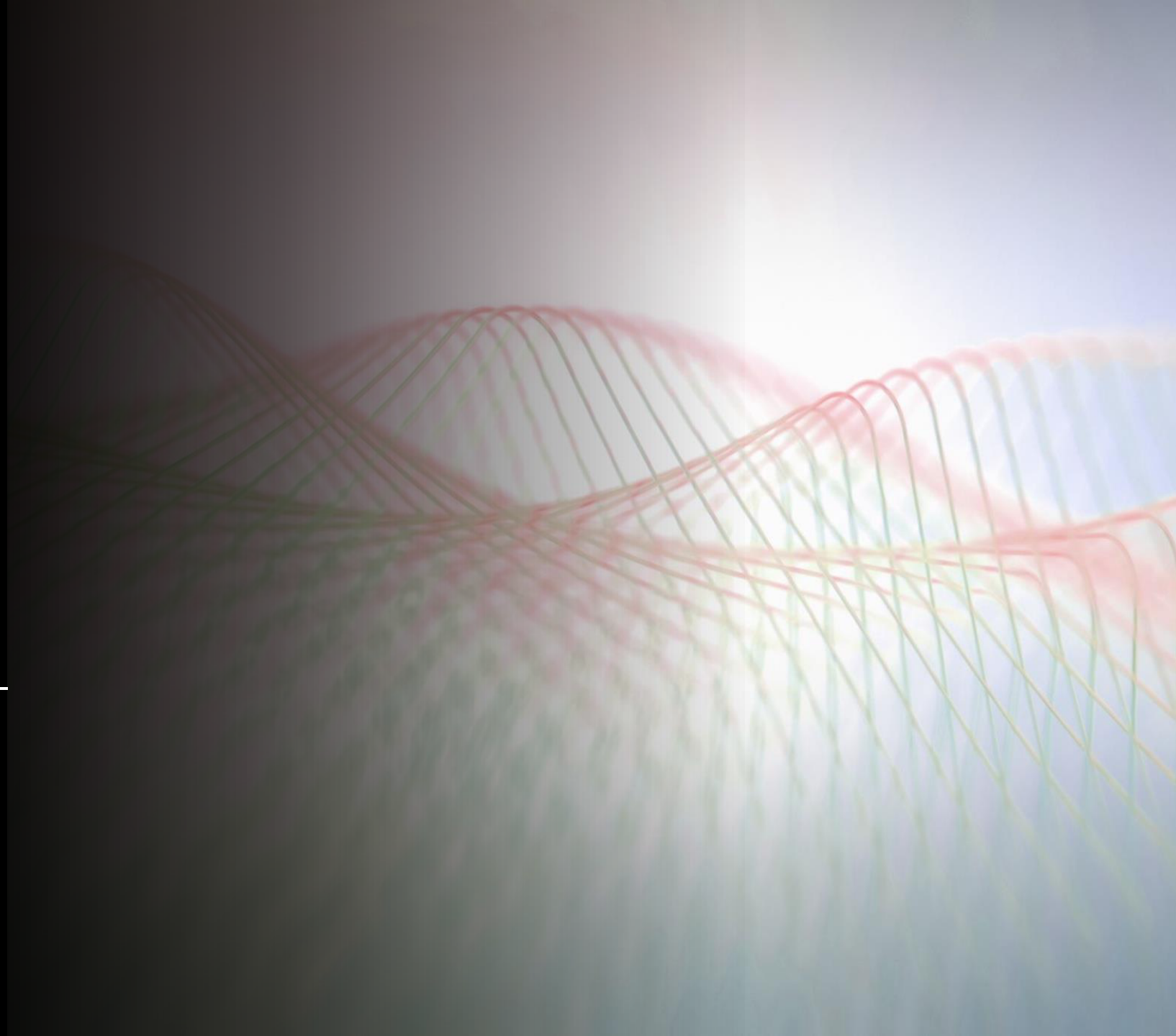
Data Structure with Python

Mr. V.M.Vasava

GPG,IT Dept.

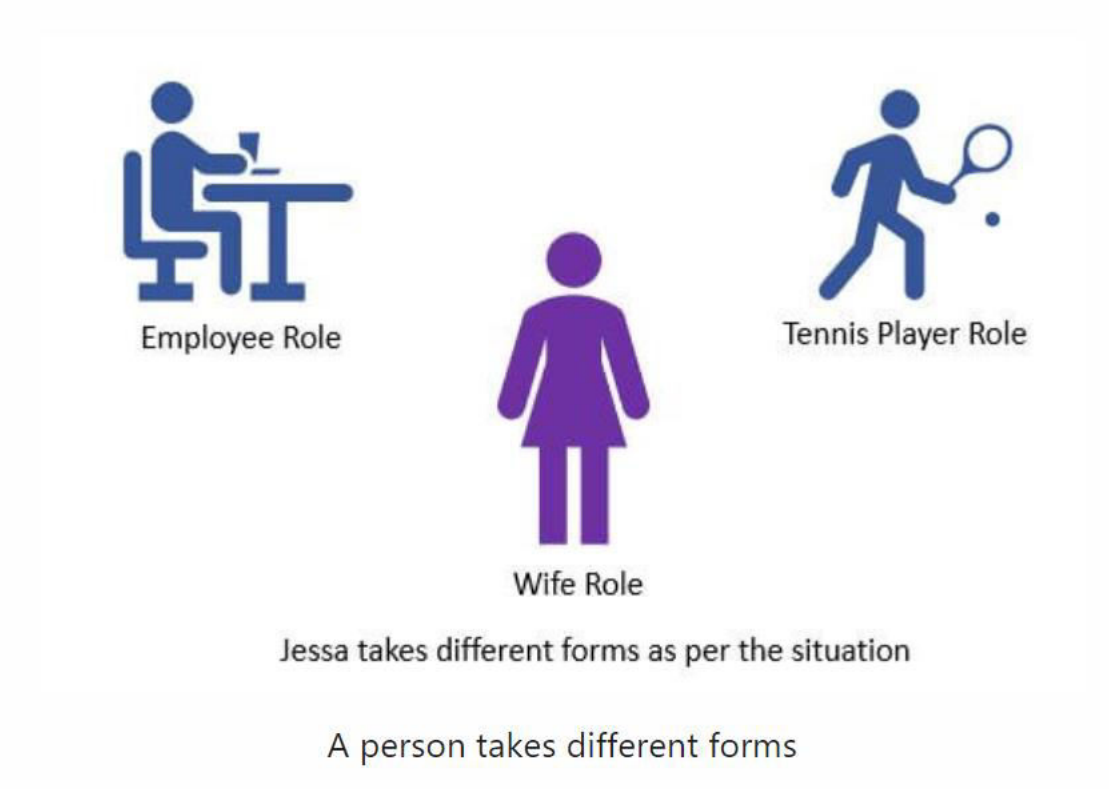


Agenda

Polymorphism

Example

Polymorphism



- Polymorphism

Polymorphism in Python is the ability of an object to take many forms.

In simple words, polymorphism allows us to perform the same action in many different ways.

Polymorphism

- Polymorphism means same function name being uses for different types.
- Polymorphism helps us in performing many different operations using a single entity.
- A basic example of polymorphism is a **‘+’** operator.
- We know we can add as well as concatenate numbers and string respectively. With the help of **‘+’** operator.
- **Example of Polymorphism in Python**

1. Polymorphism in ‘+’operators
2. Polymorphism in ‘*’operators
3. Polymorphism in Functions
4. Polymorphism in Classes

Example : **Polymorphism in ‘+’operators**

```
In [2]:
    ...: var1 = 1
    ...: var2 = 2
    ...: print("Addition of number :",var1+var2)
    ...: # Concatenation of string using '+' operator
    ...: str1 = 'Hello'
    ...: str2 = ' World'
    ...: print("Concatenation of string :", str1 + str2)
Addition of number : 3
Concatenation of string : Hello World
```

Example : **Polymorphism in '*'operators**

```
    ...: var1 = 1
    ...: var2 = 2
    ...: print("multiplication of number :",var1*var2)
    ...: # Multiplication of string using '*' operator
    ...: var = 2
    ...: str2 = 'World '
    ...: print("Multiplication of string :", var * str2)
multiplication of number : 2
Multiplication of string : World World
```

Example: **Polymorphism in functions**

```
In [4]:
   ...:
   ...: str = 'Hello'
   ...: print("Length of String: ",len(str))
   ...: # Length of dictionary using len()
   ...: MyDict = {'Name': 'Nivedita', 'Age': 5, 'Class': 1}
   ...: print("Length of Dictionary: ",len(MyDict))
Length of String:  5
Length of Dictionary:  3
```

Polymorphism in Classes

- polymorphism with classes methods having the same name.

```
class Employee:
    def info(self):    #incstance method
        name = "VMV"
        dept = "IT"
        print(name + " from "+dept)
class Admin:
    def info(self): #instance method
        name = "PAP"
        dept = "IT"
        print(name + " from "+dept)
obj_emp = Employee()
obj_adm = Admin()
obj_emp.info()
obj_adm.info()
```

```
VMV from IT
PAP from IT

In [7]:
```

Method Overriding

- When a child class method overrides the parent class method of the same name, parameters and return type, it is known as method overriding.
- Method Overriding is a part of the inheritance mechanism
- Method Overriding avoids duplication of code
- Method Overriding also enhances the code adding some additional properties.

```
In [7]:
   ...: class Add:
   ...:     def result(self,a,b):
   ...:         print("Addition",a+b)
   ...: class Multi(Add):
   ...:     def result(self,a,b):
   ...:         print("Multiplication",a*b)
   ...: m =Multi()
   ...: m.result(10,20)
Multiplication 200
```

Polymorphism though inheritance

```
...: class Add:
...:     def result(self,x,y):
...:         print("Addition",x+y)
...: class Multi(Add):
...:     def result(self,a,b):
...:         super().result(10, 20)
...:         print("Multiplication",a*b)
...: m =Multi()
...: m.result(10,20)
Addition 30
Multiplication 200
```

Advantages of Inheritance

- it provides Code reusability, readability, and scalability.
- It reduces code repetition. You can place all the standard methods and attributes in the parent class. These are accessible by the child derived from it.
- By dividing the code into **multiple classes**, the applications look better, and the error identification is easy.

Any Questions ???

OOP with Python

Mr. V. M. Vasava

GPG,IT Dept.

Agenda

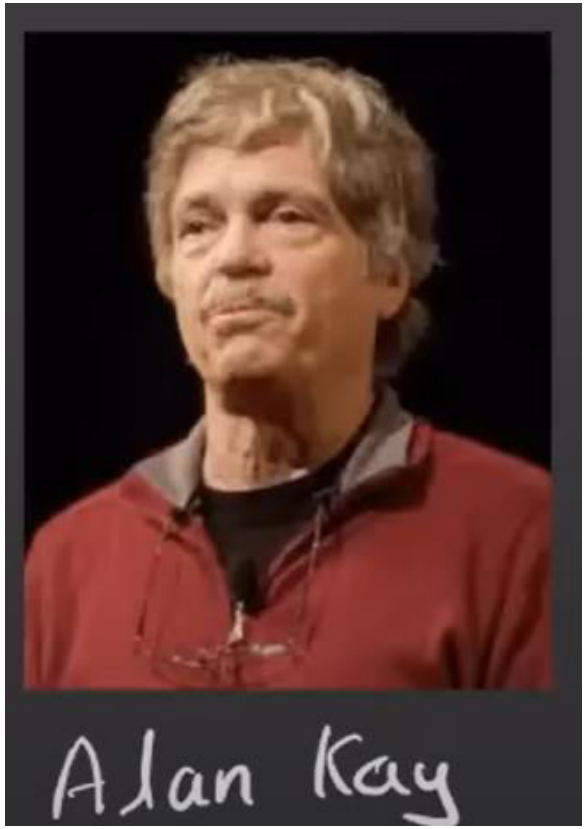
INTRODUCTION TO OOP

CLASS

OBJECT

History of OOP

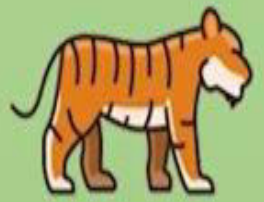
- Object Oriented Programming (OOP) was coined by Alan Kay in 1967.
- Alan Kay with Few others developed SmallTalk.
- The First programming language widely recognized as "Object Oriented" was **SIMULA in 1965.**
- **Simula introduces concepts of class, object, inheritance etc..**
- **C** Language –1972 Dennis Ritchi
- C++ -1979 Bjarne Stroustrup



Introduction about OOP

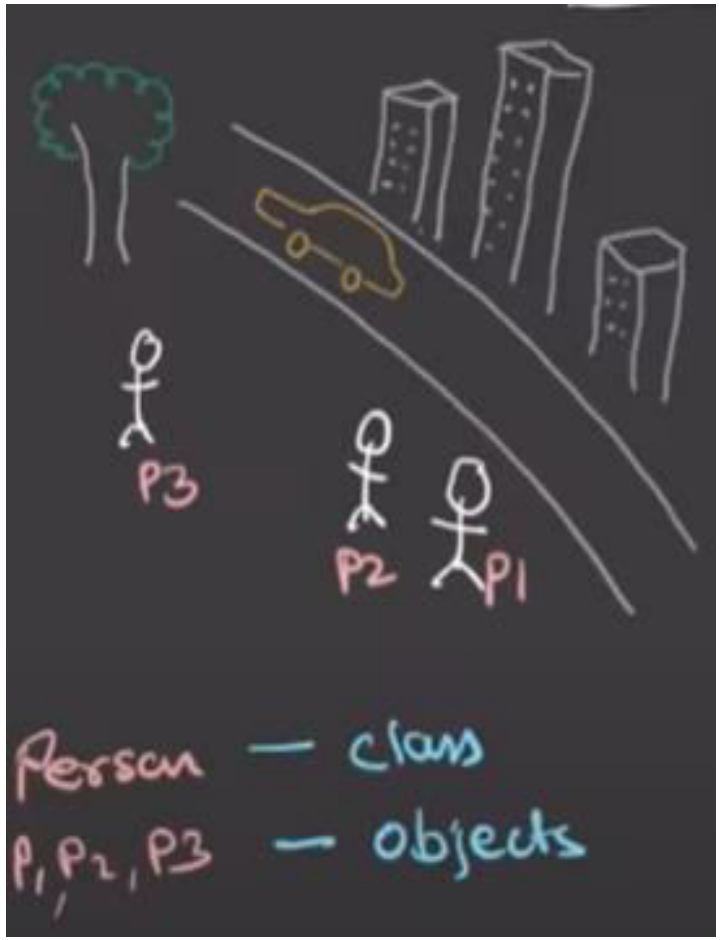
- 1.**Object-oriented programming** (OOP) is a programming paradigm based on the concept of "**objects**".
- 2.Object is something which can contains both **data** and **code**.
- Data in the form of properties (often known as attributes)
- Code in the form of methods (actions object can perform).

Real World Analogy



- Python is an object-oriented language like other general-purpose programming languages
- The concept of Object-Oriented Programming(OOP) in Python focuses on creating "objects" and making reusable codes
- Our world is full of objects like car, dog, tiger, tree or a balloon

Object



- Real World Entity is known as an object.
- Object of same nature or characteristcs belongs to the same category, known as **class.**

- **Noun** **Verb**

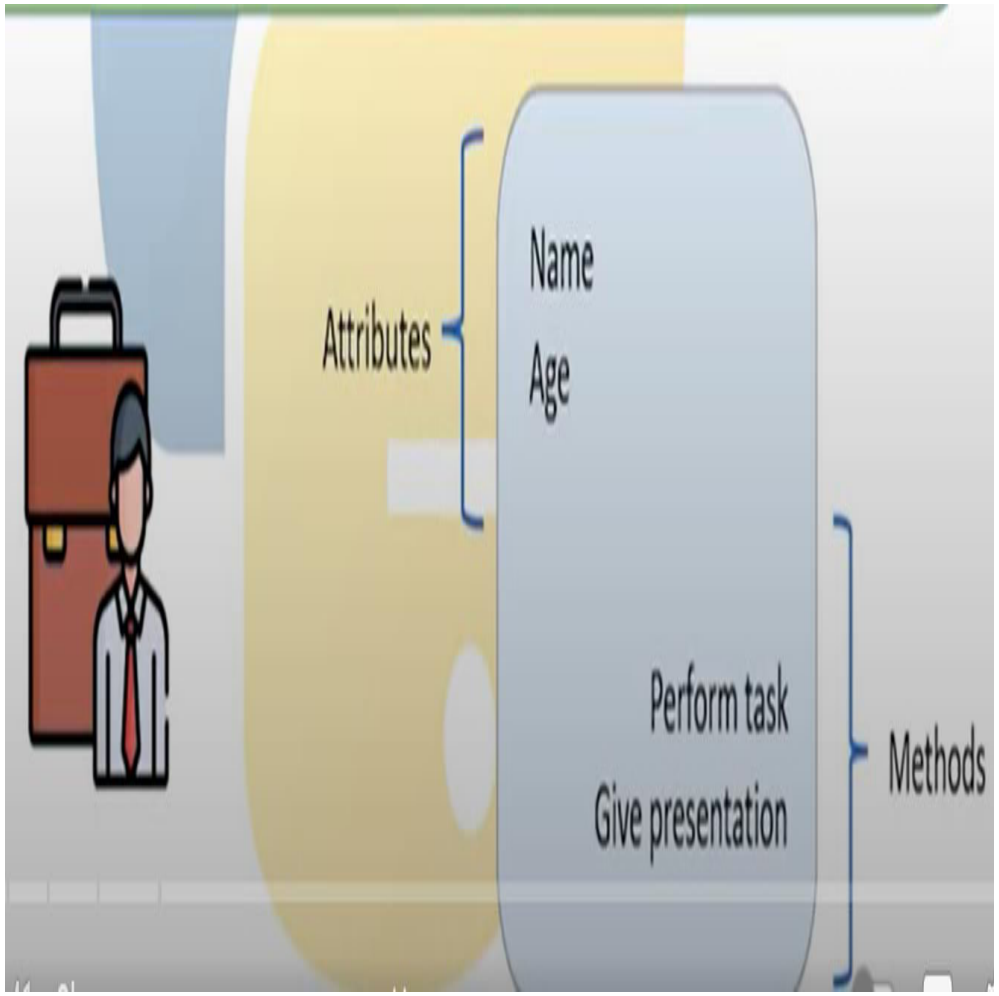
1. Common Noun

   e.g. Teacher --- Class

2. Proper Noun

   e.g. Dr. Manish --Object

Class



- **A class** is a blueprint for the object.
- A class contains the properties (attribute) and action (**behavior)** of the object.
- **Properties** represent variables, and the **methods** represent actions.
- Hence class includes both variables and methods.

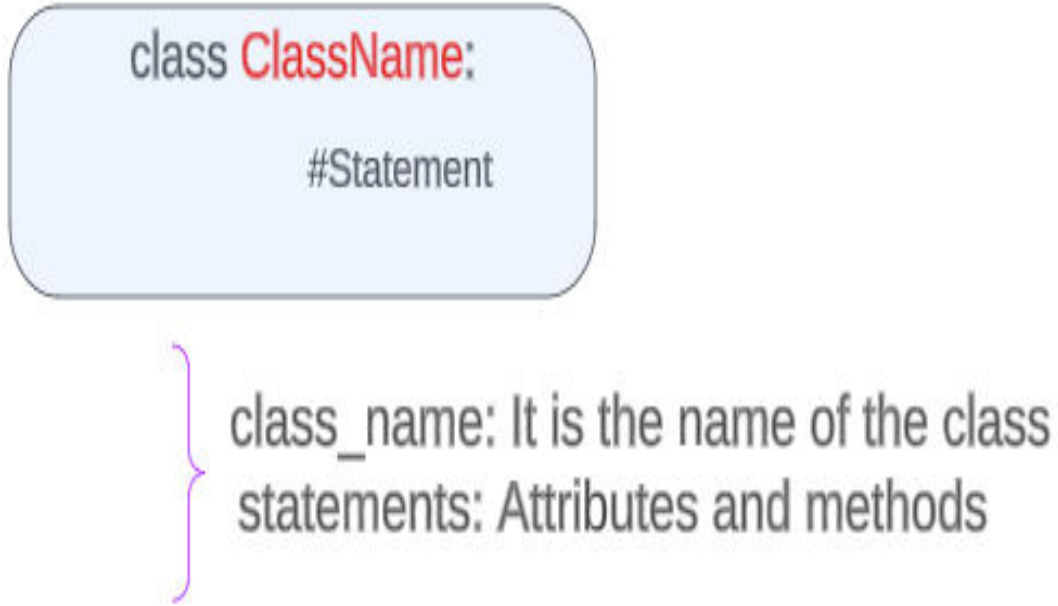
- Example: an Employee class then it should contains an attributes and methods i.e. email id, age,salary etc.

Class in python

- **Class**: The class is a user-defined data structure that binds the data members and methods into a single unit.
- A python class is a group of attributes and methods.
- Attributes: It is represented by variable that contains data.
- Method: performs an action of task. It is similar to function.

Class in python

- In Python, class is defined by using the **class** keyword. The syntax to create a class is given below.




```
In [5]: class Car:   #Blueprint of Class
   ...:     pass
   ...:             #Define Attributes ant Method
   ...: car1=Car()   #Create object of class
   ...: print(type(car1))
<class '__main__.Car'>

In [6]:
```

Object in python

- An Object(Instance) is known as instantiation of a class.
- object is an entity that has a state and behavior associated with it.
- An Object is class type variable **or class instance**. To use a class, we should create an object to the class.

Object

Syntax: object name= class name()

object name =class name(arg)

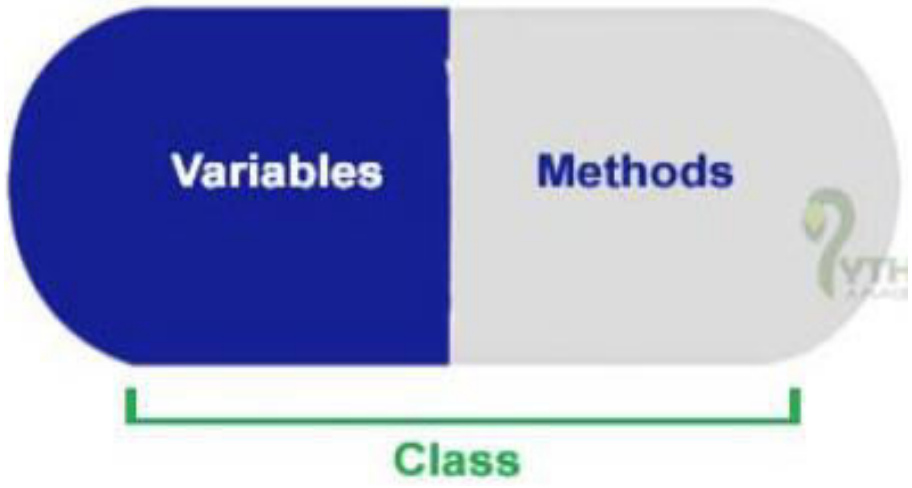
Example:

```
In [2]: class Car:      #Blueprint of Class
   ...:     pass
   ...:                 #Define Attributes ant Method
   ...: car1=Car()      #Create object of class
   ...: car2=Car()
   ...: car1.window=2           #properties of car
   ...: car1.tyre=4
   ...: car2.window=4
   ...: car2.tyre=6
   ...: print(car1.tyre)        #print values
   ...: print(car2.tyre)
4
6

In [3]:
```

Key Principles of OOP

1. Encapsulation
2. Data Hiding
3. Abstraction
4. Polymorphism
5. Inheritance

OOP with python




```
In [8]: class Car:    #Blueprint of Class
   ...:     a=10      #variable
   ...:     def disp(self):      #Define Method
   ...:         print("welcome to GPG")
   ...:
   ...: car1=Car()    #Create object of class
   ...: car1.disp()
   ...: car2=Car()
welcome to GPG

In [9]:
```

- Encapsulation
- An Act of combining properties and method related to the same entity is known as Encapsulation.
- For example: person class
- Attributes(variables): **pid,name,age**
- Methods(Functions): getName(),setName(),getAge(),editAge()

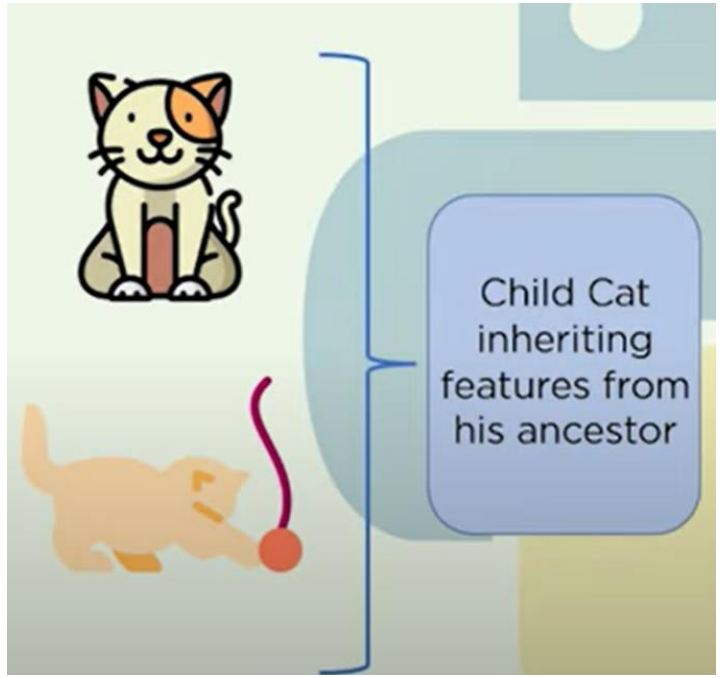
Data Hiding

- Data Hiding is a technique to hide internal object details (data members).
- Data Hiding guarantees exclusive data access to class members only and protects and maintains object integrity.

Abstraction

- Abstraction is hiding unnecessary implementations details to reduce the complexity and increase efficiency.
- The user is only able to view basic functionalities whereas the internal details are hidden.

Inheritance



Inheritance

- The inheritance is the process of acquiring the properties of one class to another class.
- It Provides reusability of code
- For example, a sports car can inherit the properties and methods of a car.

Polymorphism

- Polymorphism consists of two words, "poly" and "morphs". Poly means many, and morph means shape
- Polymorphism means using a familiar interface for multiple forms (data types). In simple terms, we understand that a task can be performed in various ways
- + operator is used for integer data types to perform arithmetic addition operations, whereas for string data types, + operator is used to perform concatenation. This is one of the examples of polymorphism in Python

Advantages of OOP

- **Reusability:** OOP code can be reused by creating objects from classes. This can save time and effort when developing software.
- **Modularization:** OOP code can be divided into modules, which makes it easier to understand and maintain.
- **Abstraction:** OOP allows us to hide the implementation details of an object, which makes the code more robust and easier to maintain.
- **Encapsulation:** OOP allows us to bind data and methods together, which makes the code more secure and efficient.
- **Inheritance:** OOP allows us to create new classes that inherit the properties and methods of existing classes. This makes it easier to create complex applications.
- **Polymorphism:** OOP allows us to create objects that can take on different forms. This makes it more flexible and powerful.

Any Questions???

Types of Methods in OOP

V. M. Vasava

GPG,Surat

IT Dept.

Agenda

INTRODUCTION ABOUT METHODS IN OOP

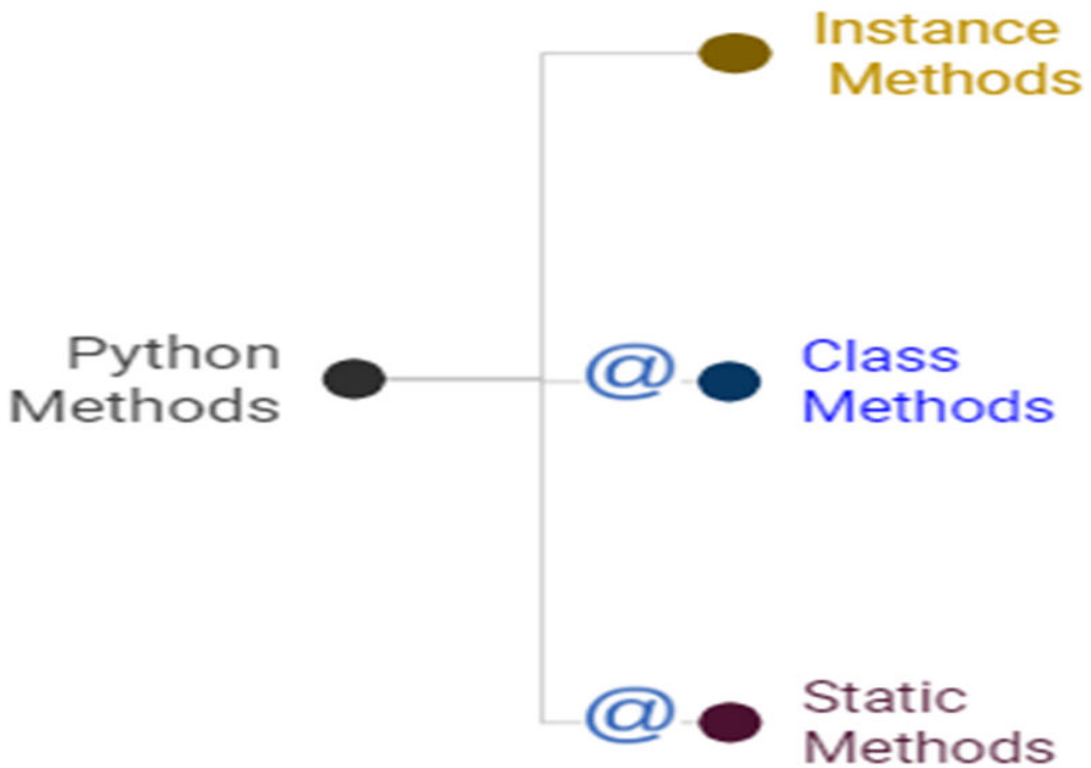
DIFFERENT TYPES OF METHODS

ACCESS METHODS WITH EXAMPLE

Methods in python

- Methods are functions defined inside the body of a class
- The method is a function that is associated with an object
- In Python, a method is not unique to class instances. Any object type can have methods
- Methods define the behavior of an object

Types of Method



- Most Common
- Must have self parameter
- No decorator needed
- Can be accessed through object (instance of Class)

- Doesn't need self parameter
- Need cls as parameter
- Need decorator @classmethod
- Can be accessed directly through the class. Don't need instance of class.

- Doesn't need self parameter
- Doesn't need self or cls as parameter
- Need decorator @staticmethod
- Can only access variables passed as argument.
- Static method cannot be accessed through class or it's instance.

1.Instance Method

- Instance method are the methods which act upon the **instance object variable** of the class.
- The first parameter for instance methods should be self variable which refers to instance.
- The instance method performs a set of actions on the data/value provided by the instance variables.
- It can access or modify the object state by changing the value of instance object variables.

Syntax:

```
def method_name(Self):
        function body
 def method_name(Self,f1,f2):
        function body
```

Without parameter

With parameter wise

Create and Access

Instance Object Variable

1. through __init__() via self
2. through instance method via self
3. through instance object

Class object variable

1. inside the class
2. through class name

Example

```
In [9]: class SavingAccount:
   ...:     def __init__(self):  #Instance Method
   ...:         self.acctname="V. M. Vasava"
   ...:     def show(self):    #Instance Method
   ...:         print(self.acctname)  #to Access variable via self
   ...: a=SavingAccount()
   ...: a.show()  #To access instance method via Instance Object
   ...: print(a.acctname) #To access variable through instance object
V. M. Vasava
V. M. Vasava
```

Instance method with argument

```
In [10]: class Student:
    ...:     def __init__(self,maths,science,eng): #instance method
with argument
    ...:         self.m=maths
    ...:         self.s=science
    ...:         self.e=eng
    ...:     def avg(self):
    ...:         return(self.m+self.s+self.e)/3
    ...:
    ...: s=Student(67, 53, 45)    #Create Instance object variable
    ...: s.avg()    #Call Instance method through object
Out[10]: 55.0
```

With Argument

Example

```
class Student:
    def __init__(self,nm,m):  #Instance Method with argument
        self.name=nm
        self.marks=m
    def display(self):
        print(self.name,self.marks)
    def change_data(self):
        self.name="abc"
        self.marks=30
std1=Student("Ankita",15)   #Create Instance object variable
std2=Student("Nidhi",10)
std3=Student("Ashwini",13)
print(std1.__dict__)
std1.display()   #Call instance method via object
std1.change_data()
print(std1.__dict__)
```

```
{'name': 'Ankita', 'marks': 15}
Ankita 15
{'name': 'abc', 'marks': 30}
```

2.Class Method

- Class method are the method which act upon the class variable or static variable of the class.
- Decorator @classmethod need to write above the class method.
- By default, the first parameter of class method is **cls** which refer to the class itself.
- Syntax:

```
@classmethod
Def method_name(cls):
        method body
Def method_name(cls,f1,f2):
        method body
```

Decorator

Class method without parameter/Formal arguments

Class method with parameter/Formal arguments

Example

```
In [8]: class Employee:
   ...:     company_name="Infosys"  #Class object variable
   ...:     def __init__(self,nm,sal):
   ...:         self.name=nm
   ...:         self.salary=sal
   ...: e1=Employee("PAP",60000)
   ...: e2=Employee("VMV",65000)
   ...: e2.company_name="TCS"    # Can't Modify
   ...: print(Employee.company_name)  #To access using classname
   ...: print(e1.company_name)       #to access using instance object var.
   ...: print(e2.__dict__)
   ...: print(Employee.company_name)
Infosys
Infosys
{'name': 'VMV', 'salary': 65000, 'company_name': 'TCS'}
Infosys
```

Example

We can access class method using className:

Syntax: classname.methodname

```
class Employee:
    company_name="Infosys"  #Class object variable
    def __init__(self,nm,sal):
        self.name=nm
        self.salary=sal
    @classmethod
    def getCompany(cls):
        cls.company_name="TCS"
        print("Company Name:",cls.company_name)
e1=Employee("PAP",60000)
e2=Employee("VMV",65000)
Employee.getCompany()     #To access using classname
print(e2.company_name)          #to access using instance object var.
print(e2.__dict__)
```

```
Company Name: TCS
TCS
{'name': 'VMV', 'salary': 65000}

In [4]:
```

Class Method with argument

- We can access class method below syntax:
- Syntax: classname.methodname(arg1,arg2,)

```
In [4]: class Mobile:
   ...:     fp='Yes'       #class variable
   ...:     @classmethod  #Decorator
   ...:     def show_model(cls,r):   #class method with argument
   ...:         cls.ram=r
   ...:         print("Finger Print:",cls.fp)
   ...:         print("RAM",cls.ram)  #Accessing class variable inside method
   ...: realme=Mobile()
   ...: Mobile.show_model('16GB')  #calling method with aargument
Finger Print: Yes
RAM 16GB
```

3.static method

- A static method is a general utility method that performs a task in isolation.
- We use static method when we want to pass some values from outside &perform some action in method.
- **@staticmethod Characteristics**
- Declares a static method in the class.
- It cannot have cls or self parameter.
- The static method cannot access the class attributes or the instance attributes.
- The static method can be called using ClassName.MethodName() and also using object.MethodName().
- It can return an object of the class.

Static Method

- Syntax:

```
@staticmethod
def func(args, ...)
    body of method
```

Decorator

Static method with argument

```
@staticmethod
def func()
    method body
```

Static method without argument

Static Method without argument

- Syntax: ClassName.MethodName()

```
In [3]: class Mobile:
   ...:     @staticmethod     #Decorator
   ...:     def show_model():  #static method
   ...:         print("Realme x")
   ...: realme=Mobile()
   ...: Mobile.show_model()  #call static method
Realme x

In [4]:
```

Example

```
class Bank:
    bank_name="SBI"
    rate_intrest=12.25
    @staticmethod
    def simple_interest(prin,n):  #Define static method
        si=(prin*n*Bank.rate_intrest)/100
        print(si)
prin=float(input("enter principle amount:"))
n=int(input("enter principle amount:"))
Bank.simple_interest(prin, n)  #To call static method
```

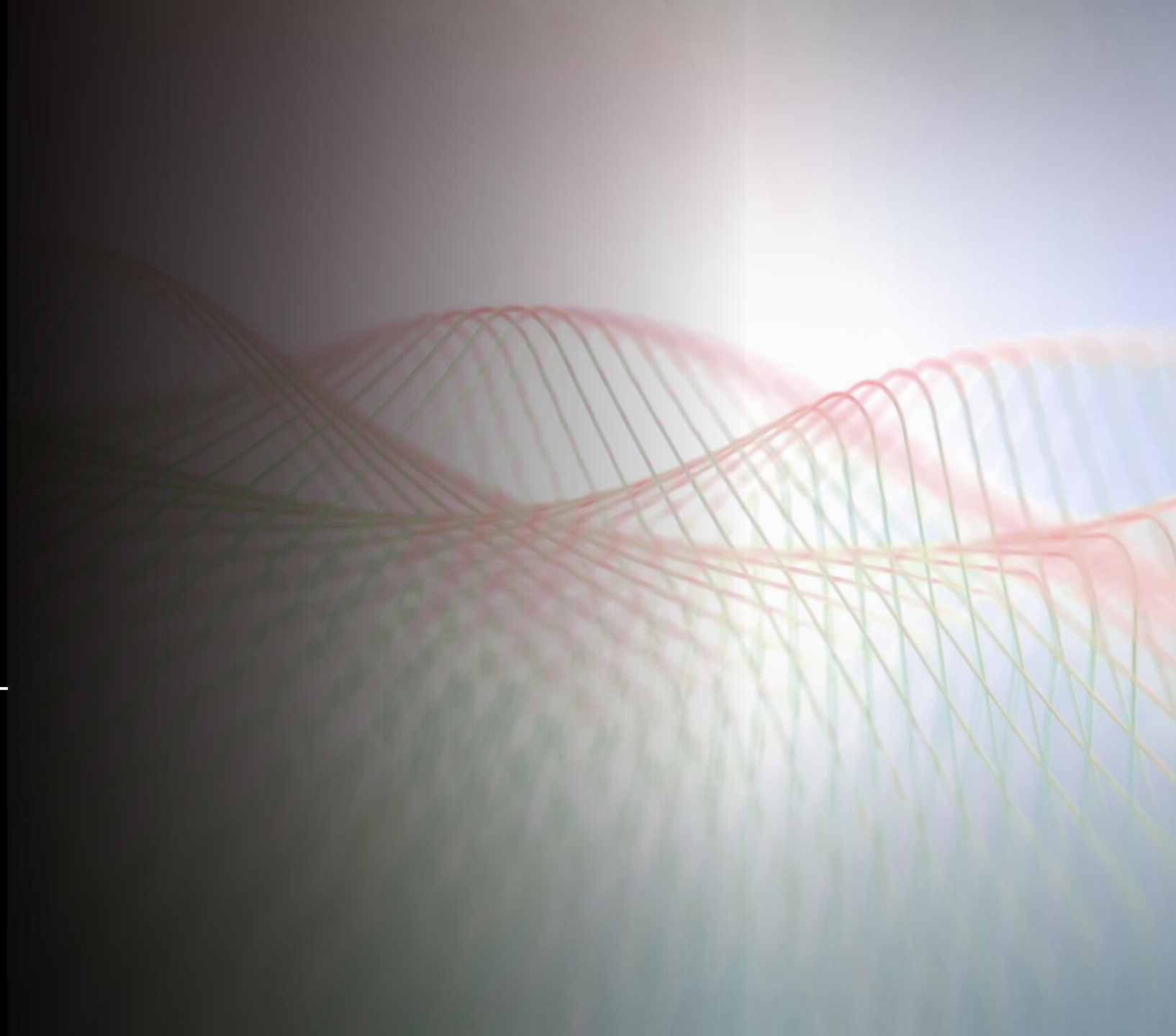
```
enter principle amount:29000
enter principle amount:3
10657.5

In [8]:
```

Any Questions????

Data Structure with Python

Mr. V.M.Vasava

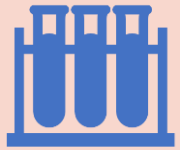
GPG,IT Dept.



Agenda

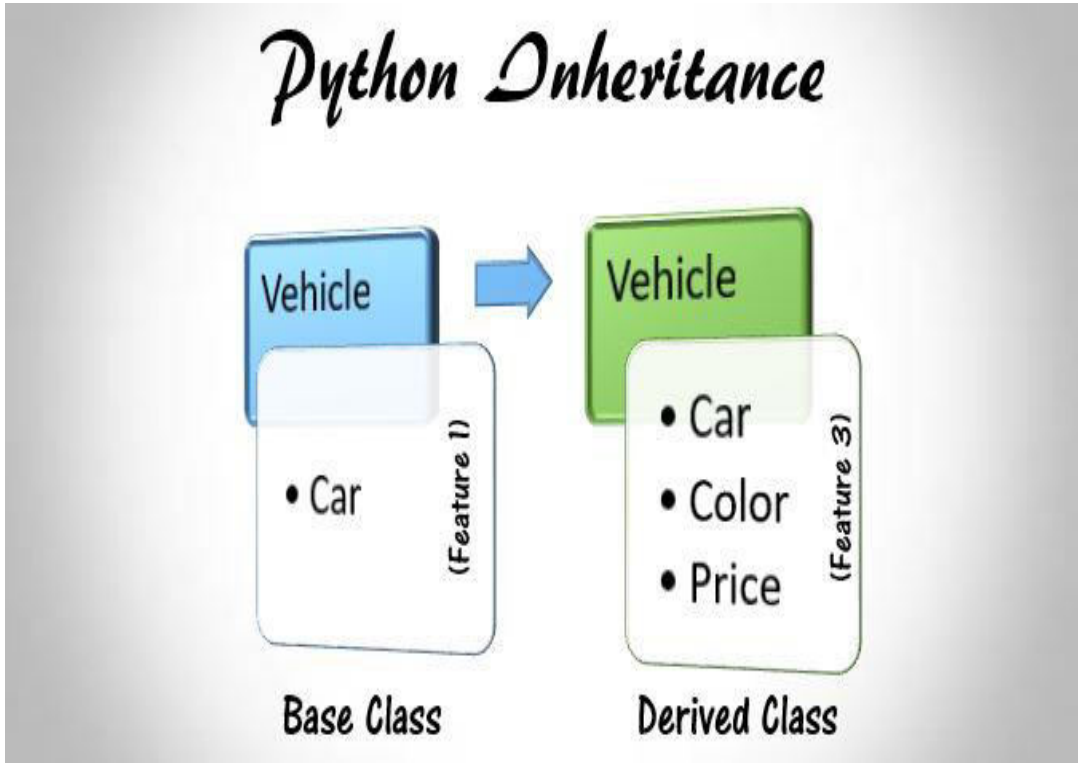
Inheritance

Types of Inheritance



Example

Inheritance



- Inheritance

The inheritance is the process of acquiring the properties of one class to another class.

- The **Parent class** is the class which provides features to another class. The parent class is also known as **Base class** or **Superclass**.
- The **Child class** is the class which receives features from another class. The child class is also known as the **Derived Class** or **Subclass**.

Declaration of child class

- In Python, we use the following general structure to create a child class from a parent class.
- **Syntax:**

```
class ChildClassName(ParentClassName):
                    Method of ChildClass
```

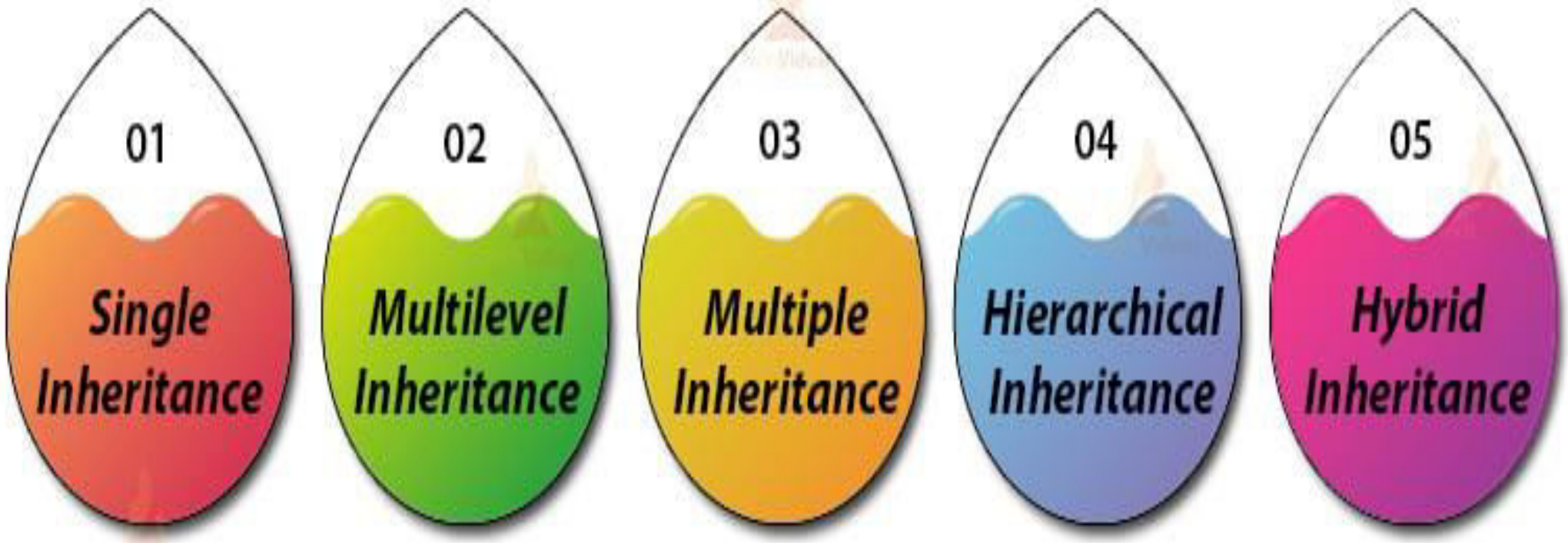
Ex. class Mobile(object):

member of child class

class Mobile

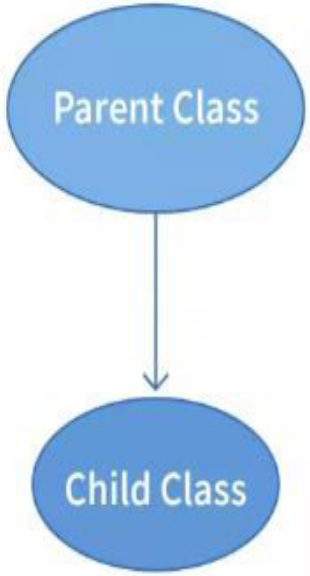
member of child class

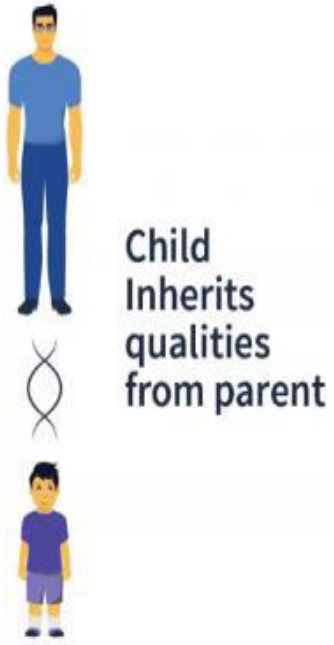
Types of Python Inheritance

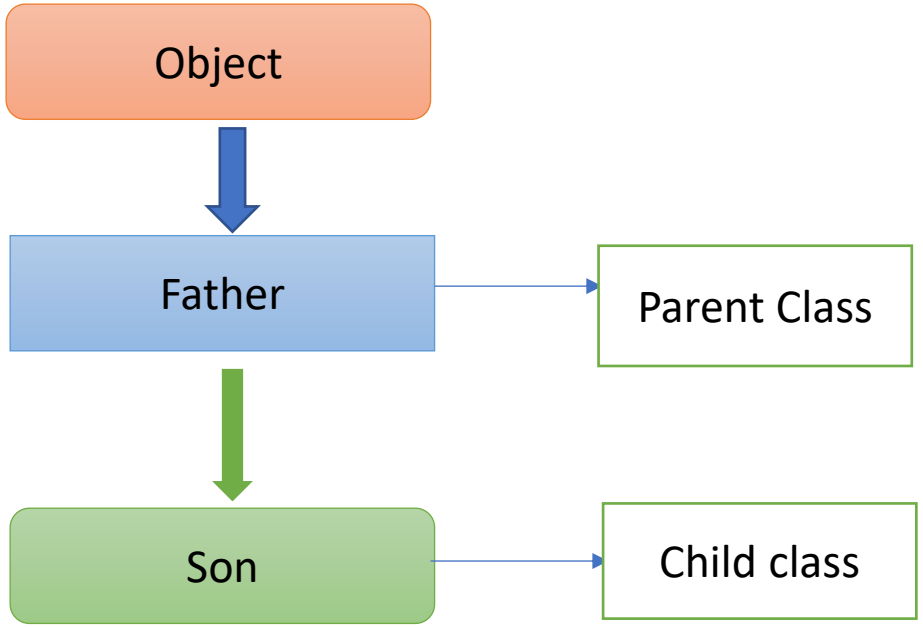


Single inheritance

- If a class is derived from one base class(parent class),it is called single inheritance.







1.Simple inheritance

Syntax:

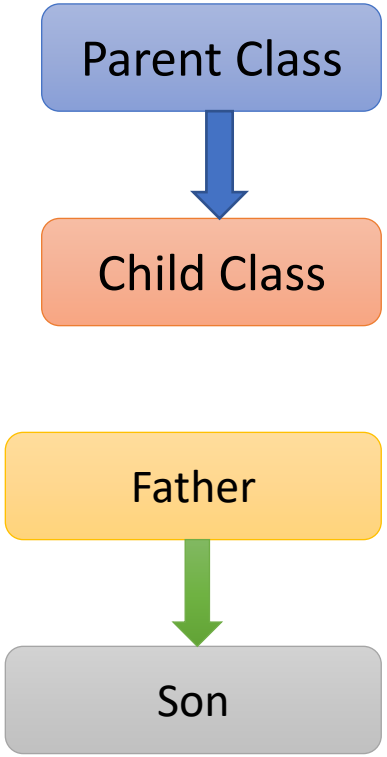
```
class ParentClassName(object):
        member of Parent class
class ChildClassName(ParentClassName):
        member of Child class
```

Example:

```
class Father:
    member of class Father
class son(Father):
    member of class son
```

Parent Class

Child class



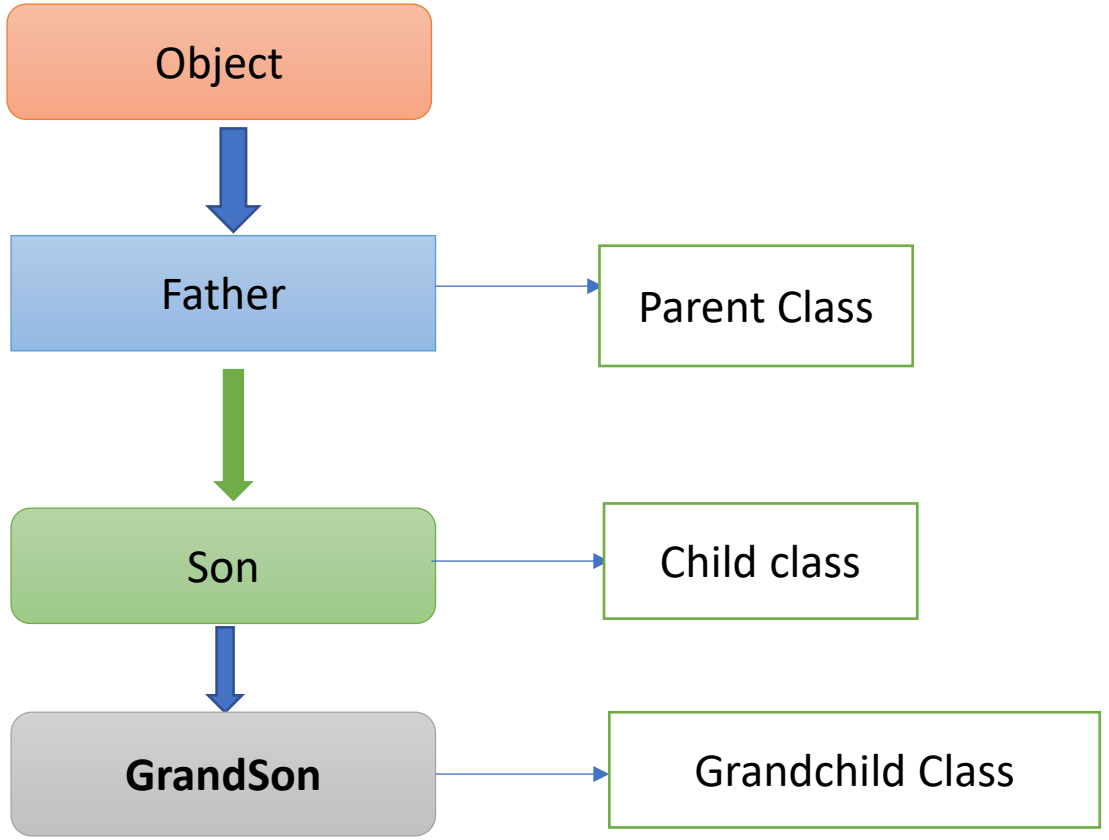
Example

```
#single inheritance
class Father:
    money=20000
    def show(self): #instance method
        print("parent class instance method")
    @classmethod
    def showmoney(cls): #classmethod
        print("parent class class method",cls.money)
class Son(Father):  #inherite parent class
    def disp(self):
        print("child class instance method")
s=Son()  #create object
s.disp()  #call method
s.show()
s.showmoney()
```

```
child class instance method
parent class instance method
parent class class method 20000

In [13]:
```

2.Multilevel Inheritance

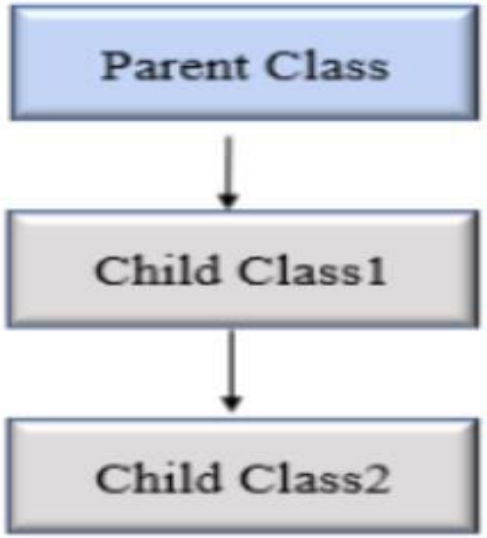
- In multilevel inheritance, a class inherits from a child class or derived class. one or more class act as a base class.



Declaration of Multilevel Inheritance

- **Syntax:**

```
class ParentClassName(object):
        Member of ParentClass
class ChildClassName(ParentClassName):
        Member of ChildClass
class GrandChildClassName(ChildClassName):
        Member of GrandChildClass
```



Python Multilevel Inheritance

Example

```
#multilevel inheritance
class Father:
    def showF(self): #instance method
        print("parent class  method")
class Son(Father):
    def showS(self): #instance method
        print("child class method")
class GrandSon(Son):
    def showG(self): #instance method
        print("Grandson class method")
obj=GrandSon()
obj.showG()  #call method
obj.showS()
obj.showF()
```

```
Grandson class method
child class method
parent class  method

In [6]:
```

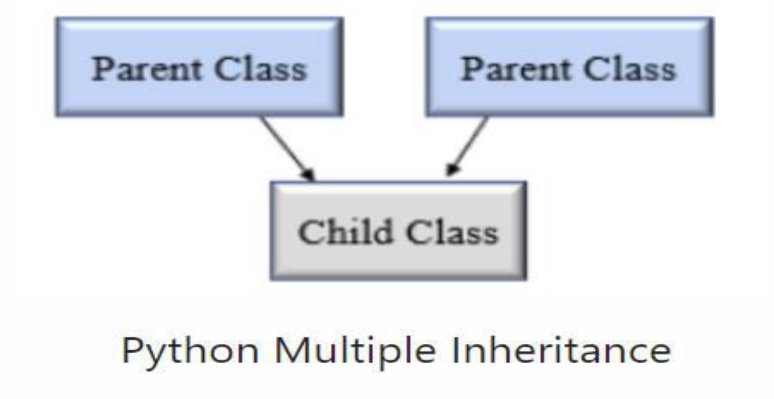
Super() method:

- The super() function is used to give access to methods and properties of a parent or sibling class.
- The super() function returns an object that represents the parent class.

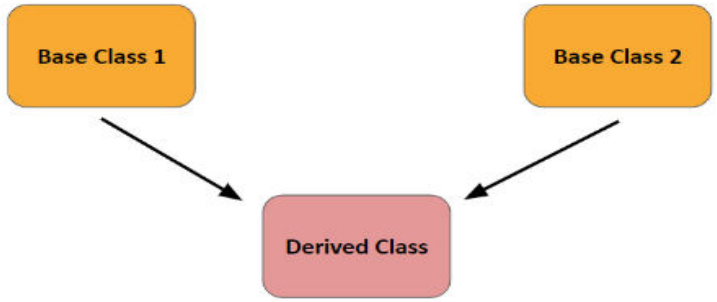
```
#multilevel inheritance
class Father:
    def __init__(self):
        print("Father class constructor")
    def showF(self): #instance method
        print("parent class  method")
class Son(Father):
    def __init__(self):
        print("Son class constructor")
    def showS(self): #instance method
        print("child class method")
class GrandSon(Son):
    def __init__(self):
        super().__init__()    #calling son class constructor
        print("GrandSon class constructor")
    def showG(self): #instance method
        print("Grandson class method")
obj=GrandSon()
```

```
Son class constructor
GrandSon class constructor

In [9]:
```

3.Multiple Inheritance



Python Multiple Inheritance



- In multiple inheritance, one child class can inherit from multiple parent classes.
- So here is one child class and multiple parent classes.
- Syntax:

```
class ParentClassName1(object):
                    Member of ParentClass
class ParentClassName2(object):
                    Member of ParentClass
class ChildClassName(ParentClassName1,
ParentClassName2):
                    Member of ChildClass
```

Example

```
class Vehicle:  #Blueprint of Vehicle //Base Class
    def info(self):
        print("This is Vehicle")
class Car(Vehicle):    #Derived class1
    def car_info(self, name):
        print("Car name is:", name)
class Motorbike(Vehicle):#Derived class2
    def bike_info(self, name):
        print("Motorbike name is:", name)

obj1 = Car()
obj1.info()
obj1.car_info('BMW')
obj2 = Motorbike()
obj2.info()
obj2.bike_info('Splender')
```

```
This is Vehicle
Car name is: BMW
This is Vehicle
Motorbike name is: Splender

In [2]:
```

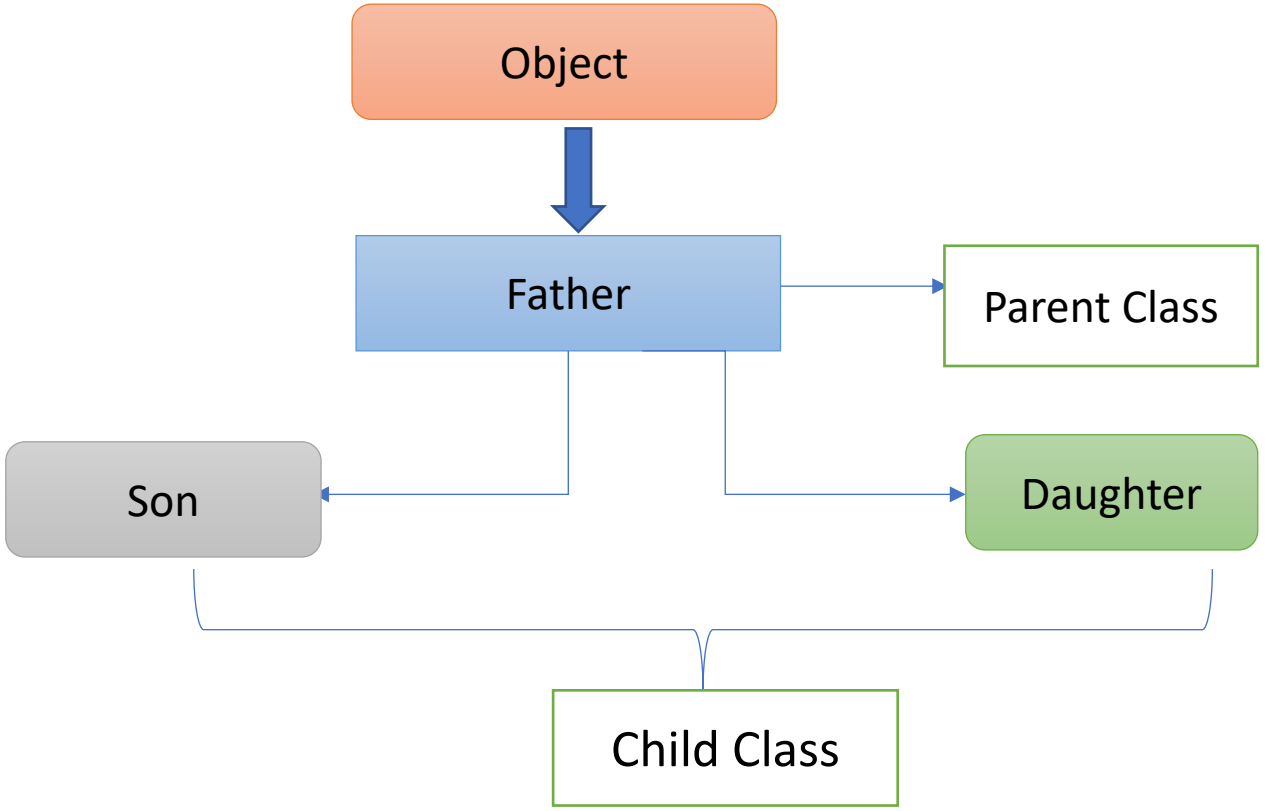
Example

```
#Multiple Inheritance
class Father:
    def showF(self): #instance method
        print("Father class  method")
class Mother:
    def showM(self): #instance method
        print("Mother class  method")
class Son(Father,Mother):
    def showS(self): #instance method
        print("Son class  method")
s=Son()
s.showS()
s.showM()
s.showF()
```

```
Son class  method
Mother class  method
Father class  method

In [10]:
```

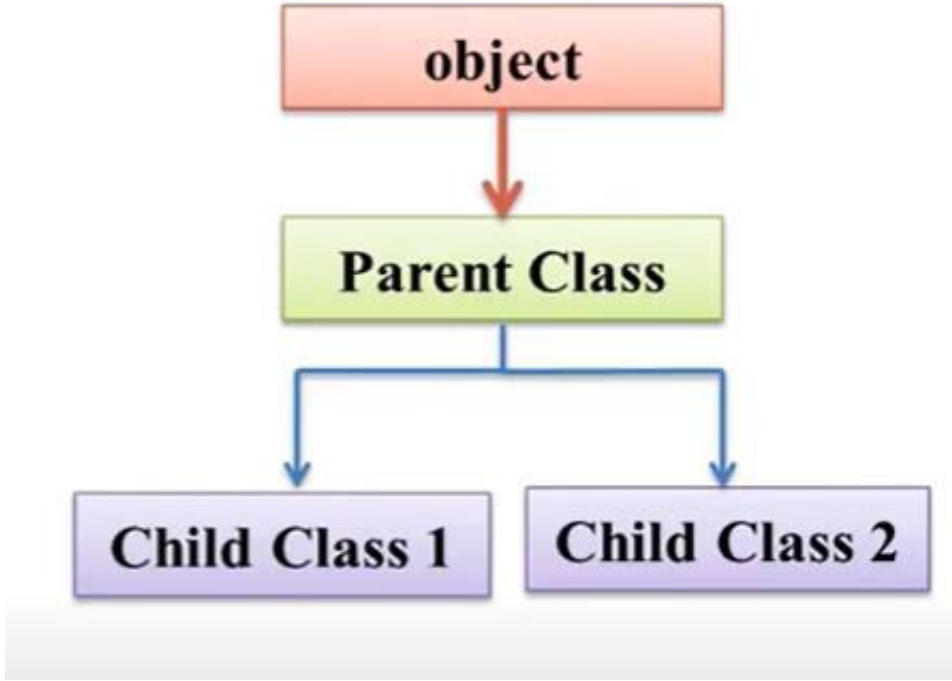
Hierarchical Inheritance

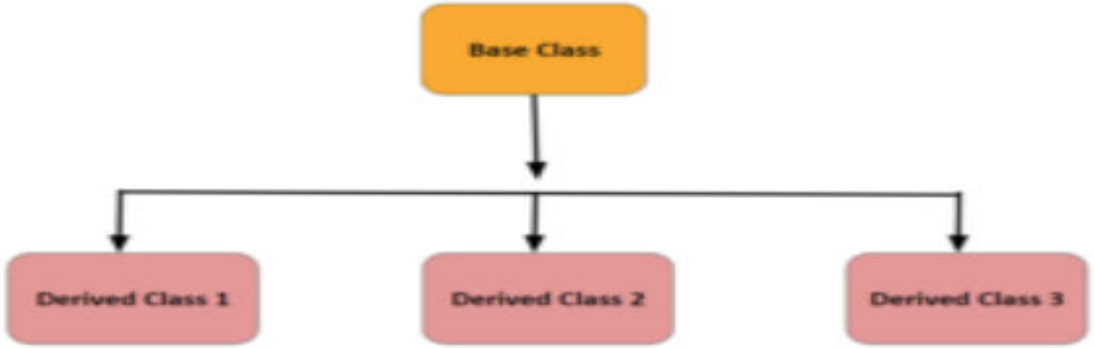
- In this type of inheritance, two or more child classes derive from one parent class.



Syntax:

```
class ParentClassName(object):
                    Member of ParentClass
class ChildClassName1(ParentClass):
                    Member of ChildClass1
class ChildClassName2(ParentClass):
                    Member of ChildClass2
```



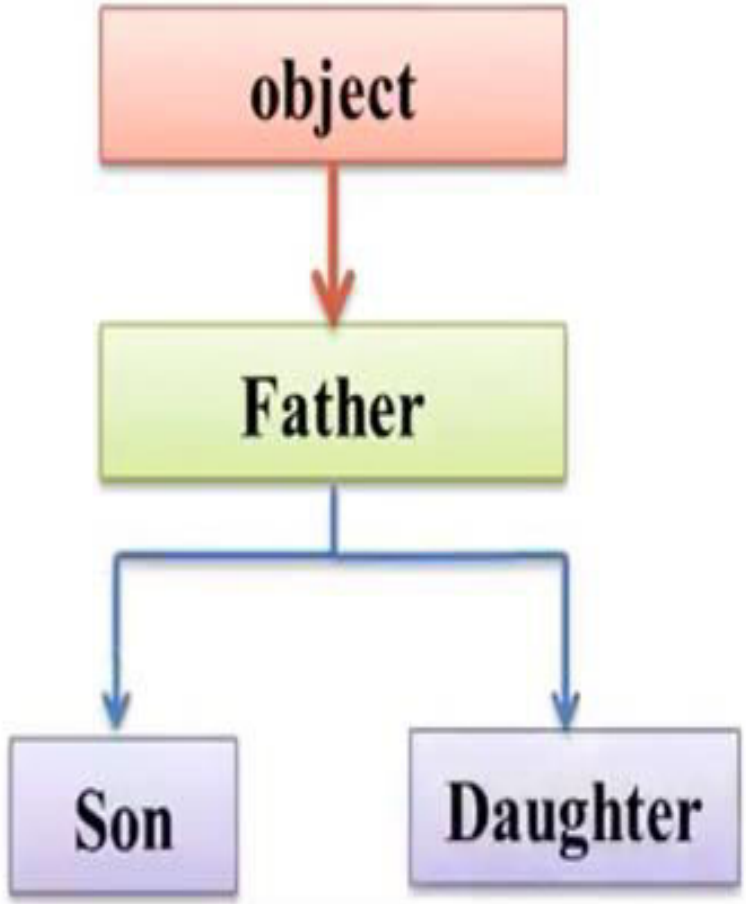


```
class Father (object):
    members of class Father

class Son (Father):
    members of class Son

class Daughter (Father):
    members of class Daughter
```

Parent Class

Child Class

Child Class



Example

```
#Hirierachical Inheritance
class Father:
    def showF(self): #instance method
        print("Father class  method")
class Son(Father):
    def showS(self): #instance method
        print("Son class  method")
class Daughter(Father):
    def showD(self): #instance method
        print("Daughter class  method")
s=Son()
s.showS()   #access self method
s.showF()   #acess Father method
```

```
Son class  method
Father class  method

In [2]:
```

Example

```
class Vehicle:   #Blueprint of Vehicle //Base Class
    def info(self):
        print("This is Vehicle")
class Car(Vehicle):    #Derived class1
    def car_info(self, name):
        print("Car name is:", name)
class Motorbike(Vehicle):#Derived class2
    def bike_info(self, name):
        print("Motorbike name is:", name)

obj1 = Car()
obj1.info()
obj1.car_info('BMW')
obj2 = Motorbike()
obj2.info()
obj2.bike_info('Splender')
```

```
This is Vehicle
Car name is: BMW
This is Vehicle
Motorbike name is: Splender

In [3]:
```

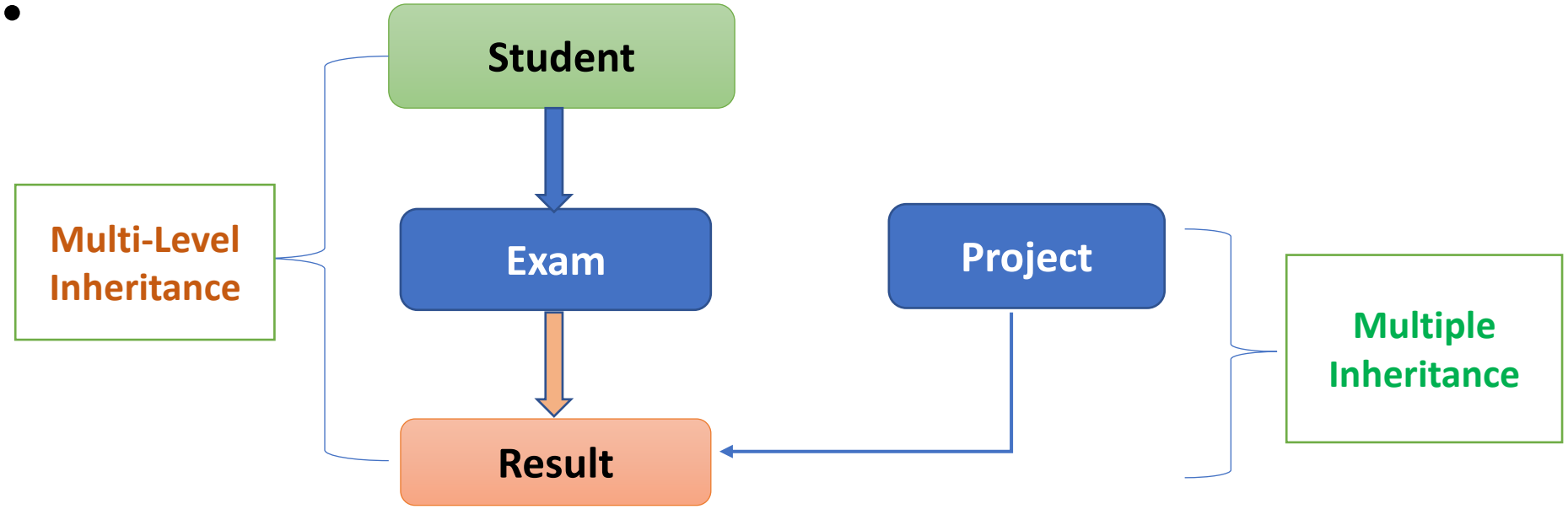
Hierarchical Inheritance using constructor

```
#Hirierachical Inheritance
class Father:
    def __init__(self):
        print("Father class constructor")
    def showF(self): #instance method
        print("Father class  method")
class Son(Father):
    def __init__(self):
        super().__init__()   #calling Father class constructor
        print("Son class constructor")
    def showS(self): #instance method
        print("Son class  method")
class Daughter(Father):
    def __init__(self):
        print("Daughter class constructor")
    def showD(self): #instance method
        print("Daughter class  method")
s=Son()
```

```
Father class constructor
Son class constructor

In [3]:
```

Hybrid Inheritance

- Combination of two or more types of inheritance to design a program.
-



Example

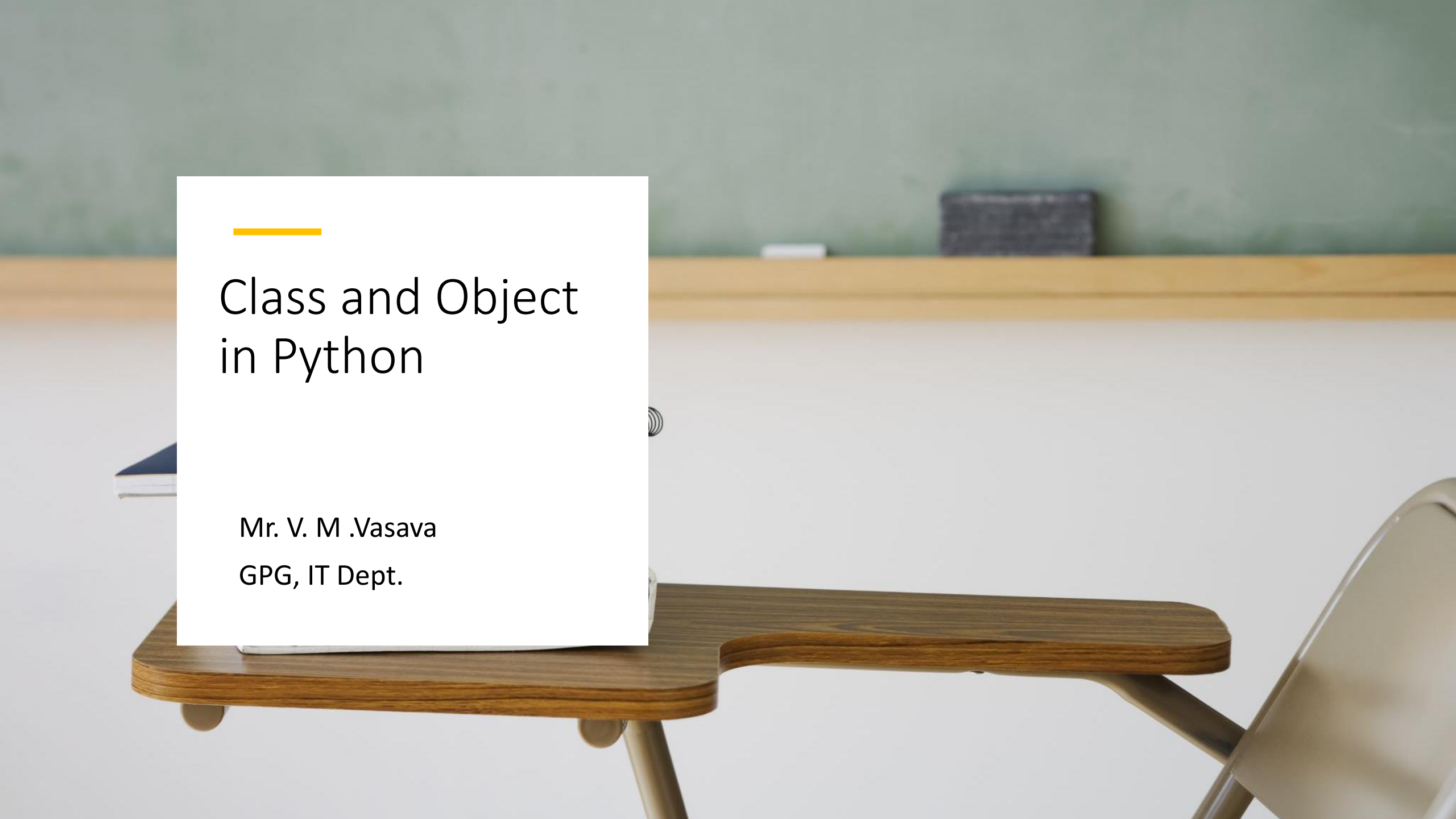
```
class School:
    def fun1(self):
        print("i am in school class")
class Teacher1(School):
    def fun2(self):
        print("i am Teacher1")
class Teacher2(School):
    def fun3(self):
        print("i am Teacher2")
class Student(Teacher1,Teacher2):
    def fun4(self):
        print("i am Student")
obj=Student()
obj.fun1()
obj.fun2()
obj.fun3()
obj.fun4()
```

```
i am in school class
i am Teacher1
i am Teacher2
i am Student

In [6]:
```

Advantages of Inheritance

- it provides Code reusability, readability, and scalability.
- It reduces code repetition. You can place all the standard methods and attributes in the parent class. These are accessible by the child derived from it.
- By dividing the code into **multiple classes**, the applications look better, and the error identification is easy.

Any Questions ???



Class and Object in Python

Mr. V. M .Vasava

GPG, IT Dept.

Agenda

1.Introduction about Class

2.Basic Naming rules for create class in python

3.Object

4.Accessing class member using object

5. self variable in class

6.Constructors

Class in python

- **Class**: The class is a user-defined data structure that binds the data members and methods into a single unit.
- A python class is a group of attributes and methods.
- Attributes: It is represented by variable that contains data.
- Method: performs an action of task. It is similar to function.

1.Class -Class keyword is used to create class.
2.Object- Object represents the base class name from where all classes in python are derived. This class also derived from object class and its optional.
3.__init__(): This method is used to initialize the variables. This is a special method which automatically calls.
4.Self:- self is a variable which refers to current class instance/object.

Create class

- Class ClassName(object):
  def __init__(self):
    Self.variable_name=value
    Self.variable_name='value'

Attributes

Method

def method_name(self):
  Body of Method

```
In [3]: class Myclass(object):
   ...:     def show(self):
   ...:         print("i am in method")
   ...: x=Myclass()
   ...: x.show()
i am in method

In [4]:
```

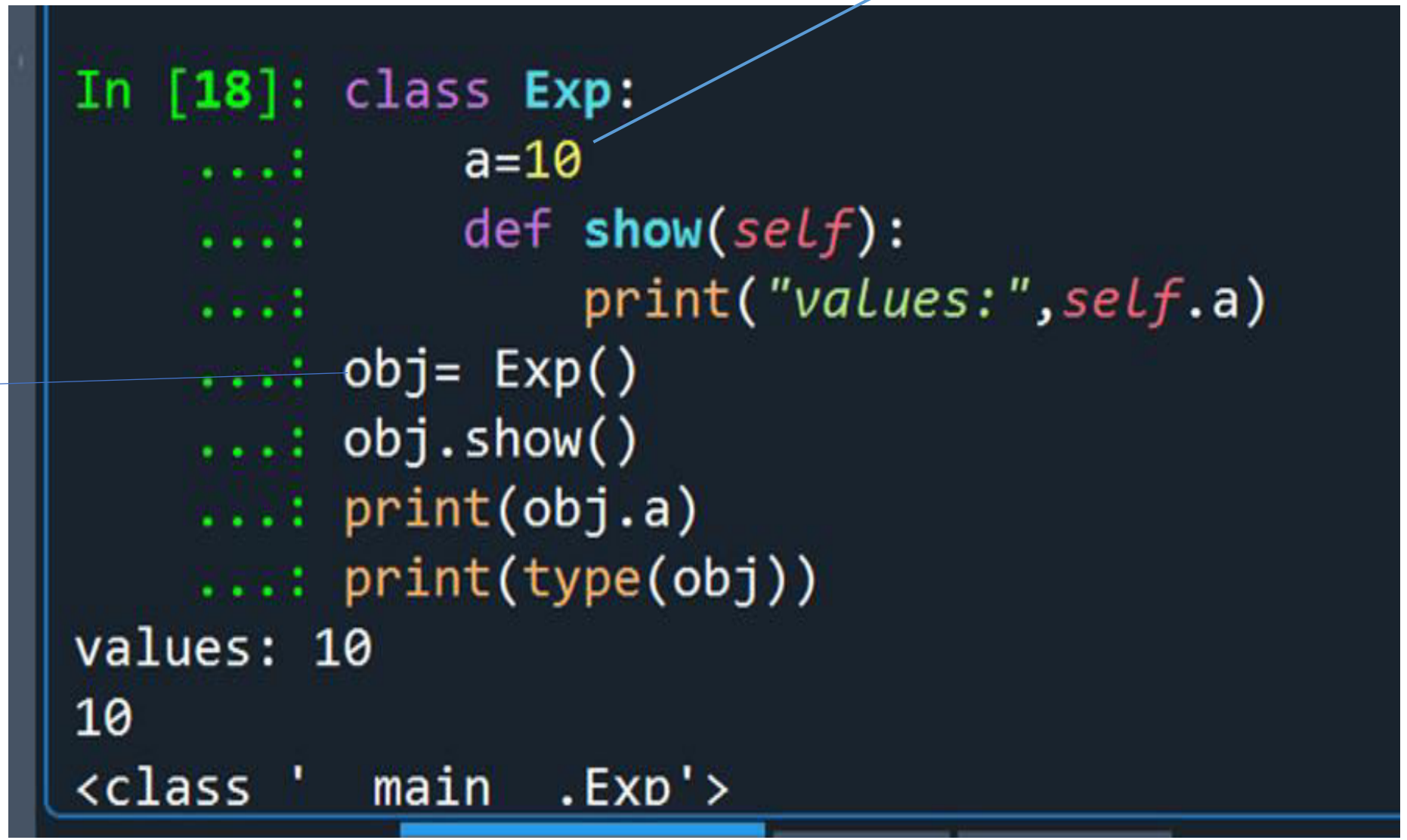
Rules

- The class name can be any valid identifier.
- It can’t be python **reserved keyword**.
- A valid class name start with letter, followed by any number of letter,number or underscores(_).
- A Class name generally start with **Capital Letter**.

Example

Class object Variable

Instance Object



Continued..

- Instance Object are empty object at the time of creations.
- Variables in different type in class:

1. Class object variable
2. Instance object variable
3. Local variable
4. Global variable

__init__() method

- In Python classes, “__init__” method is reserved.
- It is automatically called when you create an object using a class.
- it is used to initialize the variables of the class. It is equivalent to a constructor.

```
In [8]: class Test:    #Blueprint of class
   ...:       x=10      #class object variable
   ...:       y=20
   ...:       def __init__(self):  #Constructor
   ...:             self.x=4          #Self,x =local variable
   ...: t=Test()    #Instance object
   ...: print(t.x) #accessing value
4
```

Self variable

```
In [3]: class Myclass(object):
   ...:     def show(self):
   ...:         print("i am in method")
   ...: x=Myclass()
   ...: x.show()
i am in method

In [4]:
```

self is a default variable that contains the memory address of the current object.

This variable is used to refer all the instance variable and method.

When we create object of a class, the object name contains the memory location of the object.

This memory location is internally passed to *self*, as *self* knows the memory address of the object so we can access variable and method of object.

self is the first argument to any object method because the first argument is always the object reference. This is automatic, whether you call it *self* or not.

```
def __init__(self):
```

Object

```
In [18]: class Exp:
    ...:     a=10
    ...:     def show(self):
    ...:         print("values:",self.a)
    ...: obj= Exp()
    ...: obj.show()
    ...: print(obj.a)
    ...: print(type(obj))
values: 10
10
<class '__main__.Exp'>
```

- object is an entity that has a state and behavior associated with it.
- Object is class type variable **or class instance**. To use a class, we should create an object to the class.
- Instance creations represents allotting memory necessary to store the actual data of the variable.
- Each time you create an object of class copy of each variables defined in the class is created.
- **Syntax:** object name= class name()
- object name =class name(arg)

Accessing class members using object

We can access variable and method of a class using class object or instance of class.

Object_name.variable_name

Object_name.method_name

```
In [2]: class Dept:
   ...:     def __init__(self):
   ...:         self.sem="3 sem"
   ...:     def show(self):
   ...:         print("Welcome to IT",self.sem)
   ...:
   ...: dit=Dept()
   ...: #call method show using object
   ...: dit.show()
Welcome to IT 3 sem
```

```
class Classname :
    def __init__(self):
        self.variable_name = value
        self.variable_name = 'value'

    def method_name(self):
        Body of Method

    def method_name(self, f1, f2):
        Body of Method
```

Formal Argument

Formal Argument

```
class Classname :
    def __init__(self, f1, f2):
        self.variable_name = value
        self.variable_name = 'value'

    def method_name(self):
        Body of Method

    def method_name(self, f1, f2):
        Body of Method
```

Formal Argument

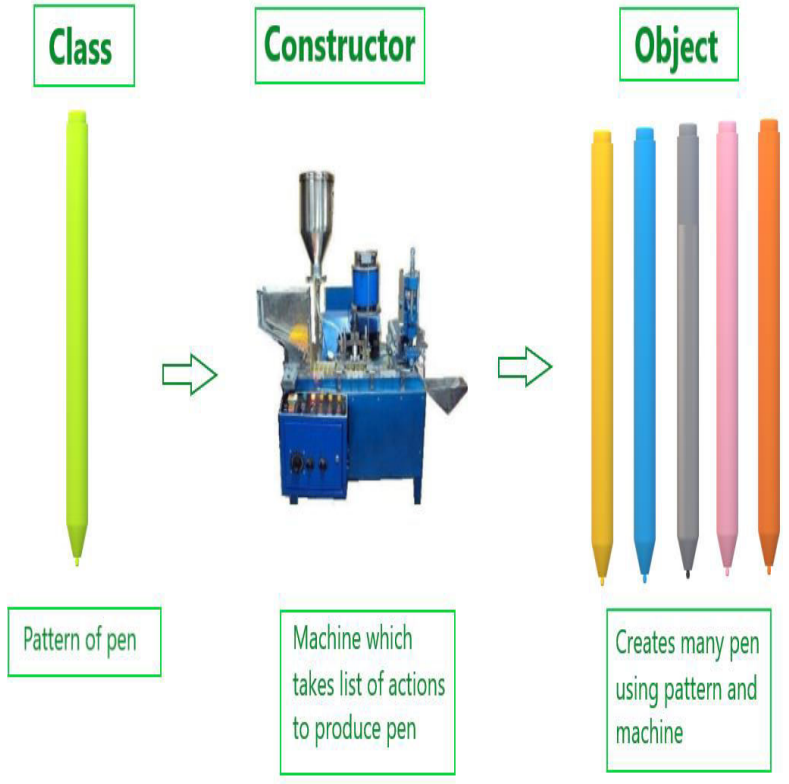
Example

```
In [11]: class Test:    #Blueprint of class
    ...:     x=10      #class object variable
    ...:     y=20
    ...:     def __init__(self,a):  #Constructor
    ...:         self.a=a           #Self,x =local variable
    ...: t=Test(4)   #Instance object
    ...: t1=Test(5)
    ...: print(t.a) #accessing value
    ...: print(t1.a) #accessing value
4
5
```

Example

```
class Test:
    x=10
    def f1(self):
        self.m1=4
    def __init__(self,a):
        self.a=a
t1=Test(5)
t2=Test(8)
t1.f1()
#All variable in above code
#x-->Class object variable
#m1-->local variable
#a-->local variable
#t1.a-->Instance object variable
#t2.a-->Instance object variable
#t1-->global variable
#t2-->global variable
#self-->Local variable
#Test-->global variable
```

Constructors

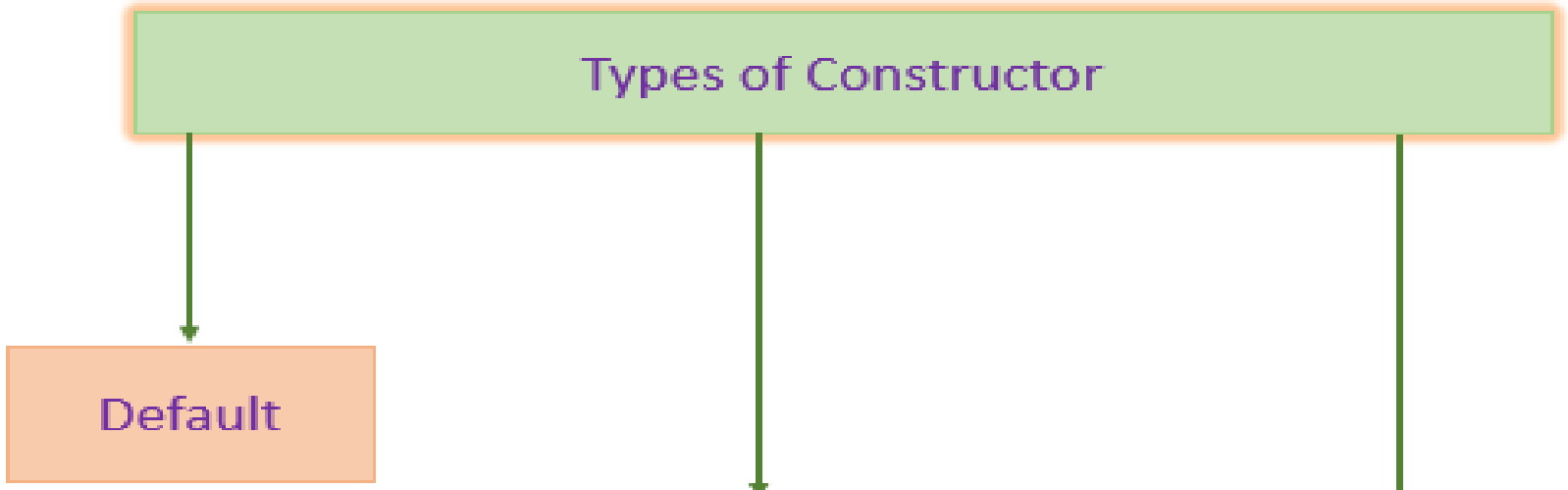


- The primary objective of Constructors is to assign values to the instance attributes of the class.
- A constructor is a special method in a class used to create and initialize an object of a class.
- This method is defined in class.
- The constructor is executed automatically at the time of object creation.
- **It cannot return any value other than none**.

Rules of Python Constructor

- It starts with the def keyword, like all other functions in Python.
- It is followed by the word init, which is prefixed and suffixed with double underscores with a pair of brackets, i.e., __init__().
- It takes an argument called self, assigning values to the variables.
- Other Constructors in python like __new__().

Types of Constructors




```
def __init__(self):
```

```
def __init__(self, v1, v2…, vn):
```

Non-Parametrized Constructor

A constructor without any arguments is called a non-parameterized constructor.

It has only one argument, self, in the constructor,

This type of constructor is used to initialize each object with default values.

The values of attributes inside the non-parameterized constructors are defined when creating the class and can not be modified while instantiating.

```
class Dress:
    def __init__(self):     #Non Parameterized constructor
        self.cloth = "Cotton"
        self.type = "T-Shirt"
    def get_details(self):
        print("Cloth =", self.cloth)
        print("Type =" , self.type)
t_shirt = Dress() #Create instance
shirt=Dress()
t_shirt.get_details()    #Call Method via Instance
shirt.get_details()      #Attributes value are constant
```

```
Cloth = Cotton
Type = T-Shirt
Cloth = Cotton
Type = T-Shirt
```

Parameterized Constructor in Python

1. A constructor accepts defined parameters or arguments along with self is called a parameterized constructor.
2. The first parameter to constructor is self that is a reference to the being constructed, and the rest of the arguments are provided by the programmer.
3. A parameterized constructor can have any number of arguments.
4. The values of attributes inside these constructors can be modified while instantiating with the help of parameters.

```
class Dress:
    def __init__(self, cloth, type, quantity): #Parameterized constructor
        self.cloth = cloth
        self.type = type
        self.quantity = quantity
    def get_details(self):
        print("Cloth =", self.cloth)
        print("Type =" , self.type)
shirt = Dress("silk", "shirt", 20)  #Create instance
t_shirt = Dress("cotton", "t-shirt",5)
shirt.get_details()         #Call Instance method
t_shirt.get_details()
```

```
Cloth = silk
Type = shirt
Cloth = cotton
Type = t-shirt
```

Default

- The default constructor is a simple constructor which doesn't accept any arguments.
- Its definition has only one argument which is a reference to the instance being constructed.
- It generates an empty constructor that has no code in it.

```
In [1]: class Myclass():
   ...:     # a method
   ...:     def show(self):
   ...:         print("i am in method")
   ...: # creating an object of the class
   ...: x=Myclass()
   ...: # calling the instance method using the object x
   ...: x.show()
i am in method

In [2]:
```

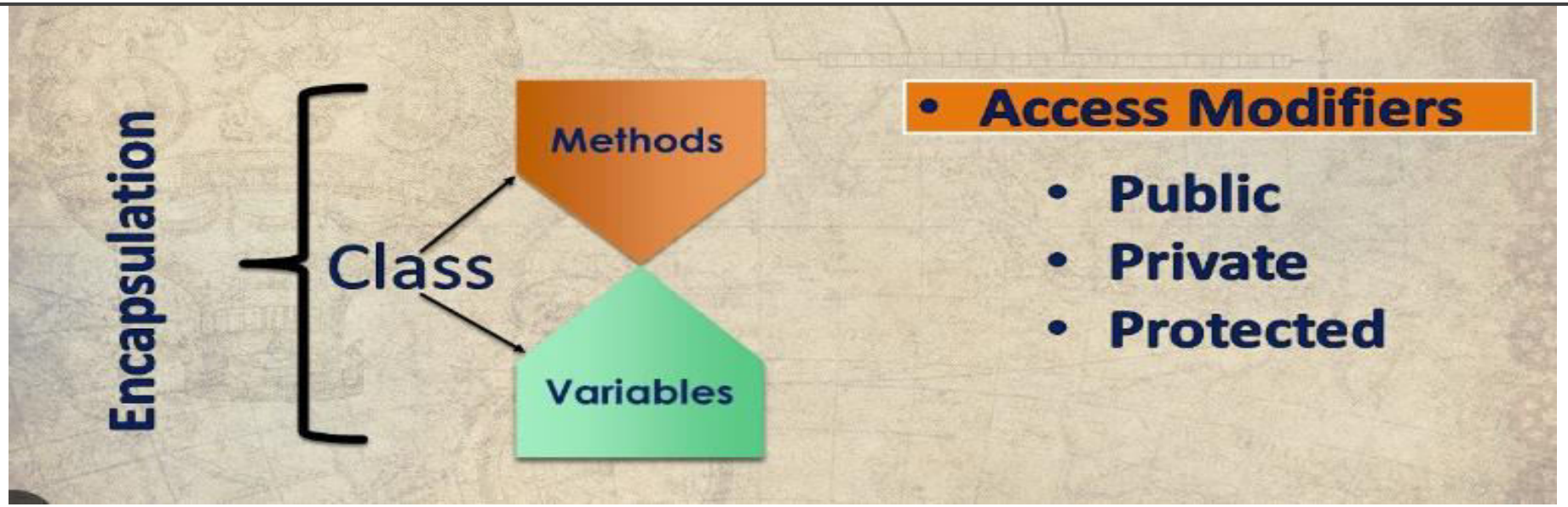
Constructor With Default Values

- Python allows us to define a constructor with default values.
- The default value will be used if we do not pass arguments to the constructor at the time of object creation.

```
class Student:
    # constructor with default values age and classroom
    def __init__(self, name, age=12, classroom=104):
        self.name = name
        self.age = age
        self.classroom = classroom

    # display Student
    def show(self):
        print(self.name, self.age, self.classroom)

# creating object of the Student class
dit = Student('Ashwini')
dit.show()

dit1 = Student('Nidhi', 13)
dit1.show()
```

```
Ashwini 12 104
Nidhi 13 104

In [4]:
```

Any Questions???



Data Encapsulation in Python

Mr. V. M. Vasava

GPG, IT Dept.

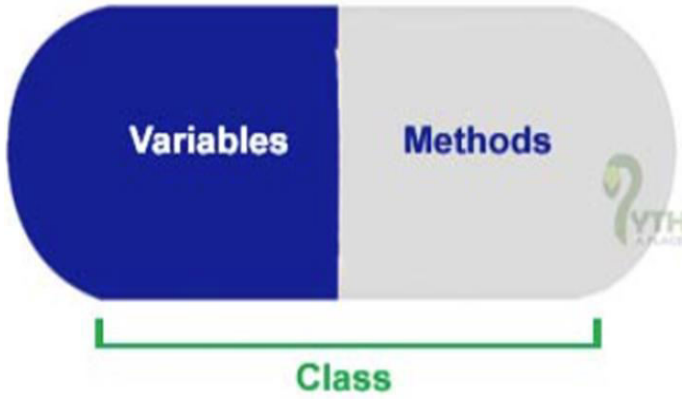
Agenda

Introduction about Data Encapsulation

Access Specifier in Python

Example

What is Data Encapsulation in Python?



- Encapsulation in python is the process of wrapping up variables and methods into a single entity.

1. __a ---> Private Members (Double Underscores)
2. _a --->Protected Members (Single Underscores)
3. Public Members

Need of Encapsulation

1. Encapsulation provides well-defined, readable code

2. Prevents Accidental Modification or Deletion

3. Encapsulation provides security

Access Modifiers in Python encapsulation

- Generally, we restrict **data access outside the class** in encapsulation.
- Encapsulation can be achieved by declaring the data members and method of class **as private**.
- Public Members
- Private Members
- **Protected** Members

Access Modifier in python

- Public member: accessible **anywhere** by using object reference.
- Private member: Accessible within the class. Accessible via methods only.
- Protected member: Accessible within class and it's subclasses.

Example

- Public member: accessible **anywhere** by using object reference.

Syntax :Object.Variables & Methods

```
# illustrating public members & public access modifier
class Employee:
    def __init__(self,name,salary):
        self.name=name
        self.salary=salary
    def display(self):
        # accessing public data member
        print(f"name is: {self.name} and salary is :{self.salary}")
e=Employee('vmv',1000)
# accessing public data member
print("Name: ", e.name)
# calling public member function of the class
e.display()
```

```
Name:   vmv
name is: vmv and salary is :1000

In [6]:
```

Private Member

- Private member: Accessible within the class.
  Accessible via methods only.
- The class members declared private should neither be accessed outside the class nor by any base class.
- In Python, there is no existence of Private instance variables that cannot be accessed except inside a class.
- Declare a private member prefix the member name with **double underscore** "__".

Example: Private Member

```
# illustrating private members access modifier
class Finance:
    def __init__(self):
        self.__revenue=10000  #private data
        self._no_of_sales=115 #protected data
f1=Finance()
class HR:
    def __init__(self):
        self.num_of_emp=12
        f1.revenue=12
        print(f1.__revenue)
h1=HR()
```

```
AttributeError: 'Finance' object has no
attribute '_HR__revenue'


In [12]:
```

Example

```
# illustrating private members & private access modifier
class Rectangle:
  __length = 0 #private variable
  __breadth = 0#private variable
  def __init__(self): #constructor
    self.__length = 5
    self.__breadth = 3
    #printing values of the private variable within the class
    print(self.__length,self.__breadth)
r = Rectangle() #object created
print(r.length) #printing values of the private variable outside the class
print(r.breadth)
```

```
5 3
Traceback (most recent call last):

  Cell In[1], line 10
    print(r.length) #printing values of the private variable
outside the class

AttributeError: 'Rectangle' object has no attribute 'length'
```

Example

```
class Student:
    __collegeName = 'GPG,Surat' # private class attribute

    def __init__(self, name, age):
        self.__name=name  # private instance attribute
        self.__age=age # private instance attribute
    def __display(self):  # private method
        #print('This is private method.')
        print(f"Name is: {self.__name} and Age is :{self.__age}")
class Dept(Student):
    def disp(self):  # public method
        print('This is Public method.')
d=Dept('Vmv',35)
d.disp()
d._Student__display()
print(d._Student__collegeName)
```

```
This is Public method.
Name is: Vmv and Age is :35
GPG,Surat

In [2]:
```

Protected

Class properties and methods with protected access modifier can be accessed within the class and from the class that inherits the protected class.

```
In [5]: class Person:
   ...:     def __init__(self, name, age):
   ...:         self._name=name  # protected attribute
   ...:         self._age=age # protected instance attribute
   ...: class Student(Person):   #Protected can access only derived class
   ...:     def __init__(self,name,age):
   ...:         super().__init__(name,age)
   ...:     def display_info(self):
   ...:         print(f"The person name is {self._name} and the age is {self._age}")
   ...: s=Student("Nidhi",20)
   ...: s.display_info()
The person name is Nidhi and the age is 20
```

Any Questions????