator.
6 We can access static methods by using classname or object reference

Example:
```
class Nit():
  @staticmethod
  def static_method(attr):
    print(attr)
#Calling static Method
Nit.static_method("Hello Static Method")
```

Example:
```
class Math_Compute():
    @staticmethod
    def Sum(A,B):
        print('The Sum is:',A+B)
    @staticmethod
    def Mult(A,B):
        print('The Product is:',A*B)
    @staticmethod
    def Avg(A,B):
        print('The average:',(A+B)/2)

Math_Compute.Sum(1,2)
Math_Compute.Mult(3,4)
Math_Compute.Avg(5,6)
```

Class method vs Static Method
A class method takes cls as first parameter while a static method needs no specific parameters.
We use @classmethod decorator in python to create a class method.
We use @staticmethod decorator to create a static method

When to use?
Static methods to create utility functions.

Python Metaprogramming
It is the concept of building functions and classes whose primary target is to manipulate code by modifying, wrapping or generating existing code. The major features of meta-programming are:
I. Metaclasses
II. Decorators
III. Class-decorators

Metaclasses
In Python everything have some type associated with it. You can get type of anything using type() function. Metaclass create Classes and

Classes creates objects.

Metaclass-Hierarchy.jpg..!

Example:

```
num = 23
print("Type of num is:", type(num)) #<class 'int'>

lst = [1, 2, 4]
print("Type of lst is:", type(lst)) #<class 'list'>

name = "PYTHON"
print("Type of name is:", type(name)) #<class 'str'>
```

Every type in Python is defined by Class.
Example:

```
class Student():
        pass
Stu_Obj = Student()
print("Type of stu_obj is:", type(Stu_Obj)) #<class
'__main__.Student'>
```

A Class is also an object, and just like any other object it’s a instance of something called Metaclass. A special class type creates these Class object.

Class-decorators
Decorators are a very powerful and useful tool in Python since it allows programmers to modify the behaviour of function or class.

Example:

```
class MyDecorator:
        def __init__(self, function):
                self.function = function
        def __call__(self):
                self.function()
@MyDecorator
def function():
        print("Welcome to Class Decorators")
function()
```

Example:

```
def Decorator_One(f):
    def Method():
        print("Decorating", f.__name__)
        f()
    return Method
@Decorator_One
def Decorator_Method():
    print("inside Decorator_Method()")
Decorator_Method()
```

Python property decorator--@property
It is one of the built-in decorators. The main purpose of any decorator is to change your class methods or attributes in such a way so that the user of your class no need to make any change in their code.

Property Decorators - Getters, Setters and Deleters

Setter Method:
It can be used to set values to the instance variables. setter methods also known as mutator methods.

Syntax:
```
def setVariable(self,variable):
      self.variable=variable
```

Example:
```
def setName(self,name):
         self.name=name
```

Getter Method:
It can be used to get values of the instance variables. Getter methods also known as accessor methods.

Syntax:
```
def getVariable(self):
     return self.variable
```

Example:
```
def getName(self):
      return self.name
```

Example:
```
class Student:
    def setName(self,name):
        self.name=name
    def getName(self):
        return self.name
n=int(input('Enter number of students:'))
for i in range(n):
     s=Student()
     name=input('Enter Name:')
     s.setName(name)
print('Hi',s.getName())
```

Example:
```
class Student:
    def setName(self,name):
        self.name=name
    def DelName(self):
        return self.name
n=int(input('Enter number of students:'))
for i in range(n):
     s=Student()
     name=input('Enter Name:')
     s.setName(name)
del name
```

WHY INHERITANCE(NEED OF REUSABILITY)
Example:
```
class Banking():
    def Login(self):
        print("LoginRequiredForBanking")
        print("Thank U")
class OnlineBanking():
    def Login(self):
        print("LoginRequiredForBanking")
        print("Thank U")
    def Online(self):
        print("Welcome to Online Banking")
        print("Secured Banking")
class MobileBanking():
    def Login(self):
        print("LoginRequiredForBanking")
        print("Thank U")
    def MBanking(self):
        print("Welcome to Touch Banking")
        print("It is More Comfort and AnyWhere Banking")
class PhoneBanking():
    def Login(self):
        print("LoginRequiredForBanking")
        print("Thank U")
    def PBanking(self):
        print("Welcome to Phone Banking")
        print("It is Only for Senior Citizens")
        print("Thank U")

#Creating Instance or Object
PP=PhoneBanking()
PP.Login()
PP.PBanking()
MM=MobileBanking()
MM.Login()
MM.MBanking()
OO=OnlineBanking()
OO.Login()
OO.Online()
```

Example: UseCase-II

```
class Animal():
    def Eat(self):
        print("All Animals are Eating")
class Dog():
    def Eat(self):
        print("All Animals are Eating")
    def Sound(self):
        print("Barking........!!")
class Cat():
    def Eat(self):
        print("All Animals are Eating")
    def Sound(self):
        print("Mewwwwwwwwwww!!!")
class Lion():
    def Eat(self):
        print("All Animals are Eating")
    def Sound(self):
        print("Roaring...........!!")
LL=Lion()
LL.Eat()
LL.Sound()
```

```
CC=Cat()
CC.Eat()
CC.Sound()
DD=Dog()
DD.Eat()
DD.Sound()
```

INHERITANCE:
Reuse-Class Members-Attributes & Methods
OR
It is the process of inheriting the class members from one class to another class is called Inheritance.
OR
The concept of using properties of one class into another class without creating object of that class explicitly is known as inheritance.
OR
1 A class which is extended by another class is known as 'super class'.
2 Both super class property and sub class property can be accessed through sublcass ref.variable

Types of Inheritance(s):
1. Single Level Inheritance
2. Multilevel Inheritance
3. Multiple Inheritance
4. Hierarchial Inheritance
5. Hybrid Inheritance
6. Diamond Inheritance

Outline of Inheritance:
```
BaseClass #Existing Class
Feature1
Feature2

DerievedClass #NewlyCreatedClass
Feature1
Feature2
Feature3
```

Terminology:
Parent class ==> Base class ==>Super class
Child class ==> Derived class==> Sub class

Single Level Inheritance:
The concept of inheriting properties from only one class into another class is known as single level inheritance.

Syntax:
```
class name(superclass):
   BlockOfStatements
```

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

```
Example:
class animal:
    def eat(self):
      print('Eating...')

class dog(animal):
    def bark(self):
        print("barking")

d=dog()
d.eat()
d.bark()

Example:
class Animal():
    def Eat(self):
        print("Animal Eating...!!")
class Dog():
    def Sound(self):
        print("Dog Barking..!!")

D=Dog()
D.Eat()
D.Sound()

O/P
AttributeError: 'Dog' object has no attribute 'Eat'

Example:
class Banker():
    def Services(self):
        print("Every Bank Provide Services to Customers")
class Customer(Banker):
    def Cust_Ser(self):
        print("Customer using all services from Banker")
CC=Customer()
CC.Services()
CC.Cust_Ser()

Example:
class animal:
    def __init__(self,name):
        self.name=name
class dog(animal):
    def display(self):
        print(self.name)
d=dog("Puppy")
d.display()
```

RealTime Usecases for Single Level Inheritance:

1. Student ==> Subject
2. Subject ==> Result
3. Bank ==> Customer
4. Bank ==> Services
5. Product ==> feedback