In [ ]:
```
#Creating a blank class

class cs:
    pass
print (cs)
```

In [ ]:
```
#Creating objects which belongs to class 'cs'
nikhil=cs()
sneha=cs()

print(nikhil)
```

In [ ]:
```
#Accessing members of class via objects

class student:
    name='ram'
    age=19

x=student()
print(x.name)
print(student.age)
print(student.name)
```

In [ ]:
```
#Python program to access class members using the class object

class cs:
    subject='advance python'
    def display(self):
        print("In class method....")

cs1=cs()
print(cs1.subject)
cs1.display()
```

CLASS VARIABLES AND OBJECT VARIABLES

A class can have variables defined in it. These variables are of two types:

1. CLASS VARIABLES : Class variables are owned by class
2. OBJECT VARIABLES : Object variables are owned by each object.

What does it means???

A. If a class has 'n' objects, then there will be 'n' seperate copies of the object variable as each object will have its own object variable. B. The object variable is not shared between objects. C. A change made to the object variable by one object will not be reflected in other objects. D. If a class has one class variable then there will be one copy only for that variable. All the objects of that class will share the class variable. E. Since, there exists a single copy of the class variable, any change made to the class variable by an object will be reflected in all other objects.

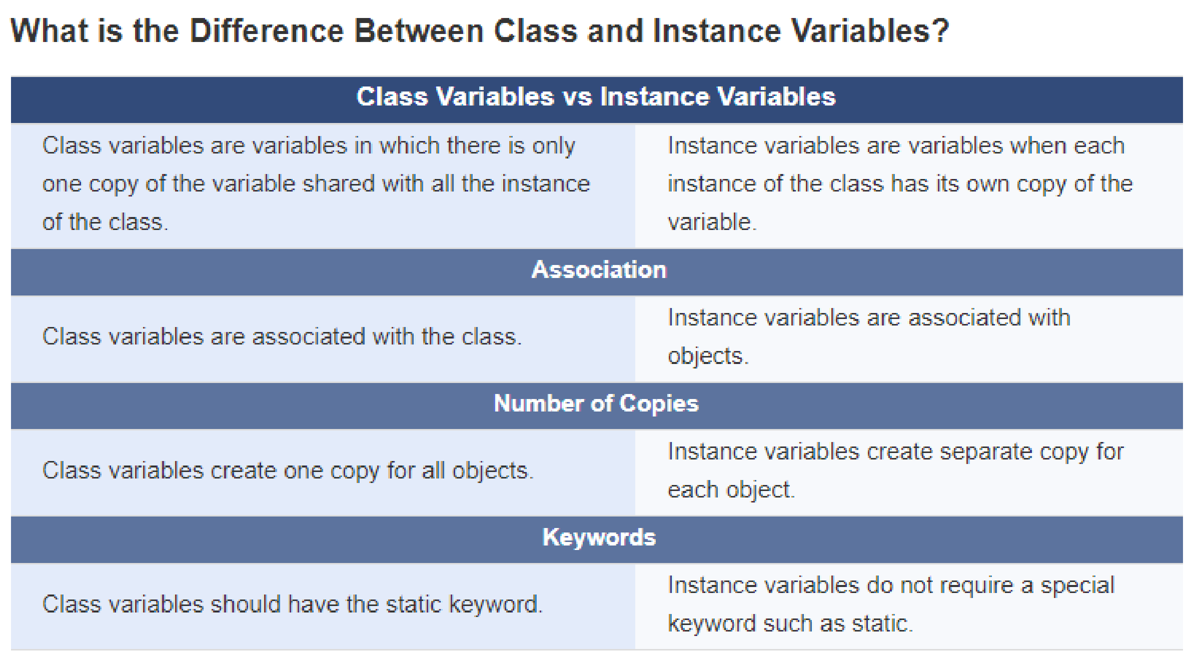
In [ ]:

```
#Program to differentiate between class and  object variables

class cs:
    class_var=0   #class variable
    def __init__(self,var):
        cs.class_var+=1
        self.var=var  #object variable
        print("The object value is : ",var)
        print("The value of class variable is: ",cs.class_var)

obj1=cs(10)
obj2=cs(20)
obj3=cs(30)
```

CLASS VARIABLES AND INSTANCE VARIABLES

- CLASS VARIABLES: Declared inside the class definition (but outside any of the instance methods). They are not tied to any particular object of the class, hence shared across all the objects of the class. Modifying a class variable affects all objects instance at the same time.
- INSTANCE VARIABLES: Declared inside the constructor method of class (the **init** method). They are tied to the particular object instance of the class, hence the contents of an instance variable are completely independent from one object instance to the other.

DIFFERENCE BETWEEN CLASS AND INSTANCE VARIABLES:



What is the Difference Between Class and Instance Variables?

Class Variables vs Instance Variables	
Class variables are variables in which there is only one copy of the variable shared with all the instance of the class.	Instance variables are variables when each instance of the class has its own copy of the variable.
Association	
Class variables are associated with the class.	Instance variables are associated with objects.
Number of Copies	
Class variables create one copy for all objects.	Instance variables create separate copy for each object.
Keywords	
Class variables should have the static keyword.	Instance variables do not require a special keyword such as static.

In [ ]:

```python
# Programme to understand class variables and instance variables:

class Employee:
    holiday=10
    pass

piyush=Employee()  #object creation
arjun=Employee()   #object creation

piyush.name='Piyush'
piyush.salary=10000
piyush.role='Trainer'

arjun.name='Arjun'
arjun.salary=500
arjun.role='Student'

print(arjun.name)
print(piyush.salary)
print(arjun.role)
print(arjun.__dict__)
print(arjun.holiday)
print(Employee.holiday)

Employee.holiday=20
print(Employee.holiday)
print(Employee.__dict__)
arjun.holiday=30
print(arjun.holiday)
print(arjun.__dict__)
print(Employee.holiday)
print(Employee.name)  #It will generate attribute error as there is no member in employ
```

CONSTRUCTOR IN PYTHON:

A constructor can simply be defined as a special type of method or function which can be used to initialize instances of various members in a class.

In Python, there are two different types of constructors.

1. Non-parameterized Constructor: The constructors in Python which have no parametres present is known as a non parameterized constructor.
2. Parameterized Constructor: A constructor that has a parametre pre defined is known as a parameterized constructor.

A constructor is defined the moment we create an object inside a class. The presence of a constructor also verifies that enough resources are present, so that a start up task can easily be executed via an object of a class.

SELF PARAMETER:

1. The self argument refers to the object itself.
2. The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the object.
3. It does not have to be named self , any valid identifier can be used, but it has to be the first parameter of any instance method in the class.
4. A method can have any number of parameters, but the first parameter will always be a variable called self.

It is used in method definitions and in variable initialization. The self method is explicitly used every time we define a method.

USE OF SELF

The self is used to represent the instance of the class. With this word, you can access the attributes and methods of the class in python. It binds the attributes with the given arguments. The reason why we use self is that Python does not use the '@' syntax to refer to instance attributes.In Python, we have methods that make the instance to be passed automatically, but not received automatically.

In [ ]:
```
#PROGRAM TO DEMONSTRATE USE OF SELF:

class food:
    def __init__(self, fruit, color):   # init method or constructor
        self.fruit = fruit
        self.color = color

    def show(self):
        print("fruit is", self.fruit)
        print("color is", self.color )
```

```
apple = food("apple", "red")
grapes = food("grapes", "green")

apple.show()
grapes.show()
```

In [ ]:

```
#PYTHON CLASS SELF CONSTRUCTOR: self is also used to refer to a variable field within t

class person:
    def __init__(self,n):
        self.name=n

    def get_person_name(self):
        print(self.name)

p1=person("abc")
p1.get_person_name()
```

NOTE: In the above example, self refers to the name variable of the entire Person class. Here, if we have a variable within a method, self will not work. That variable is simply existent only while that method is running and hence, is local to that method. For defining global fields or the variables of the complete class, we need to define them outside the class methods.

In [ ]:

```
#SELF is not a keyword,you can any other word instead of it

class this_is_class:
    def show(in_place_of_self):
        print("It is not a keyword ""and you can use a different keyword")


object = this_is_class()
object.show()
```

CREATING A METHOD OF CLASS

METHOD: A Python method is like a Python function, but it must be called on an object and to create it, you must put it inside a class.So, a function created/defined inside a class is known as "method".These methods are the reusable piece of code that can be invoked/called at any point in the program.

SYNTAX: class class_name: def method_name(self[,arg1,arg2,arg3....]): statement 1 statement 2 . . . statement n

In [ ]:

```
#PROGRAM TO CALCULATE AREA OF RECTANGLE USING METHOD:

class Rectangle:
    def __init__(self,l,b):
        self.length=l
        self.breadth=b
    def area(self):
        print("Area of rectangle is:",self.length*self.breadth)

l=int(input("Enter the length of rectangle: "))
b=int(input("Enter the breadth of rectangle: "))
area=Rectangle(l,b)
area.area()
```

GETTER'S AND SETTER'S METHODS:

GETTER METHODS: Getters are the methods that are used in Object-Oriented Programming (OOPS) to access a class's private attributes. A getter is a method that gets the value of a property.

SETTER'S METHODS: The setter is a method that is used to set the property's value. It is very useful in object-oriented programming to set the value of private attributes in a class.

In [ ]:

```
class student:
    def __init__(self,age=0):
        self.age=age
    #using the getter method

    def get_age(self):
        return self.age
    #using setter method

    def set_age(self,a):
        self.age=a


#using the setter function
sneha=student()

sneha.set_age(20)

print(sneha.get_age())
```

In [ ]:

```
#CONSTRUCTOR AND SELF

class Test:


    def __init__(self):
        self.name="Mohit"
        self.age=29

    def update(self):
        self.age=30

    def compare(self,other):
        if self.age==other.age:
            return True
        else:
            return False



t1=Test()
t2=Test()
print (id(t1))  #object 1 having different memory location/address  NOTE: Size of memor
                                                       #             and size of
print (id(t2))  #object 2 having different memory location/address  #Que: Who allocates
                                                                    #Ans: Constructor


print (t1.name)
print(t2.age)

t1.name='puneet'
t2.age=99
print(t1.name)
print(t2.age)

t1.update()
print(t1.age)

if(t1.compare(t2)):
    print("They are same")
else:
    print("They are different")
```

TYPES OF METHODS:

PYTHON PROVIDES US THREE TYPES OF METHODS:

1. INSTANCE METHODS
2. CLASS METHODS
3. STATIC METHODS

INSTANCE METHOD:

The purpose of instance methods is to set or get details about instances (objects), and that is why they're known as instance methods. They are the most common type of methods used in a Python class.

They have one default parameter- self, which points to an instance of the class.

In [ ]:

```
#EXAMPLE OF INSTANCE METHOD:

class Computer:

    def instance_method(self):
        return "This is an instance method"
```

```
obj1=Computer()

print(obj1.instance_method())
```

In [ ]:

```
#Instance method with some parameter

class Computer:

    def instance_method(self,a):
        return f"This is an instance method with parameter a= {a}."

obj1=Computer()
print(obj1.instance_method(12))
```

DECORATORS IN PYTHON:

Python has an interesting feature called decorators to add functionality to an existing code. A Python decoratoris a function that takes in a function,adds some functionality to it and returns the original function.

Prerequisites for learning decorators:

1. A function is an instance of object type
2. Function can be stored in a variable
3. Function can be passed as a parameter to another function.
4. Function can return function from a function.
5. Function can be stored in different data structures like list etc.

In [ ]:

```
#Program to demonstrate function as an argument

def inc(x):
    return x+1

def operate(fun,x):  #fun is function as an argument i.e, 'inc' function.
    result=fun(x)
    return result


print(operate(inc,5))
```

In [ ]:

```
#Program to demonstrate function inside a function

def greetings(message):
    x="Hello"

    def display():   #inner function to greetings/nested function
        print(x,message)

    display()
```

```
print(greetings("Good morning class"))
```

In [ ]:

```
#Program to demonstrate function returning a function

def greetings(message):
    x="Hello"

    def display():
        print(x,message)

    return display   #function returning a function

func=greetings("Good morning class.")
func()
```

In [ ]:

```
#Program to demonstrate decorator

def deco_hello(fun):   #fun as an argument which means function as an object
    def inner():
        c=fun()
        return c + "How are you class?"
    return inner  #for returning inner function

@deco_hello
def hello():
    return "Helloo!!!"


#hello=deco_hello(hello)
print(hello())
```

In [ ]:

```
#Decorator function with two arguments:
#Decorator function to decorate addition to give factorial.

def deco_add(fun):
    def factorial(x,b):
        c=fun(x,b)
        fact=1
        while c>0:
            fact = fact * c
            c=c-1
        return fact
    return factorial
@deco_add
def add(a,y):
    return a+y
print(add(3,2))
```

In [ ]:

```
#Decorator function with two arguments:
#Decorator function to decorate addition to give square of sum.

def deco_add(fun):
    def square(a,b):
        c=fun(a,b)
        return c*c
    return square

@deco_add

def add(x,y):
    return x+y

#add =deco_add(add)
print(add(7,6))
```

CHAINING DECORATORS:

Multiple decorators can be chained in Python.

This is to say, a function can be decorated multiple times with different (or same) decorators. We simply place the decorators above the desired function.

SYNTAX:

@function1 @function2 def function(name): print(f"{name}")

In [ ]:

```
#Program to demonstrate chaining of decorators.

def deco2(fun):
    def inner(x,y):
        x=fun(x,y)
        return x*x
    return inner

def deco1(fun):
    def inner(x,y):
        x=fun(x,y)
        return 2*x
    return inner

@deco1
@deco2

def number(x,y):
    return x+y

print(number(4,2))
```

In [ ]:

```
help(classmethod)
```

CLASS METHOD:

Class methods are methods that are called on the class itself, not on a specific object instance. Therefore, it belongs to a class level, and all class instances share a class method.

1. A class method is bound to the class and not the object of the class. It can access only class variables.
2. It can modify the class state by changing the value of a class variable that would apply across all the class objects.

In method implementation, if we use only class variables, we should declare such methods as class methods. The class method has a cls as the first parameter, which refers to the class.

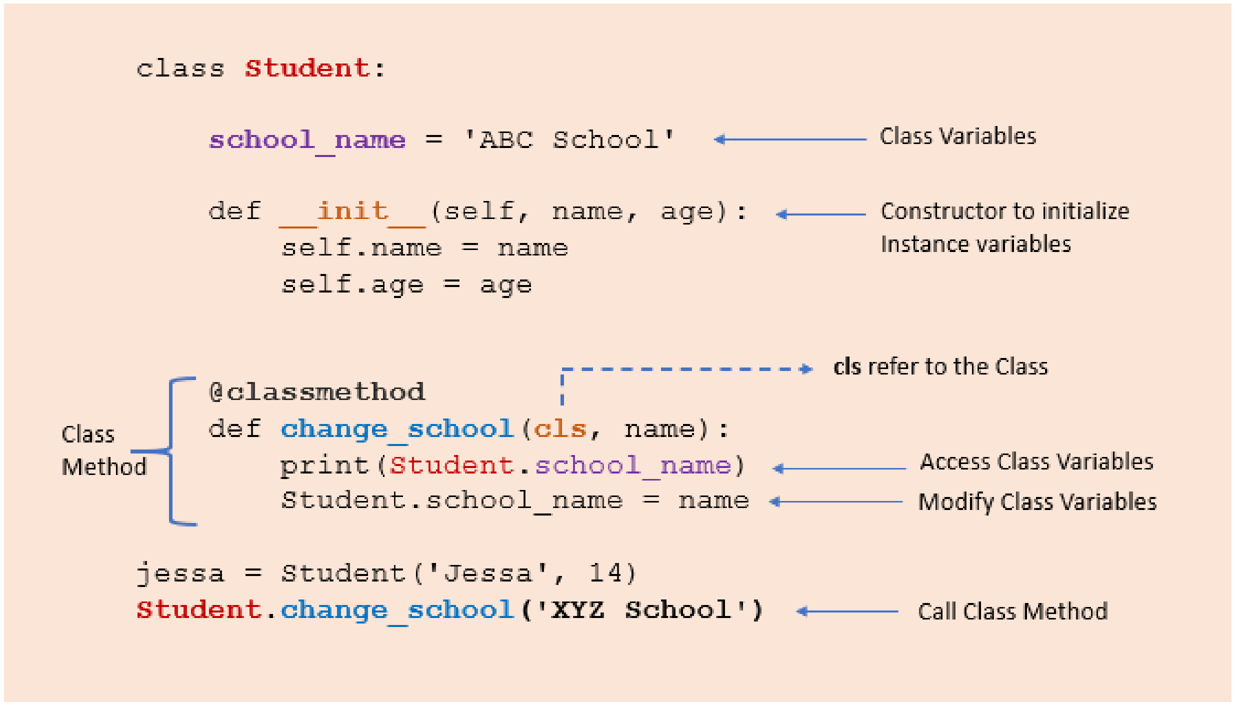
Class methods are used when we are dealing with factory methods. Factory methods are those methods that return a class object for different use cases. Thus, factory methods create concrete implementations of a common interface.

The class method can be called using: ClassName.method_name() as well as by using an object of the class.

The purpose of the class methods is to set or get the details (status) of the class. That is why they are known as class methods. They can't access or modify specific instance data. They are bound to the class instead of their objects.

Two important things about class methods:

1. In order to define a class method, you have to specify that it is a class method with the help of the @classmethod decorator.
2. Class methods also take one default parameter- cls, which points to the class. Again, this not mandatory to name the default parameter “cls”. but it is always better to go with the conventions.



```
class Student:

    school_name = 'ABC School'

    def __init__(self, name, age):
        self.name = name
        self.age = age


    @classmethod
    def change_school(cls, name):
        print(Student.school_name)
        Student.school_name = name

jessa = Student('Jessa', 14)
Student.change_school('XYZ School')
```

DEFINING A CLASS METHOD:

Any method we create in a class will automatically be created as an instance method. We must explicitly tell Python that it is a class method using the @classmethod decorator or classmethod() function.

Class methods are defined inside a class, and it is pretty similar to defining a regular function.

Like, inside an instance method, we use the self keyword to access or modify the instance variables. Same inside the class method, we use the cls keyword as a first parameter to access class variables. Therefore the class method gives us control of changing the class state.The class method can only access the class attributes, not the instance attributes

In [ ]:
```
#Program to demonstrate class method using @classmethod decorator

from datetime import date

class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    @classmethod
    def calculate_age(cls, name, birth_year):
        # calculate age an set it as a age
        # return new object
        return cls(name, date.today().year - birth_year)

    def show(self):
        print(self.name + "'s age is: " + str(self.age))

obj1 = Student('Mohit', 29)
obj1.show()

# create new object using the factory method
obj2 = Student.calculate_age("Deepu", 1994)
obj2.show()
```

Creating class method using classmethod() function:

Apart from a decorator, the built-in function classmethod() is used to convert a normal method into a class method. The classmethod() is an inbuilt function in Python, which returns a class method for a given function.

SYNTAX: classmethod(function)

1. It is the name of the method you want to convert in class method
2. It returns the converted class method.

POINTS TO REMEMBER:

1. The method you want to convert as a class method must accept class (cls) as the first argument, just like an instance method receives the instance (self).
2. The class method is bound to class rather than an object. So we can call the class method both by calling class and object.
3. A classmethod() function is the older way to create the class method in Python. In a newer version of Python, we should use the @classmethod decorator to create a class method.

In [ ]:
```
#Program to demonstrate class method using CLASSMETHOD() function

class NIET:
    name='Computer Science'  #class variable


    def branch_name(cls):
        print("Branch Name is: ",cls.name)


NIET.branch_name=classmethod(NIET.branch_name)  # Creating class method
NIET.branch_name()      #Call of class method
```

Access Class Variables in Class Methods:

Using the class method, we can only access or modify the class variables. Let’s see how to access and modify the class variables in the class method.

Class variables are shared by all instances of a class. Using the class method we can modify the class state by changing the value of a class variable that would apply across all the class objects.

In [ ]:
```
#Program to demonstrate the accessing class variable in class method

class NIET:
    branch_name='Computer Science'

    def __init__(self,name,age):
        self.name=name
        self.age=age

    @classmethod
    def change_branch(cls,branch_name):
        NIET.branch_name=branch_name   #classname.classvariable

    def display(self):                 #Instance method
        print(self.name,self.age,'NIET:',NIET.branch_name)

student=NIET('Pranav',18)
student.display()

NIET.change_branch('AIML')
student.display()
```

DYNAMICALLY ADD CLASS METHOD TO CLASS:

Typically, we add class methods to a class body when defining a class. However, Python is a dynamic language that allows us to add or delete methods at runtime. Therefore, it is helpful when you wanted to extend the class functionality without changing its basic structure because many systems use the same structure.

We need to use the classmethod() function to add a new class method to a class.

In [ ]:

```
class NIET:
    branch_name='Computer Science'

    def __init__(self,name,age):
        self.name=name
        self.age=age

    def display(self):
        print(self.name,self.age)



def questions(cls):   #method define outside the class
    print("Below assignment is for Computer Science: ",cls.branch_name)

NIET.questions=classmethod(questions)#Adding class method at runtime to class

student=NIET("Prachi",19)
student.display()

NIET.questions()  #Call of the new method
```

In [ ]:

```
# Instantiation in class method
class Rectangle:
    def __init__(self, length, breadth):
        self.length = length
        self.breadth = breadth
    def __area(self):
        return self.length * self.breadth
    def display(self):
        print("length:",self.length)
        print("breadth:",self.breadth)
        print("Area=",self.__area())

r1=Rectangle(5,10)
#print(r1.display())
r1.display()
```

In [ ]:

```
class Rectangle:
    count=0
    def __init__(self, length, breadth):
        self.length = length
        self.breadth = breadth
        Rectangle.count+=1
    def __area(self):
        return self.length * self.breadth
    def display(self):
        print("length:",self.length)
        print("breadth:",self.breadth)
        print("Area=",self.__area())
    def Square(cls,side):
        s=cls(side, side)
        print("Total rectangle created=",cls.count)
        return s

Rectangle.Square = classmethod(Rectangle.Square)
s1=Rectangle(10,20)
s1.display()
s2 = Rectangle.Square(10)
s2.display()
s3 = s1.Square(20)
s3.display()
```

In [ ]:

```
class Rectangle:
    count=0
    def __init__(self, length, breadth):
        self.length = length
        self.breadth = breadth
        Rectangle.count+=1
    def __area(self):
        return self.length * self.breadth
    def display(self):
        print("length:",self.length)
        print("breadth:",self.breadth)
        print("Area=",self.__area())
    @classmethod
    def Square(cls,side):
        s=cls(side, side)
        print("Total rectangle created=",cls.count)
        return s
s1=Rectangle(10,20)
s1.display()
s2 = Rectangle.Square(10)
s2.display()
s3 = s1.Square(20)
s3.display()
```

In [ ]:

```
class Emp:
    count=0
    total_sal=0
    def __init__(self):
        self.name=input("Enter name of Emp:")
        self.des=input("Enter des of Emp:")
        self.salary=int(input("Enter the salary"))
        Emp.count+=1
        Emp.total_sal+=self.salary
    def show(self):
        print("Name of emp:",self.name)
        print("Post of emp:",self.des)
        print("Salary of emp:",self.salary)
    @classmethod
    def average(cls):
        s=cls.total_sal/cls.count
        print("Average salary of the company=%.2f"%(s))
e1=Emp()
e2=Emp()
e3=Emp()
e1.average()
e2.average()
e3.average()
Emp.average()
```

In [ ]:

```
help(staticmethod)
```

STATIC METHODS IN PYTHON:

1. A static method is a general utility method that performs a task in isolation.
2. A static method is bound to the class and not the object of the class. Therefore, we can call it using the class name.
3. A static method doesn't have access to the class and instance variables because it does not receive an implicit first argument like self and cls. Therefore it cannot modify the state of the object or class.

1. Static methods, much like class methods, are methods that are bound to a class rather than its object.

2. They do not require a class instance creation. So, they are not dependent on the state of the object.

The difference between a static method and a class method is:

1. Static method knows nothing about the class and just deals with the parameters.
2. Class method works with the class since its parameter is always the class itself.
3. They can be called both by the class and its object.

In [1]:

```
#Program to demonstrate static method

class Employee:
    @staticmethod
    def sample(x):
        print('Inside static method', x)

# call static method
Employee.sample(10)

# can be called using object
emp = Employee()
emp.sample(10)
```

```
Inside static method 10
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_17120/1309734421.py in <module>
     11 # can be called using object
     12 #emp = Employee()
---> 13 emp.sample(10)

NameError: name 'emp' is not defined
```

In [3]:

```
# Understanding Static Method:
class A:
    x=10
    def __init__(self,x):
        self.x=x
    def display(x):
        return x
a=A(5)
print(a.x)
print(A.x)
print(A.display(20))
print(a.display())
```

```
5
10
20
<__main__.A object at 0x000001EECF7A9700>
```

DEFINING STATIC METHOD IN PYTHON:

Any method we create in a class will automatically be created as an instance method. We must explicitly tell Python that it is a static method using the @staticmethod decorator or staticmethod() function.

Static methods are defined inside a class, and it is pretty similar to defining a regular function.

In [6]:

```
# Understanding Static Method:
class A:
    x=10
    def __init__(self,x):
        self.x=x
    @staticmethod
    def display(x):
        return x
a=A(5)
print(a.x)
print(A.x)
print(A.display(20))
print(a.display(30))
```

```
5
10
20
30
```

In [2]:

```
# Example of static method
# Allotment of an elective subject after validating the available choices:
class Choice:
    def __init__(self,n,r,subject):
        self.name=n
        self.roll=r
        self.subject=subject
    def display(self):
        print("------Student Information------")
        print("--> Name of Student:",self.name)
        print("--> RollNo. of Student:",self.roll)
        print("--> Subject Allotted :",self.subject)
    @staticmethod
    def validate_subject(subjects,x):
        if x in subjects:
            print("Option is available")
            return True
        else:
            print("Option is not available")
            return False
subjects=["DS", "CSA", "FOC", "OS", "TOC"]
sub=input("Enter the subject of your choice:")
if Choice.validate_subject(subjects,5):
    name=input("Enter your Name:")
    roll=int(input("Enter your RollNo:"))
    c = Choice(name,roll,sub)
    c.display()
```

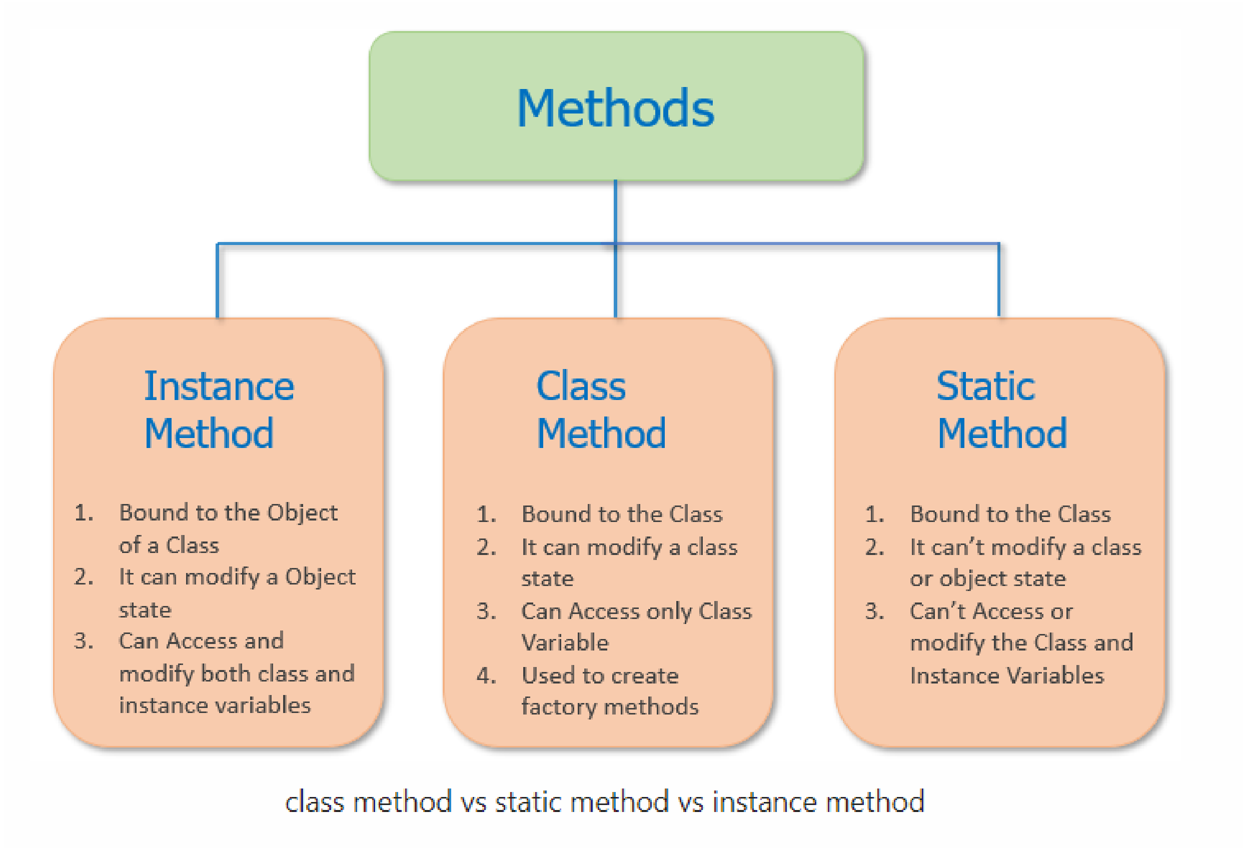
```
Enter the subject of your choice:DS
Option is available
Enter your Name:xyz
Enter your RollNo:123
------Student Information------
--> Name of Student: xyz
--> RollNo. of Student: 123
--> Subject Allotted : DS
```

CLASS METHOD VS STATIC METHOD VS INSTANCE METHOD

1. Instance method performs a set of actions on the data/value provided by the instance variables. If we use instance variables inside a method, such methods are called instance methods.

2. Class method is method that is called on the class itself, not on a specific object instance. Therefore, it belongs to a class level, and all class instances share a class method.

3. Static method is a general utility method that performs a task in isolation. This method doesn’t have access to the instance and class variable.


Methods
Instance Method
1. Bound to the Object of a Class
2. It can modify a Object state
3. Can Access and modify both class and instance variables
Class Method
1. Bound to the Class
2. It can modify a class state
3. Can Access only Class Variable
4. Used to create factory methods
Static Method
1. Bound to the Class
2. It can't modify a class or object state
3. Can't Access or modify the Class and Instance Variables
class method vs static method vs instance method

In [6]:

```
#Meter class having attribute 'length' and meter to cm conversion method.

class Meter:
    def __init__(self,m=0):
        self.length=m
    def mtoc(self):
        return self.length*100

m1=Meter()
#print(m1.length)
m1.length=20
print("In meter: ",m1.length)
print("In centimeter: ",m1.mtoc())
```

```
In meter:  20
In centimeter:  2000
```

In [9]:

```
#We can use length as private method and getter and setter methods.
#Here if length is a negative attribute then it should raise an error.

class Meter:
    def __init__(self,m=0):
        self.set_length(m)

    def mtoc(self):
        return self.get_length()*100

    def get_length(self):
        print("-----GEtting Length-----")
        return self._length
    def set_length(self,val):
        print("-----Setting Value-----")
        if val<0:
            raise ValueError("Length can not be less than 0.")
        self._length=val

m1=Meter(20)
print("In meter= ",m1.get_length())
print("In centimeter= ",m1.mtoc())
m1.set_length(-5)
print("In meter= ",m1.get_length())
```

```
-----Setting Value-----
-----GEtting Length-----
In meter=  20
-----GEtting Length-----
In centimeter=  2000
-----Setting Value-----
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_6120/2248502010.py in <module>
     21 print("In meter= ",m1.get_length())
     22 print("In centimeter= ",m1.mtoc())
---> 23 m1.set_length(-5)
     24 print("In meter= ",m1.get_length())

~\AppData\Local\Temp/ipykernel_6120/2248502010.py in set_length(self, val)
     15         print("-----Setting Value-----")
     16         if val<0:
---> 17             raise ValueError("Length can not be less than 0.")
     18         self._length=val
     19 

ValueError: Length can not be less than 0.
```

In [10]:

```
help(property)
```

```
Help on class property in module builtins:

class property(object)
 |  property(fget=None, fset=None, fdel=None, doc=None)
 |  
 |  Property attribute.
 |  
 |    fget
 |      function to be used for getting an attribute value
 |    fset
 |      function to be used for setting an attribute value
 |    fdel
 |      function to be used for del'ing an attribute
 |    doc
 |      docstring
 |  
 |  Typical use is to define a managed attribute x:
 |  
 |  class C(object):
 |      def getx(self): return self._x
 |      def setx(self, value): self._x = value
 |      def delx(self): del self._x
 |      x = property(getx, setx, delx, "I'm the 'x' property.")
 |  
 |  Decorators make defining new properties or modifying existing ones easy:
 |  
 |  class C(object):
 |      @property
 |      def x(self):
 |          "I am the 'x' property."
 |          return self._x
 |      @x.setter
 |      def x(self, value):
 |          self._x = value
 |      @x.deleter
 |      def x(self):
 |          del self._x
 |  
 |  Methods defined here:
 |  
 |  __delete__(self, instance, /)
 |      Delete an attribute of instance.
 |  
 |  __get__(self, instance, owner, /)
 |      Return an attribute of instance, which is of type owner.
 |  
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |  
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |  
 |  __set__(self, instance, value, /)
 |      Set an attribute of instance to value.
 |  
 |  deleter(...)
 |      Descriptor to change the deleter on a property.
 |  
 |  getter(...)
 |      Descriptor to change the getter on a property.
 |  
 |  setter(...)
 |      Descriptor to change the setter on a property.
 |  
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |  
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |  
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |  
 |  __isabstractmethod__
 |  
 |  fdel
 |  
 |  fget
 |  
 |  fset
```

In [17]:

```
#Example meter class having length attributes and meter to cm conversion method
#Using property class

class Meter:
    def __init__(self,m=0):
        self.length=m

    def mtoc(self):
        return self.length*100

    def get_length(self):
        print('----Getting Value----')
        return self._length
    def set_length(self,val):
        print("----Setting Value----")
        if val<0:
            raise ValueError("Length can not be less than 0")
        self._length=val

#Creating a property object
    length=property(get_length,set_length)
m1=Meter(50)
print("In meter: ",m1.length)
print("In centimeter: ",m1.mtoc())
m1.length=-87
print("In meter: ",m1.length)
```

```
----Setting Value----
----Getting Value----
In meter:  50
----Getting Value----
In centimeter:  5000
----Setting Value----

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_6120/798403761.py in <module>
     23 print("In meter: ",m1.length)
     24 print("In centimeter: ",m1.mtoc())
---> 25 m1.length=-87
     26 print("In meter: ",m1.length)

~\AppData\Local\Temp/ipykernel_6120/798403761.py in set_length(self, val)
     15         print("----Setting Value----")
     16         if val<0:
---> 17             raise ValueError("Length can not be less than 0")
     18         self._length=val
     19 

ValueError: Length can not be less than 0
```

In [18]:

```
#Using @property decorator

class Meter:
    def __init__(self,m=0):
        self.length=m

    @property
    def length(self):
        print('----Getting Value----')
        return self._length
    @length.setter
    def length(self,val):
        print('----Setting Value----')
        if val<0:
            raise ValueError("Length can not be 0")
        self._length=val

    def mtoc(self):
        return self.length*100

m1=Meter(69)
print("In meter: ",m1.length)
print("In centimeter: ",m1.mtoc())
m1.length=-98
print("In meter: ",m1.length)
```

```
----Setting Value----
----Getting Value----
In meter:  69
----Getting Value----
In centimeter:  6900
----Setting Value----
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_6120/1358775413.py in <module>
     22 print("In meter: ",m1.length)
     23 print("In centimeter: ",m1.mtoc())
---> 24 m1.length=-98
     25 print("In meter: ",m1.length)

~\AppData\Local\Temp/ipykernel_6120/1358775413.py in length(self, val)
     13         print('----Setting Value----')
     14         if val<0:
---> 15             raise ValueError("Length can not be 0")
     16         self._length=val
     17 

ValueError: Length can not be 0
```

MAGIC METHOD IN PYTHON:

1. Magic methods in Python are the special methods that start and end with the double underscores.
2. They are also called dunder methods.
3. Magic methods are not meant to be invoked directly by you, but the invocation happens internally from the class on a certain action.

- Example 1 : For example when you add two numbers using '+' operator, internally , the **add**() method will be called.

- Example 2 : For example, when you create an object of a class, internally, the **init**() method will be called.

- Built-in classes in Python define magic methods.
- Use the dir() function to see the magic methods inherited by class.

In [19]:

```
dir(int)
```

Out[19]:

```
['__abs__',
 '__add__',
 '__and__',
 '__bool__',
 '__ceil__',
 '__class__',
 '__delattr__',
 '__dir__',
 '__divmod__',
 '__doc__',
 '__eq__',
 '__float__',
 '__floor__',
 '__floordiv__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getnewargs__',
 '__gt__',
 '__hash__',
 '__index__',
 '__init__',
 '__init_subclass__',
 '__int__',
 '__invert__',
 '__le__',
 '__lshift__',
 '__lt__',
 '__mod__',
 '__mul__',
 '__ne__',
 '__neg__',
 '__new__',
 '__or__',
 '__pos__',
 '__pow__',
 '__radd__',
 '__rand__',
 '__rdivmod__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__rfloordiv__',
 '__rlshift__',
 '__rmod__',
 '__rmul__',
 '__ror__',
 '__round__',
 '__rpow__',
 '__rrshift__',
 '__rshift__',
 '__rsub__',
 '__rtruediv__',
 '__rxor__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__sub__',
 '__subclasshook__',
 '__truediv__',
 '__trunc__',
 '__xor__',
 'as_integer_ratio',
 'bit_length',
 'conjugate',
 'denominator',
 'from_bytes',
 'imag',
 'numerator',
 'real',
 'to_bytes']
```

In [25]:

```
#add magic method:
# The add is a magic method which gets called when we add two numbers using '+' operato

num=10
print(num+5)
print(num.__add__(5))


a=5
b=6
print(a+b)
print(int.__add__(a,b))
```

```
15
15
11
11
```

In [26]:

```
print(object)
```

```
<class 'object'>
```

In [28]:

```
#new() method
# In python the __new__() magic method is implicitly called before the __init__() metho
#The __new__() method returns a new object, which is then initialized by __init__().

class A:
    def __new__(cls):
        print("The new magic method is called")
        inst=object.__new__(cls)
        return inst

    def __init__(self):
        print("__init__ magic method is called")
        self.name='python'

a=A()
```

```
print(a)
print(a.name)
```

```
The new magic method is called
__init__ magic method is called
<__main__.A object at 0x000002935A1C7CD0>
python
```

- **str**() Method

- Another useful magic method is **str**().

- It is overridden to return a printable string representation of any user defined class.
- We have seen str() built-in function which returns a string from the object parameter.
- For example, str(12) returns '12'. When invoked, it calls the **str**() method in the int class.

In [30]:

```
num=12
x=str(num)

#This is equivalent to:
y=int.__str__(num)
print(type(num))
print(type(x))
print(type(y))
```

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

- **str**() method returns an informal string representation of its object.
- Another magic method **repr**() returns the formal string representation of the object
- When **repr**() method is implemented only, then in print() **repr**() method is invoked automatically.
- **repr** gets invoked when we call repr() function.
- When both **str**() and **repr**() are implemented, the print() function invoke the **str**() method.

In [31]:

```
s='Hello Python'
print(str(s))
print(repr(s))
```

```
Hello Python
'Hello Python'
```

In [32]:

```
import datetime
today=datetime.datetime.now()

#Print readable format for date-time object
print(str(today))

#Print the official format for date-time object
print(repr(today))
```

```
2022-04-22 10:18:27.882776
```

```
datetime.datetime(2022, 4, 22, 10, 18, 27, 882776)
```

In [33]:

```
class Employee:
    def __init__(self):
        self.name='ABC'
        self.salary=10000

    def __str__(self):
        return 'Name= ' + self.name +',Salary= '+str(self.salary)
    def __repr__(self):
        return 'Hello= ' + self.name

e1=Employee()
print(e1)
print(repr(e1))
```

```
Name= ABC,Salary= 10000
Hello= ABC
```

In [ ]:

INHERITANCE IN CLASS METHODS:

Inheritance is a powerful feature in object oriented programming.

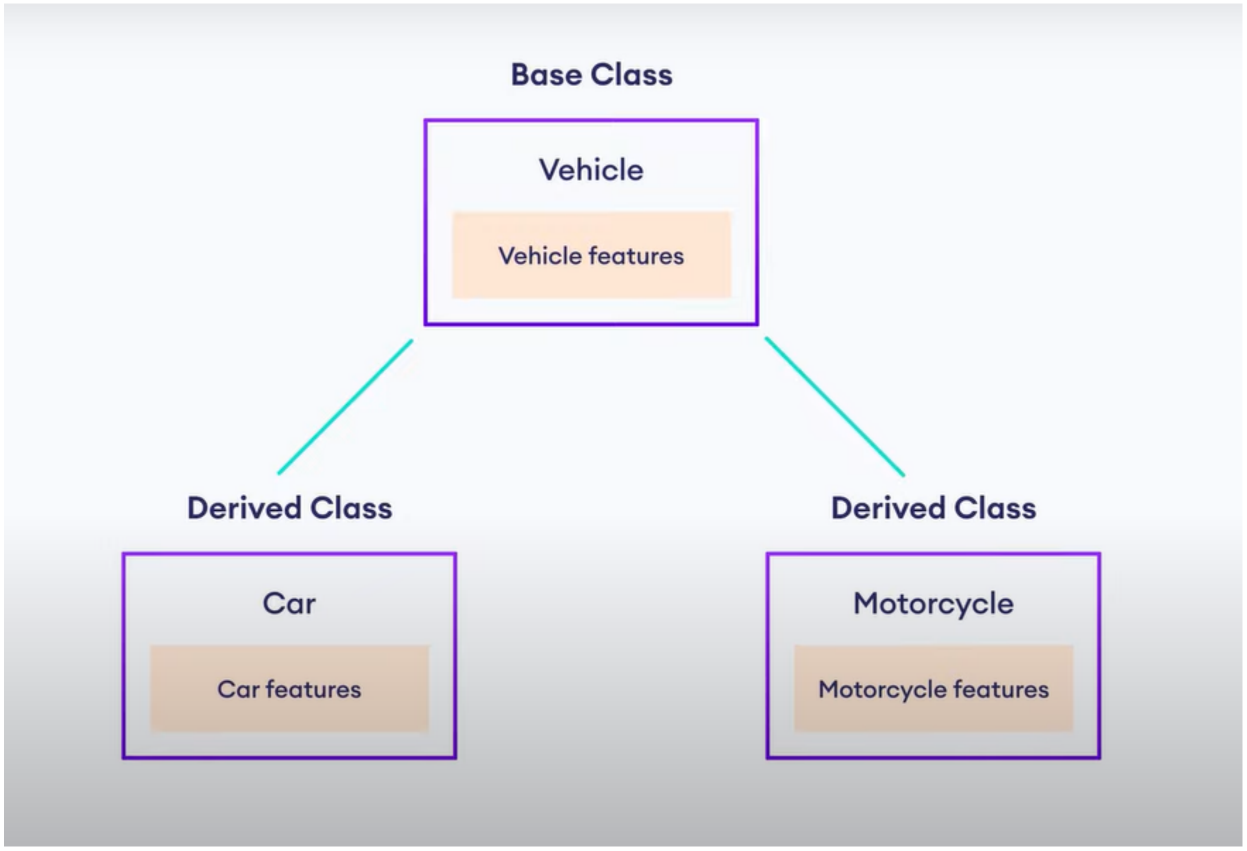
It refers to defining a new class with little or no modification to an existing class. The new class is called derived (or child) class and the one from which it inherits is called the base (or parent) class.

BENEFITS OF INHERITANCE:

1. Inheritance depicts relationships that resemble real-world scenarios.
2. It provides the feature of re-usability which allows the user to add more features to the derived class without altering it.
3. If a class Y inherits from class X, then automatically all the sub-classes of Y would inherit from class X.

Python Inheritance Syntax

```
class SuperClassName:
  Body of Super class


class DerivedClass_Name(SuperClass):
  Body of derived class
```



In [ ]:

```
# Program to demonstrate Inheritance in class

class Animal:
    def legs(self):
        print("All have four legs")

class Dog(Animal):
    def sound(self):
        print("Bow Bow")

class Cat(Animal):
    def sound(self):
        print("Meow Meow")
```

```
labra=Dog()
labra.sound()
labra.legs()

scottish_fold=Cat()
```

In [ ]: