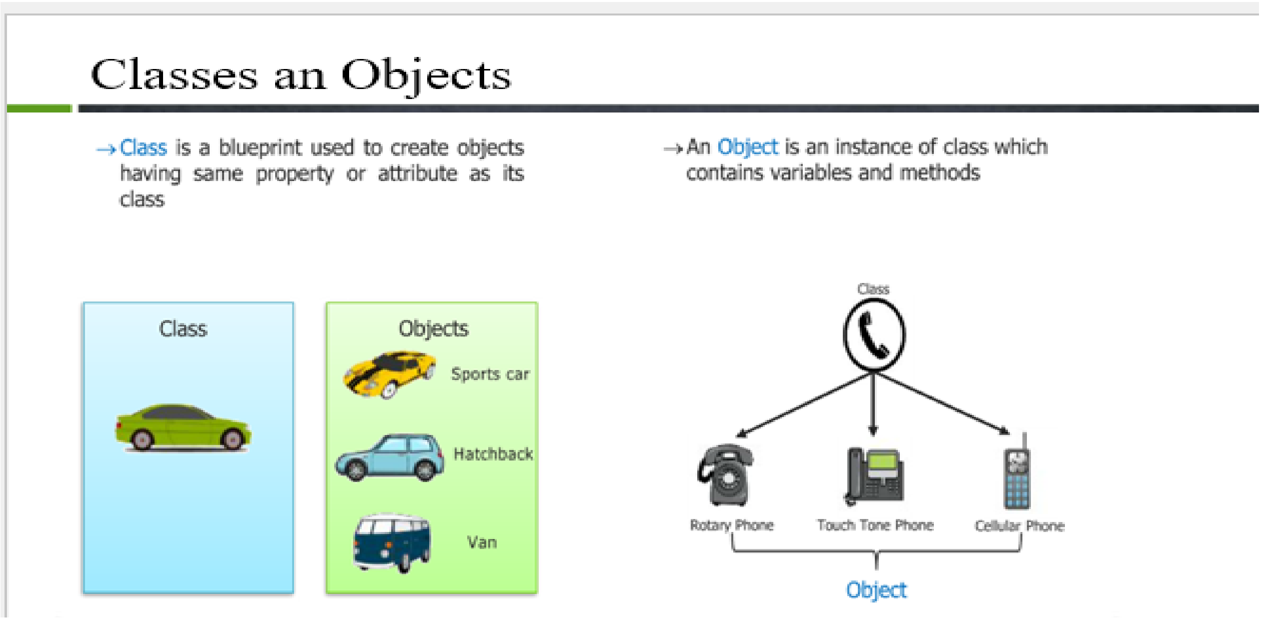
Classes & objects (OOPs)

- Python is a Powerful language that supports the object-oriented programming paradigm.
- A programming technique that requires the use of objects and classes is known as OOP.
- Object-Oriented Programming is based on the principle of writing reusable code that the user can access multiple times.



Classes an Objects

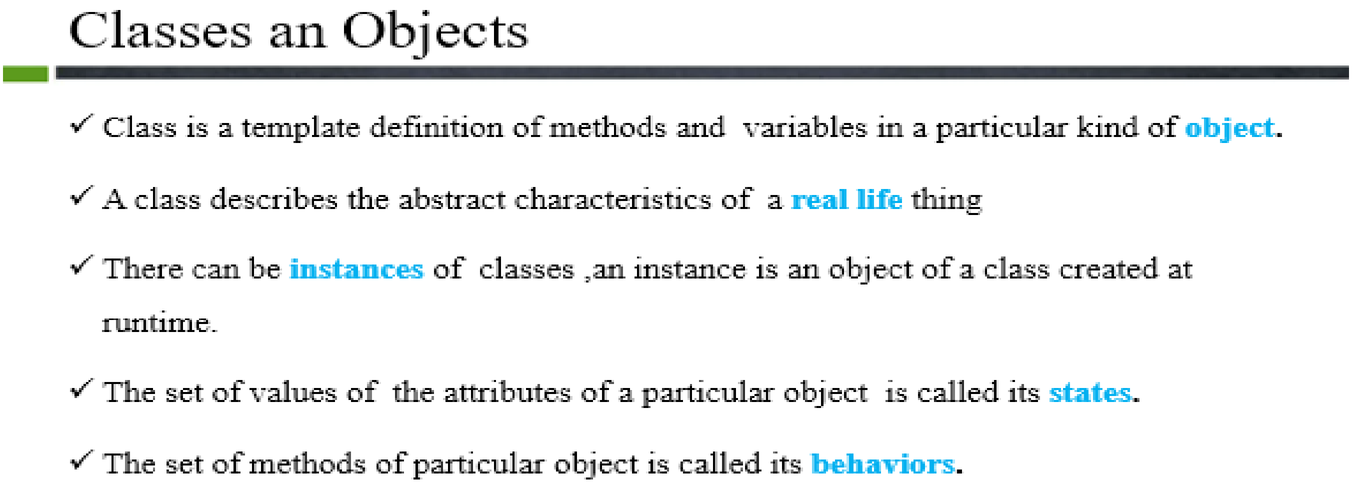
→Class is a blueprint used to create objects having same property or attribute as its class

→An Object is an instance of class which contains variables and methods

What is Python Class and Object?

- A class is a collection of objects, and an object is defined as an instance of a class processing attributes.
- A class is a blueprint of object that formed in future.

- The object is an entity that has a state and behavior.



Classes an Objects

✓ Class is a template definition of methods and variables in a particular kind of **object**.

✓ A class describes the abstract characteristics of a **real life** thing

✓ There can be **instances** of classes ,an instance is an object of a class created at runtime.

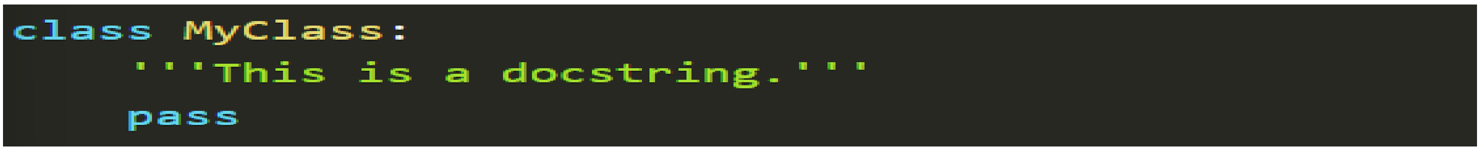
✓ The set of values of the attributes of a particular object is called its **states**.

✓ The set of methods of particular object is called its **behaviors**.

- Along with classes and objects there are some related terminology to OOPs.
  - **Instances**
  - **Constructor**
  - **Methods**
  - **Abstraction**
  - **Inheritance**
  - **Encapsulation**
  - **Polymorphism**

Object-oriented vs. Procedure-oriented Programming

Index	Object-oriented programming	Procedure Oriented Programming
1	Object-oriented programming is the problem-solving approach. The computation is done by using objects.	It is Structure oriented. Procedural programming uses a list of instructions. It performs the computation step by step.
2	OOP makes development and maintenance easier.	When the project becomes lengthy, it is not easy to maintain the code.
3	OOP provides a proper way for data hiding. It is more secure than procedural programming. You cannot access private data from anywhere.	Procedural programming does not provide any proper way for data binding, so it is less secure. In Procedural programming, we can access the private data.
4	Program is divided into objects	The program is divided into functions.



```
class MyClass:
    '''This is a docstring.'''
    pass
```

Some Important Points regarding classes and objects:

- Classes are not like functions, so we do not have to use the keyword def to create a class.
- we use the keyword **Class** along with the name of the class.
- we do not call a class as a whole; instead, we use an object to access its different attributes.
- We can assign new values and can also overwrite the previous values with the help of an object.
- In short, an object gives us permission to access the whole class.
- We can access variables in a class, like:

```
  Object_name.variable_name = “abc”
```

- Here we are setting a variable equal to abc. By doing this, its previous value will be overwritten.

Creating Object:

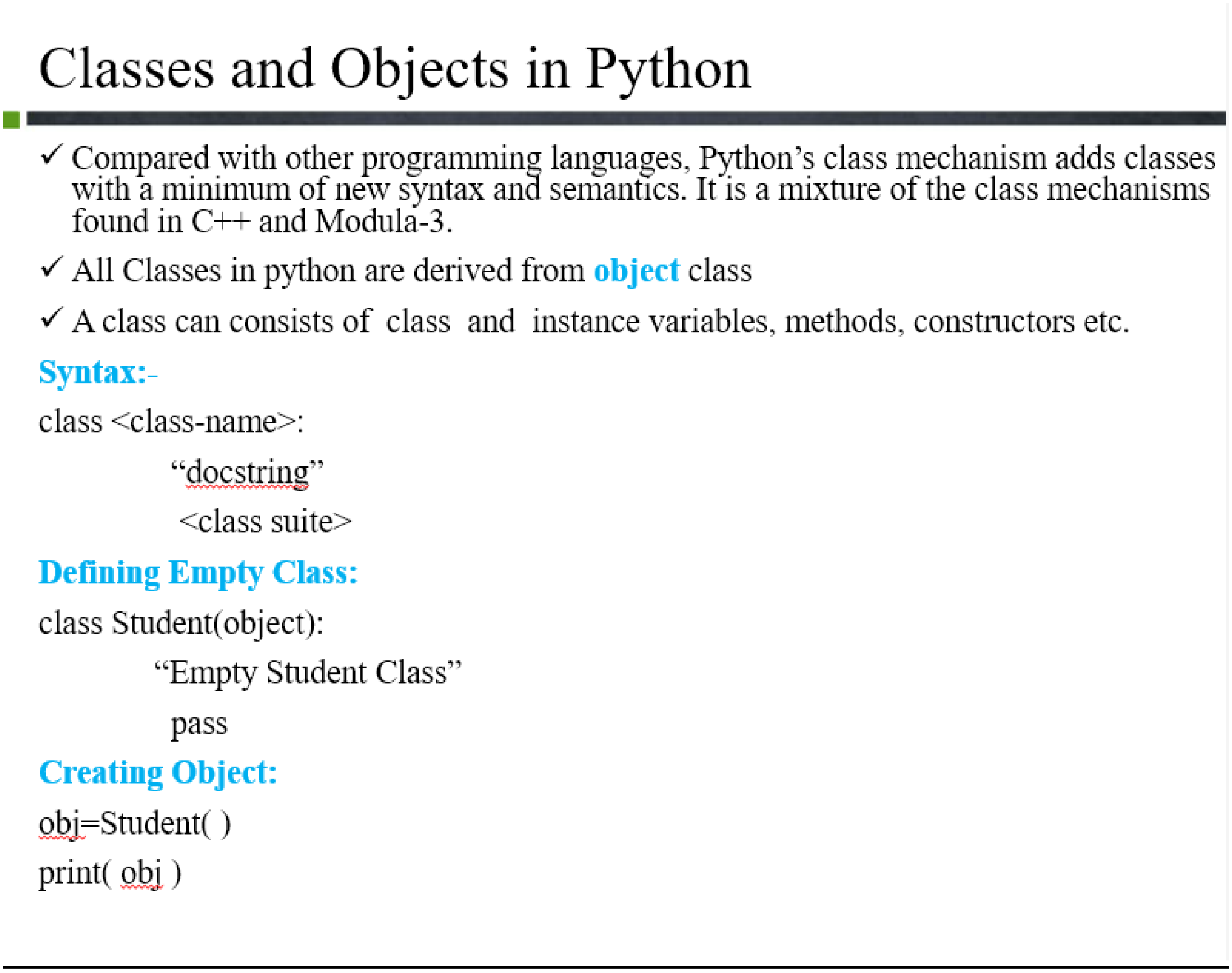
- we have a class named Student. We can create an object of it by these certain lines of code:

```
Stu1 = Student()
Stu2 = Student()
```

- we have created two objects of class Student. We can access every item in the Student class using these objects.
- There is no restriction on the number of objects a class may have, and also, there is no limit to the number of classes a program may have.

An Object consists of:

- The **State**, which is represented by attributes of an object which reflect the properties of an object.
- Methods of an object represent the object’s behaviour and the response of an object with other objects.
- **Identity**, which gives a unique name to an object so that one object can interact with other objects



Classes and Objects in Python

- ✓ Compared with other programming languages, Python’s class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3.
- ✓ All Classes in python are derived from **object** class
- ✓ A class can consists of class and instance variables, methods, constructors etc.

Syntax:-

```
class <class-name>:
        “docstring”
         <class suite>
```

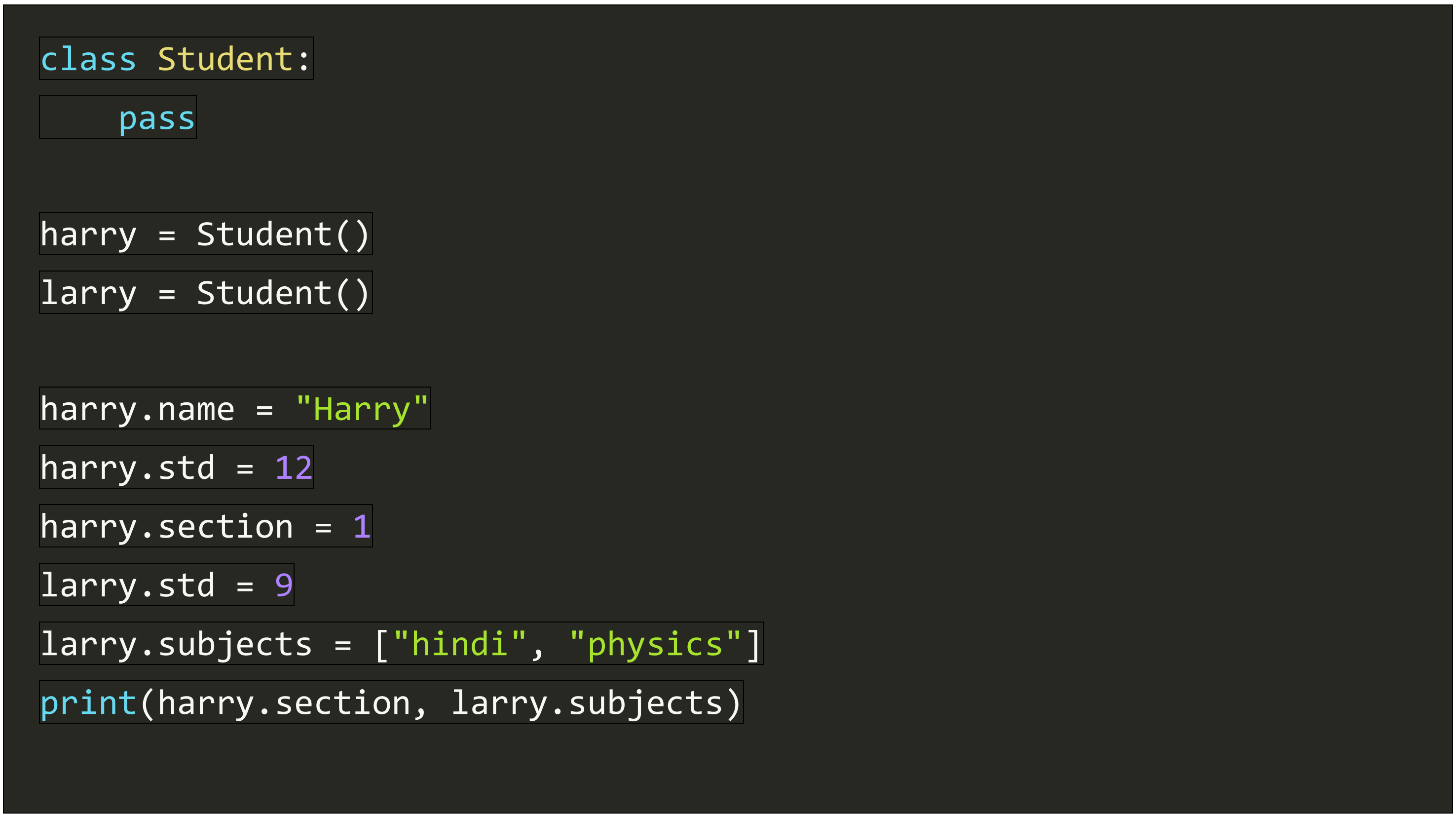
Defining Empty Class:

```
class Student(object):
        “Empty Student Class”
          pass
```

Creating Object:

```
obj=Student( )
print( obj )
```

Code File:



```
class Student:
    pass

harry = Student()
larry = Student()

harry.name = "Harry"
harry.std = 12
harry.section = 1
larry.std = 9
larry.subjects = ["hindi", "physics"]
print(harry.section, larry.subjects)
```

Instance & Class Variables

```
Class Student:
    Pass
shiva=Student()
ritika=Student()


shiva.name=’Shiva’
shiva.std=’B.tech’
shiva.section=’1’
shiva.subjects=[‘Python’,’Mathematics’,’Electronics’,’Data-Structure-
Algo’]


ritika.name=’Ritika’
ritika.std=’B.tech’
ritika.section=’1’
ritika.subjects=[‘Python’,’Mathematics’,’Electronics’,’Data-Structure-
Algo’]
print(shiva.subject,ritika.subjects)
```

we have to work with two types of variables:

- **Instance variable**
- **Class variable**

Instance Variable:

- "Instance variables are the variables for which the value of the variable is different for every instance."
- We can say that value is different for every object that we create.
- When we create a class, we defined a few variables along with it.
- Suppose we create max_marks as class variable which is same for every student object .
- When we change value of max_marks form new instance variable for shiva object of student class.
- Shiva.max_marks=60 form new instance variable.

Code that show how new instance formed:

```
class Student:
    max_marks=100
shiva=Student()
print(Student.max_marks)
print(shiva.max_marks)
# New instance variable max_marks form
shiva.max_marks=60
# Does not effect the value of class variable
print(Student.max_marks)
print(shiva.max_marks)
Student.max_marks=80
# change only happen in class variable does not effect
instance variable max_marks of shiva object
print(Student.max_marks)
print(shiva.max_marks)
```

```
Output:
100
100
100
60
80
60
```

Class Variable:

- ***"Class attributes are owned by the class directly, which means that they are not tied to any object or instance."***
- if we want to change the age for every instance from 16 to 17, then we can do it by using the class variable, which in this case is Student
- "It is worth noting that updating the value of the class variable will not change it for the instance variables of the objects, such as in the case above."
- For example we change Student.max_marks=80 change cannot reflect in shiva.max_marks

Following are the differences between Class and instance variables.

Instance variables	Class variables
When an object is created with the use of the new keyword, instance variables are created. They are destroyed when the object is destroyed.	When the program starts, static variables are created and destroyed when the program stops.
Instance variables can be accessed by calling the variable name inside the class. ***ObjectReference.VariableName.***	Static variables can be accessed by calling using a class name. ***ClassName.VariableName.***
Every instance of the class has its own copy of that variable. Changes made to the variable don't affect the other instances of that class.	There is only one copy of that variable that is shared with all instances of the class. If changes are made to that variable, all other instances will be affected.

Method:

- A method is just like a function, with a **def** keyword and a single parameter in which the object's name has to be passed.
- Using methods makes the process simpler and a lot faster.

- Using methods makes the process simpler and a lot faster.

Self Keyword:

- The self keyword is used in the method to refer to the instance of the current class we are using.
- The self keyword is passed as a parameter explicitly every time we define a method.

```
def read_number(self):
        print(self.num)
```

Constructor:

__init__method:-

- "__init__" is also called a constructor in object-oriented terminology.
- **"constructor in Python is used to assign values to the variables or data members of a class when an object is created."**
- Python treats the constructor differently as compared to C++ and Java.

- The constructor is a method with a def keyword and parameters, but the purpose of a constructor is to assign values to the instance variables of different objects.
- We can give the values by accessing each of the variables one by one, but in the case of the constructor, we pass all the values directly as parameters.
- Self keyword is used to assign value to a constructor too.
- We declare a constructor in Python using the def keyword:

```
def __init__(self):
    # body of the constructor
```

Term and how it is uses:

- The def keyword is used to define the function.
- The first argument refers to the current object which binds the instance to the init() method.
- In init() method ,arguments are optional. Constructors can be defined with any number of arguments or with no arguments.

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

p1 = Person("John", 36)
print(p1.name)
#Output: John
```

Types of constructors in Python:

We have two types of constructors in Python.

1. The default constructor is the one that does not take any arguments.
2. Constructor with parameters is known as ***parameterized*** constructor.

Methods vs Function:

Methods and Functions are very similar, yet there are some difference:

- Methods are explicitly for Object-Oriented Programming.
- The method can be used by object that it is called for. In simple terms for a method, the parameter must be an object.
- The method can only access the data that is initialized in the class the method is formed in.

Code:

```
Class Employee:
   no_of_leaves=9
   def __init__(self,name,salary,role):
       self.name=name
       self.salary=salary
       self.role=role
   def printdetails(self):
       return f”The name is {self.name}. Salary is {self.salary} and role is {self.role}”


shiva=Employee(“Shiva”,300,”Hacker”)
#ritika=Employee() ->Throw error because less parameter in Employee
#shiva.name=”shiva”
#shiva.salary=500
#shiva.role=”Instrutor”
print(shiva.salary)
```

Class Methods in Python:

- We have dealing with static methods until now.
- In object-Oriented programming, there is a concept of a class method.
- They are very different from static methods as they are limited in their functionality to the built-in class.
- They can be called by using the class name and also can be accessed by using the object.

- we cannot change the value of a variable defined in the class from outside using an object.
- Instead, if we try that, a new instance variable will be created for the class having the value we assigned.
- But no change will occur in the original value of the variable.
- we are going to know the working of a new keyword, i.e., cls. Class methods take **cls** parameter that points to the class and not the object instance when the method is called.

```
class myClass:
    @classmethod
    def myfunc (cls, arg1, arg2, ...):
                          ....
```

- ➢ myfunc defines the function that needs to be converted into a class method .
- ➢ returns: @classmethod returns a class method for function.
- ➢ Because the class method only has access to the cls argument, it cannot modify the object instance state.

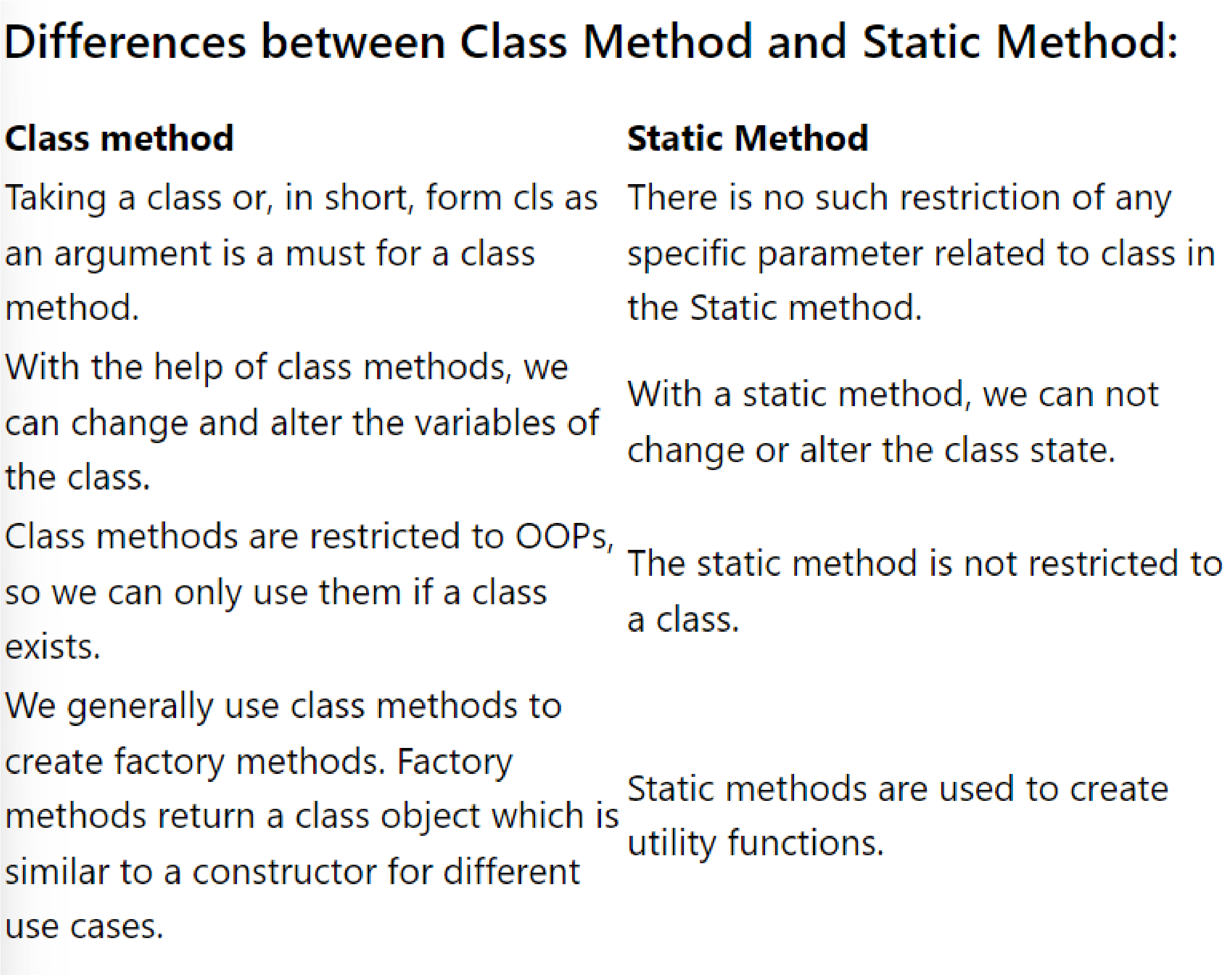
@classmethod

- ➢ It is decorator is a built-in function in Python. It can be applied to any method of the class.
- ➢ We can change the value of variables using this method.
- ➢

```
class Student:
    max_marks=100
    def change(self):
        Student.max_marks=70

    @classmethod
    def change_classmethod(cls):
        cls.max_score = 700
shiva=Student()
shiva.change()
shiva.change_classmethod()
print(Student.max_score)
Student.max_score=600
print(Student.max_score)
print(Student.max_marks)
print(shiva.max_marks)
```

Output:

```
700

600

70

70
```



Differences between Class Method and Static Method:

Class method	Static Method
Taking a class or, in short, form cls as an argument is a must for a class method.	There is no such restriction of any specific parameter related to class in the Static method.
With the help of class methods, we can change and alter the variables of the class.	With a static method, we can not change or alter the class state.
Class methods are restricted to OOPs, so we can only use them if a class exists.	The static method is not restricted to a class.
We generally use class methods to create factory methods. Factory methods return a class object which is similar to a constructor for different use cases.	Static methods are used to create utility functions.

Alternative constructor:

```
class Date:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

@classmethod
    def from_dash(cls,string):
          return cls(*string.split("-"))

date1=Date.from_dash("2008-12-5")
```

```python
print(date1.year)
#Output: 2008
```

```python
class Employee:
    no_of_leaves = 8

    def __init__(self, aname, asalary, arole):
        self.name = aname
        self.salary = asalary
        self.role = arole

    def printdetails(self):
        return f"The Name is {self.name}. Salary is {self.salary} and role is {self.role}"

    @classmethod
    def change_leaves(cls, newleaves):
        cls.no_of_leaves = newleaves

    @classmethod
    def from_dash(cls, string):
        # params = string.split("-")
        # print(params)
        # return cls(params[0], params[1], params[2])
        return cls(*string.split("-"))



harry = Employee("Harry", 255, "Instructor")
rohan = Employee("Rohan", 455, "Student")
karan = Employee.from_dash("Karan-480-Student")

print(karan.no_of_leaves)
```

```
# rohan.change_leaves(34)
#
# print(harry.no_of_leaves)
```

Static Methods in python:

- It is also easier to implement than a class method because it can be accessed without any object. However, we can also access it using a class or any instance.
- we use the **@staticmethod** decorator, which is a built-in decorator. Also, there is no need to import any module to use decorators. Using a static method in a class, we permit it to be accessed only by the class objects or inside the class.

Limitations of static method over class method:

- Unlike, class method, a static method cannot alter or change any variable value or state of the class.
- Static methods do not have any knowledge related to the

Advantage of Python static method:

- Static methods have a very clear use case. When we need some functionality not for an Object but with the complete class, we make a method static. This is advantageous when we need to create utility methods.

Code:

```
class Employee:
    no_of_leaves = 8

    def __init__(self, aname, asalary, arole):
        self.name = aname
        self.salary = asalary
        self.role = arole

    def printdetails(self):
        return f"The Name is {self.name}. Salary is {self.salary} and role is {self.role}"

    @classmethod
    def change_leaves(cls, newleaves):
        cls.no_of_leaves = newleaves

    @classmethod
    def from_dash(cls, string):
        return cls(*string.split("-"))

    @staticmethod
    def printgood(string):
        print("This is good " + string)

harry = Employee("Harry", 255, "Instructor")
rohan = Employee("Rohan", 455, "Student")
karan = Employee.from_dash("Karan-480-Student")

Employee.printgood("Rohan")
```

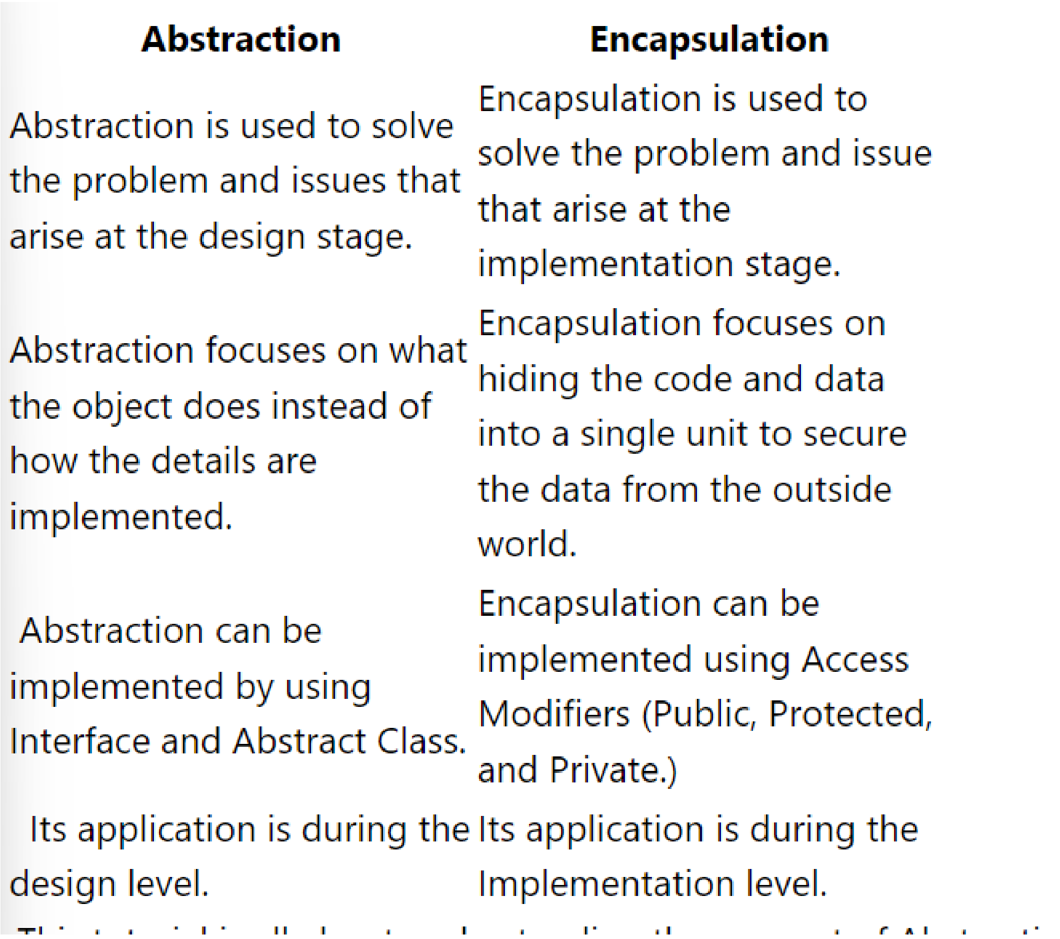
Abstraction & Encapsulation:

What is Abstraction?

- **Abstraction** refers to hiding unnecessary details to focus on the whole product instead of parts of the project separately.
- It is a mechanism that represents the important features without including implementation details.
- Abstractions helps us in portioning the program into many independent concepts so we may hide the irrelevant information in the code.
- It offers the greatest flexibility when using abstract data-type objects in different situations
- Example of Abstraction -A car has an engine, tires, windows, steering wheel ,etc. all these things used to form car .Now an engine is composed of various parts such as camshaft ,valves , oil pan, etc. these flayers the engine is an abstraction.

What is Encapsulation?

- **Encapsulation** means hiding under layers. When working with classes and handling sensitive data , global access to all the variables used in the program is not secure . In Encapsulation, the internal representation of an object is generally hidden from the outside to secure the data.



Abstraction	Encapsulation
Abstraction is used to solve the problem and issues that arise at the design stage.	Encapsulation is used to solve the problem and issue that arise at the implementation stage.
Abstraction focuses on what the object does instead of how the details are implemented.	Encapsulation focuses on hiding the code and data into a single unit to secure the data from the outside world.
Abstraction can be implemented by using Interface and Abstract Class.	Encapsulation can be implemented using Access Modifiers (Public, Protected, and Private.)
Its application is during the design level.	Its application is during the Implementation level.

Types of Inheritance:
1)Single Inheritance
2)Multiple Inheritance
3)Multilevel Inheritance
4)hybrid Inheritance

Single Inheritance:

- **Inheritance** means to receive something from one's parents or ancestors.
- The concept of Inheritance is very similar in cases of classes where a class inherits all the properties and methods of its previous class that it is inheriting from.

- Inheritance is the ability to define a new class(child class) that is modified version of an existing class(parent class).

Syntax:

```
class Parent_class_Name:
#Parent_class code block
class Child_class_Name(Parent_class_name):
#Child_class code block
```

Common terms related to inheritance are as follows:

Parents: The parent class is the one that is giving access to its methods or properties to the child class or derived class.

Child: Child class is the one that is inheriting methods and properties from parent class.

Theory:

The class that is inheriting, i.e., the child class, can inherit all the functionality of the parent class and add its functionalities also. As we have already discussed that each class can have its constructors and methods, so in case of inheritance the child class can make and use its constructor and also can use the constructor of the parent class. We can simply construct it as we did for the parent class but OOP has provided us with a simple and more useful solution known as Super().
We will be discussing super() and overriding in our ***Super() and Overriding In Classes*** tutorial of the course.
Single inheritance exists when a class is only derived from a single base class. Or in other words when a child class is using the methods and properties of only a single parent class then single inheritance exists. Single inheritance and Multiple inheritance are very similar concepts, the only major difference is the number of classes.

```
class Employee:
    no_of_leaves = 8

    def __init__(self, aname, asalary, arole):
        self.name = aname
        self.salary = asalary
        self.role = arole

    def printdetails(self):
        return f"The Name is {self.name}. Salary is {self.salary} and role is {self.role}"

    @classmethod
    def change_leaves(cls, newleaves):
        cls.no_of_leaves = newleaves

    @classmethod
    def from_dash(cls, string):
        return cls(*string.split("-"))

    @staticmethod
    def printgood(string):
        print("This is good " + string)


class Programmer(Employee):
    no_of_holiday = 56
    def __init__(self, aname, asalary, arole, languages):
        self.name = aname
        self.salary = asalary
        self.role = arole
        self.languages = languages


    def printprog(self):
        return f"The Programmer's Name is {self.name}. Salary is {self.salary} and role is {self.role}.The languages are {self.languages}"
```

```
harry = Employee("Harry", 255, "Instructor")
rohan = Employee("Rohan", 455, "Student")

shubham = Programmer("Shubham", 555, "Programmer", ["python"])
karan = Programmer("Karan", 777, "Programmer", ["python", "Cpp"])
print(karan.no_of_holiday)
```

Multiple Inheritance:

- In multiple inheritance , a class is derived from more than one class i.e multiple base classes. The child class, in this case, has features of both the parent classes.

```
➢ class Base1:
➢       def func1(self):
➢             print("this is Base1 class")
➢ class Base2:
➢       def func2(self):
➢             print("this is Base2 class")
➢ 
➢ class Child(Base1 , Base2):
➢       def func3(self):
➢             print("this is Base3 class")
➢ 
➢ obj = Child()
➢ obj.func1()
➢ obj.func2()
➢ obj.func3()
```

Output:

```
this is Base1 class
this is Base2 class
this is Base3 class
```

Method Overriding:

Override means having two methods that have the same name. They may perform same tasks or different tasks. In Python, when the same method defined in the parent class is also defined in the child class, the process is know as Method overriding. This is also true when multiple classes have the same method and are linked together somehow.

There are few rules for Method overriding that should be followed:

- The name of the child method should be the same as parents.
- **Inheritance** should be there, and we need to derive a child class from a parent class.
- **Both** of their parameters should be the same.

Code:

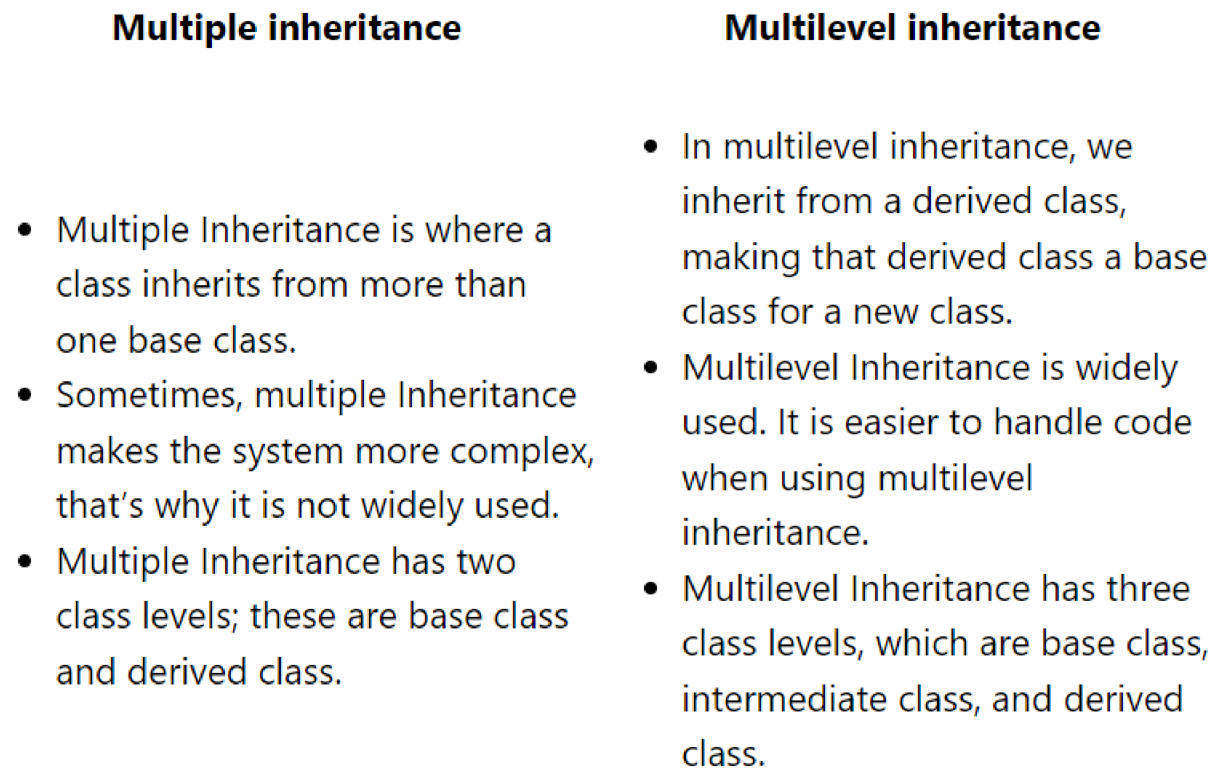
```
class Employee:
    no_of_leaves = 8
    var = 8

    def __init__(self, aname, asalary, arole):
        self.name = aname
        self.salary = asalary
        self.role = arole

    def printdetails(self):
        return f"The Name is {self.name}. Salary is {self.salary} and role is {self.role}"

    @classmethod
    def change_leaves(cls, newleaves):
        cls.no_of_leaves = newleaves

    @classmethod
    def from_dash(cls, string):
        return cls(*string.split("-"))

    @staticmethod
    def printgood(string):
        print("This is good " + string)

class Player:
    var = 9
    no_of_games = 4
    def __init__(self, name, game):
        self.name = name
        self.game =game

    def printdetails(self):
        return f"The Name is {self.name}. Game is {self.game}"

class CoolProgramer(Player, Employee):

    language = "C++"
    def printlanguage(self):
        print(self.language)

harry = Employee("Harry", 255, "Instructor")
rohan = Employee("Rohan", 455, "Student")

shubham = Player("Shubham", ["Cricket"])
karan = CoolProgramer("Karan",["Cricket"])
# det = karan.printdetails()
# karan.printlanguage()
# print(det)
print(karan.var)
```

Multilevel Inheritance:

- ➢ In multilevel inheritance, a class that is already derived from another class is derived by a third class. So in this way, the third class has all the other two former classes' features and functionalities.

```
class Parent1:
    pass
class Derived1(Parent1):
    pass
class Derived2(Derived1):
    pass
```

Multiple inheritance VS. Multilevel inheritance



Multiple inheritance	Multilevel inheritance
• Multiple Inheritance is where a class inherits from more than one base class. • Sometimes, multiple Inheritance makes the system more complex, that's why it is not widely used. • Multiple Inheritance has two class levels; these are base class and derived class.	• In multilevel inheritance, we inherit from a derived class, making that derived class a base class for a new class. • Multilevel Inheritance is widely used. It is easier to handle code when using multilevel inheritance. • Multilevel Inheritance has three class levels, which are base class, intermediate class, and derived class.

Advantages of Inheritance:

1.It reduces code redundancy

2.multilevel inheritrance provide code reusability

3.Using multilevel inheritance, code is easy to manage, and it supports code extensibility by overriding the base class functionality within child classes.

Code:

```
class Dad:
    basketball =6

class Son(Dad):
    dance =1
    basketball = 9
    def isdance(self):
        return f"Yes I dance {self.dance} no of times"

class Grandson(Son):
    dance =6
    guitar = 1

    def isdance(self):
        return f"Jackson yeah!" \
            f"Yes I dance very awesomely {self.dance} no of times"

darry = Dad()
larry = Son()
harry = Grandson()

# print(darry.guitar)

# electronic device
# pocket gadget
# phone
```

Public, Private & Protected Access Specifiers:

- ➢ In high-level programming languages like C++, Java, etc., private, protected, and public keywords are used to control the access of class members or variables. However, Python has no such keywords. Python uses a convention of prefixing the name of the variable or method with a single underscore(_) or double underscore(__) to emulate the behavior of protected and private access specifiers.

- ➢ Access modifiers are used for the restrictions of access any other class has on the particular class and its variables and methods. In other words, access modifiers decide whether other classes can use the variables or functions of a specific class or not. The arrangement of private and protected access variables or methods ensures the principle of data encapsulation

Three types of access modifiers:

- **Public Access Modifier**
- **Protected Access Modifier**
- **Private Access Modifier**

Public Access Modifier:

- In public, all the functions, variables, methods can be used publicly. Meaning, every other class can access them easily without any restriction. Public members are generally methods declared in a class that is accessible from outside the class. Any ordinary class is, by default, a public class. So, all the classes we had made till now in the previous tutorials were all public by default.

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

Protected Access Modifier:

- In the case of a protected class, its members and functions can only be accessed by the classes derived from it, i.e., its child class or classes. No other environment is permitted to access it. To declare the data members as protected, we use a single underscore "_" sign before the data members of the class.

```
class employee:
      def __init__(self, name, age):
            self._name=name # protected attribute
            self._age=age # protected attribute
```

Private Access Modifier:

- In the case of private access modifiers, the variables and functions can only be accessed within the class. The private restriction level is the highest for any class. To declare the data members as private, we use a double underscore "__" sign before the data members of the class. Here is a suggestion not to try to access private variables from outside the class because it will result in an AttributeError.

```
class employee:
      def __init__(self, name, age):
            self.__name=name # private attribute
            self.__age=age # private attribute
```

Name mangling in Python:

- Python does not have any strict rules when it comes to public, protected, or private, like java. So, to protect us from using the private attribute in any other class, Python does name mangling, which means that every member with a double underscore will be changed to _object._class__variable when trying to call using an object.
- The use of single underscore and double underscore is just a way of name mangling because Python does not take the public, private and protected terms much seriously so we have to use our naming conventions by putting

single or double underscore to let the fellow programmers know which class they can access or which they can't.

Code:

```python
# Public -
# Protected -
# Private -

class Employee:
    no_of_leaves = 8
    var = 8
    _protec = 9
    __pr = 98

    def __init__(self, aname, asalary, arole):
        self.name = aname
        self.salary = asalary
        self.role = arole

    def printdetails(self):
        return f"The Name is {self.name}. Salary is {self.salary} and role is {self.role}"

    @classmethod
    def change_leaves(cls, newleaves):
        cls.no_of_leaves = newleaves

    @classmethod
    def from_dash(cls, string):
        return cls(*string.split("-"))

    @staticmethod
    def printgood(string):
        print("This is good " + string)
```

```
emp = Employee("harry", 343, "Programmer")
print(emp._Employee__pr)
```

Polymorphism In Python:

- Polymorphism means to exist in different states. The same object or thing changing its state from one form to another is known as polymorphic. The same function or method, being used differently in different scenarios, can perfectly describe Polymorphism. It occurs mostly with base and derived classes.

Overriding and Overloading come in Polymorphism:

Polymorphism in '+' operator:-

```
print(5+6)
print("5" + "6")
```

Super() and Overriding In Classes:

How to overrode class methods in Python?

Overriding occurs when a derived class or child class has the same method that has already been defined in the base or parent class. When called, the same methods with the same name and number of parameters, the interpreter checks for the method first in a child class and runs it ignoring the method in the parent class because it is already overridden. In the case of instance variables, the case is a little different. When the method is called, the program will look for any instance variable having the same name as the one that is called in the child, then in the parent, and after that, it comes again into child class if not found.

```
class student:
    def __init__(self,year):
        self.year=year
    def std(self,name,roll):
        self.name=name
        self.roll=roll
class child(student):
    def __init__(self, year,age):
        self.year = year
        self.age=age

    def std(self, name):
        self.name = name
shiva=child(2002,19)
shiva.std('shiva')
print(shiva.name)
# print(shiva.roll) give error because std is override by same
function
```

Where does super() fit in all this?

When we want to call an already overridden method, then the use of the super function comes in. It is a built-in function, so no requirement of any module import statement. What super does is it allows us to use the method of our superclass, which in the case of inheritance is the parent class. Syntax of using super() is given below:

```
class A:
    classvar1 = "I am a class variable in class A"
    def __init__(self):
        self.var1 = "I am inside class A's constructor"
        self.classvar1 = "Instance var in class A"
        self.special = "Special"

class B(A):
    classvar1 = "I am in class B"

    def __init__(self):
        self.var1 = "I am inside class B's constructor"
        self.classvar1 = "Instance var in class B"
        # super().__init__()
        # print(super().classvar1)



a = A()
b = B()

print(b.special, b.var1, b.classvar1)
```

Diamond Shape Problem In Multiple Inheritance:

- From the start of this course, you may have noticed that I am not just teaching you the syntax so that you may learn just the practical approach to programming and could create a few programs that have no real-world value. Instead, I am trying to teach you all the theoretical and practical concepts together so you may become a successful programmer. You can do the programming by just learning the syntax, but without proper conceptual knowledge, you won't be able to develop proper logic while writing code. Our today's tutorial is also based on a theoretical concept.

- we have seen a lot of concepts related to Object-Oriented programming, such as ***Single inheritance***, ***Multiple Inheritance***, ***Multilevel Inheritance***, etc. Today we are going to discuss a problem or, more like a confusion associated with multiple inheritance. The problem is commonly known as the ***"Diamond Shape Problem."***. It is about priority related confusion, which arises when four classes are related to each other by an inheritance relationship, as shown in the image below:

Explanation:-

In the above image, we can see that class C and class B are inheriting from class A, or it can be said as class A is a parent to class B and C. And class D is inheriting from both class C and B. So, in a way, they are all in relation to one and other somehow. Let us write down the relation in code format so it will be easier to understand.

Copy

```
class A:
    pass
class B(A):
    pass
class C(A):
    pass
class D( B, C ):
    pass
```

As discussed earlier that it creates priority-related confusion, so let's clear that out here.

- If we have a method that is only present in class A and we use the class D object to call the method, it will go to class A while checking for the method name in all the classes in between and run the method in class A.
- However, if the same method is also present in class B, then it will run the B class method because, for class D, class B holds more priority than class A. The reason is that class D is derived from class B, which is further derived from A. So, a closer relation exists with B than A.
- If the same method is present in classes C and B, it may create a little bit of confusion. But as we have already discussed in Tutorial #61, that in such cases, our priority is based from left to right, meaning whichever class is on the left side will be given more priority, and then we will move towards the right one. In this case, the left class is B, so the method in B will de be executed first.

If the C class would be on the left, such as

```
class D( C,B ):
      pass
```

Then priority would be given to C.

```
class A:
    def met(self):
        print("This is a method from class A")

class B(A):
    def met(self):
        print("This is a method from class B")

class C(A):
    def met(self):
        print("This is a method from class C")

class D(C, B):
    def met(self):
        print("This is a method from class D")



a = A()
b = B()
c = C()
d = D()

d.met()
```

Operator Overloading and Dunder methods:

Operator overloading and Dunder Methods may be new concepts for some of you. However, we have already seen similar concepts in which different methods act differently on different occasions and places. Let us understand the Operator overloading first.

Operator Overloading in Python:

Operator overloading means giving new meanings to an operator. In simple words, it means to assign new functionality to an operator beyond its normal functioning. We will go with the most common and easiest that we could find related to the concept, i.e., the + sign. For numbers, it is used for addition between them, but in the case of a string, it is used to join or combine two strings, working differently in two different scenarios. The operators are methods defined in respective classes. Defining methods for operators is known as operator overloading.

Python Dunder Methods Or Special Functions :

Sunder methods in python are special methods .In python , we sometimes see method names with a double underscore (__), such as the __init__ method that every class has .These methods are called “dunder” methods. In Python, Dunder methods are used for operator overloading and customizing some other functionn’s behavior.

Python usually calls dunder methods under the hood .Suppose we want to join a string with a number using the + sign .Now joining between two different data types is not possible in Python, and the resultant in such a case will be an error .So for this purpose , we can use a function provided to us by python , named as dunder function.We will write such code in it so that it may first convert the number to a string and then join them, or any logic will be fine too until it does what we require .We can even Methods staring with a double underscore(__) and ending with a double underscore (__) represent dunder methods.

- Check https://docs.python.org/2/library/operator.html to explore more about operator overloading.

Some Dunder method:

The __init__(), __str__(), __len__() and __del__() methods

__str__ and __repr__ functions :

Both of these built-in methods are used to return a presentable description of any object rather than the default one. The difference in them is the way of writing them. The __str__ method is mainly written for the end-user, while __repr__ is written for a developer. It is overridden to return a printable string representation of any user-defined class. An interesting thing to note here is that the priority of __str__ is greater than __repr__. This means that if we pass an object into a print statement, it will return us the __str__ string even if __repr__ is also present there. In such cases, if we want to print __repr__, we have to call it exclusively with the object name in the print statement.

1. **Differnce between __str__and __repr__ functions:**
   If the implementation of __str__ is missing, then __repr__ function is used as a fallback. If the implementation of __repr__ is missing, then there will be no fallback.
2. If __repr__ function is returning the object's String representation, we can skip the implementation of __str__ function.
3. The priority of __str__ is higher than __repr__.

Code:

```
class Employee:
    no_of_leaves = 8

    def __init__(self, aname, asalary, arole):
        self.name = aname
        self.salary = asalary
        self.role = arole

    def printdetails(self):
        return f"The Name is {self.name}. Salary is {self.salary} and 
role is {self.role}"

    @classmethod
    def change_leaves(cls, newleaves):
        cls.no_of_leaves = newleaves
    def __add__(self, other):
        return self.salary + other.salary

    def __truediv__(self, other):
        return self.salary / other.salary

    def __repr__(self):
        return f"Employee('{self.name}', {self.salary},
'{self.role}')"

    def __str__(self):
        return f"The Name is {self.name}. Salary is {self.salary} and
role is {self.role}"

emp1 =Employee("Harry", 345, "Programmer")
# emp2 =Employee("Rohan", 55, "Cleaner")
print(str(emp1))
```

Use of __str__ and __repr__:

```
class student:
    def __init__(self):
        pass
    def __str__(self):
        return "shiva"
    def __repr__(self):
        return "srivastava"
# print(student)
shiva=student()
print(shiva)
print(shiva.__repr__())
```

Special Method:

```
class Book:
    def __init__(self, title, author, pages):
        print("A book is created")
        self.title = title
        self.author = author
        self.pages = pages

    def __str__(self):
        return "Title: %s, author: %s, pages: %s" %(self.title, self.author,
self.pages)
```

```
    def __len__(self):
        return self.pages

    def __del__(self):
        print("A book is destroyed")
```

__del__ dunder method:

Both __delete__ and __del__ are dunder or magic methods in Python. Dunder or magic methods in Python are the methods having two prefix and suffix underscores in the method name. Dunder here means “Double Under (Underscores)”. These are commonly used for operator overloading.

__del__

__del__ is a destructor method which is called as soon as all references of the object are deleted i.e when an object is garbage collected.

```
# Creating object of Example
# class as an descriptor attribute
# of this class
class Foo(object):
   exp = Example()

# Driver's code
f = Foo()
del f.exp

# print(exp) since exp deleted after calling del
```

Abstract Base Class & @abstractmethod:

- An abstract class is a class that holds an abstract method.
- An abstract method is a method defined inside an abstract class.
- It is important to remember that we can not make an object for an abstract class.
- **Following is the syntax for defining an abstract method in an abstract class in Python:**

```
from abc import ABC, abstractmethod
Class MyClass(ABC):
      @abstractmethod
```

```
    def mymethod(self):
        #empty body
        pass
```

```
from abc import ABC, abstractmethod
```

Important Points about abstract class in python:

1. Abstract methods are defined in the abstract class. They mostly do not have the body, but it is possible to implement abstract methods in the abstract class. Any subclass deriving from such an abstract class still needs to provide an implementation for that abstract method.
2. An abstract class can have both abstract methods as well as concrete methods.
3. The abstract class works as a template for other classes.
4. Using the abstract class, we can define a structure without properly implementing every method.
5. It is not possible to create objects of an abstract class because Abstract class cannot be instantiated.
6. An error will occur if the abstract method has not been implemented in the derived class.

Code:

```
# from abc import ABCMeta, abstractmethod
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def printarea(self):
        return 0

class Rectangle(Shape):
    type = "Rectangle"
    sides = 4
    def __init__(self):
        self.length = 6
        self.breadth = 7

    def printarea(self):
        return self.length * self.breadth

rect1 = Rectangle()
print(rect1.printarea())
```

Setters & Property Decorators:

- **Decorators** are functions that take another function as an argument and their purpose is to modify the other function without changing it.
- A **Property decorator** is a pythonic way to use getters and setters in object-oriented programming, which comes from the Python property class.
- Python property decorator is composed of four thing, i.e., getter ,setter, deleted and Doc. The first three are methods, and the fourth one is a docstring or comment.
- Use @property along with the getter method to access the value of the attribute .
- Without a setter, it is imosssible to update used in Oop to set the value passed as parameter during object creation.
- Setters are usually used in Oop to set the value of private attributes in a class .

Setters:

- Setters are a great way of performing encapsulation.
- **@function_name.setter is a setter method with which we can set the value of the attribute**

```
@function_name.setter
#def function
```

Deleter:

- Deleter is used to delete the values passed as a parameter before.
- ***@function_name.deleter*** **is a deleter method which can delete the assigned value by the setter method**

```
# Deleter method
@function_name.deleter
```

Advantages of @property in Python:

Following are some advantages of using @property in Python:

- The syntax of defining @property is very concise and readable.
- We can access instance attributes while using the getters and setter to validate new values. This will avoid accessing or modifying the data directly.
- By using @property, we can reuse the name of a property. This will prevent us from creating new names for the getters, setters, and deleters.

Code:

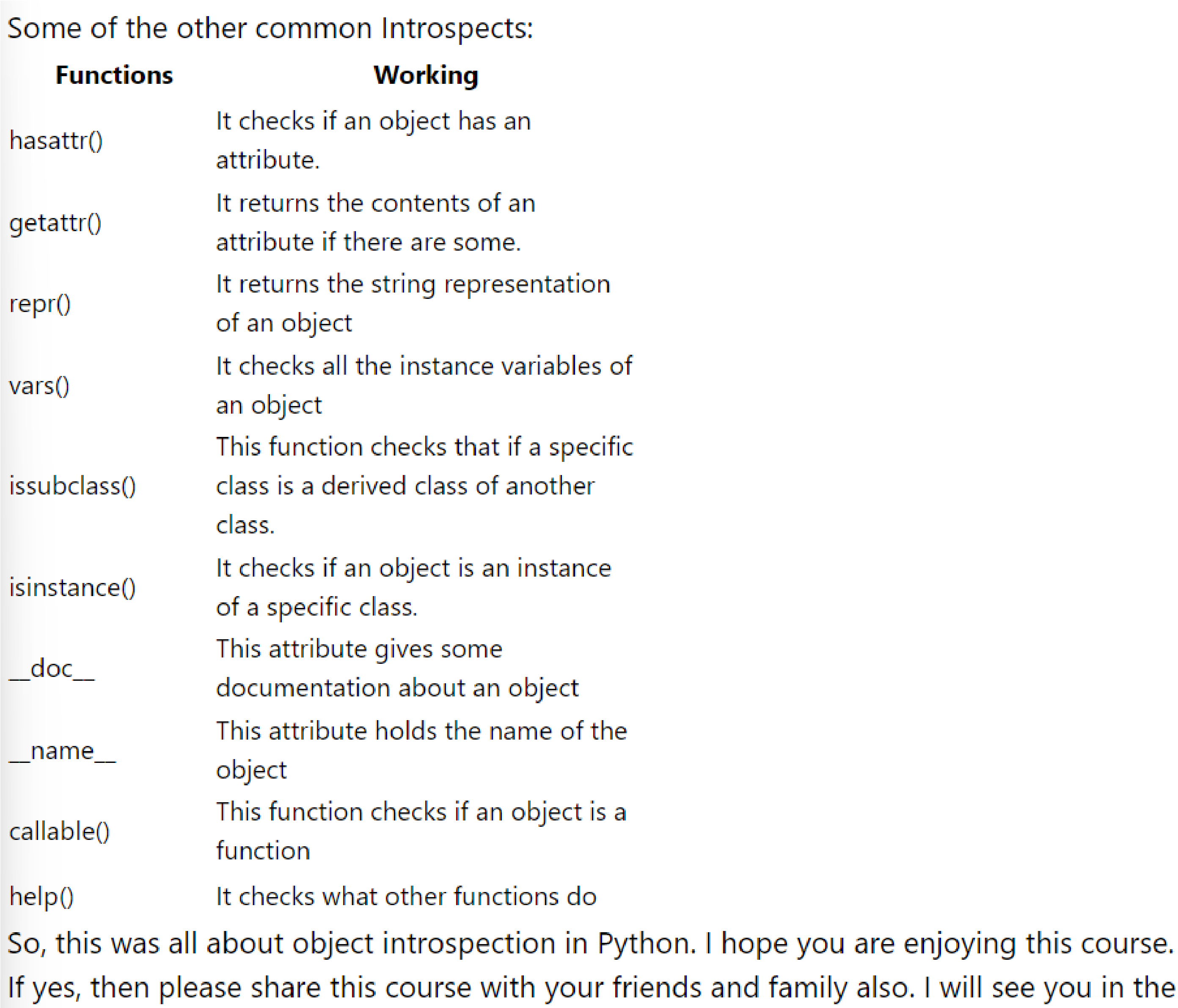
```
class Employee:
    def __init__(self, fname, lname):
        self.fname = fname
        self.lname = lname
        # self.email = f"{fname}.{lname}@codewithharry.com"

    def explain(self):
        return f"This employee is {self.fname} {self.lname}"

    @property
    def email(self):
        if self.fname==None or self.lname == None:
            return "Email is not set. Please set it using setter"
        return f"{self.fname}.{self.lname}@codewithharry.com"
    @email.setter
    def email(self, string):
        print("Setting now...")
        names = string.split("@")[0]
        self.fname = names.split(".")[0]
        self.lname = names.split(".")[1]

    @email.deleter
    def email(self):
        self.fname = None
        self.lname = None



hindustani_supporter = Employee("Hindustani", "Supporter")
# nikhil_raj_pandey = Employee("Nikhil", "Raj")

print(hindustani_supporter.email)

hindustani_supporter.fname = "US"

print(hindustani_supporter.email)
hindustani_supporter.email = "this.that@codewithharry.com"
print(hindustani_supporter.fname)

del hindustani_supporter.email
print(hindustani_supporter.email)
hindustani_supporter.email = "Harry.Perry@codewithharry.com"
print(hindustani_supporter.email)
```

Object Introspection:

- **Type(object)**
- **Id(object)**

-
  ```
  o=Myclass()
  print(dir(o))
  ```

Types of introspects:



Some of the other common Introspects:

Functions	Working
hasattr()	It checks if an object has an attribute.
getattr()	It returns the contents of an attribute if there are some.
repr()	It returns the string representation of an object
vars()	It checks all the instance variables of an object
issubclass()	This function checks that if a specific class is a derived class of another class.
isinstance()	It checks if an object is an instance of a specific class.
__doc__	This attribute gives some documentation about an object
__name__	This attribute holds the name of the object
callable()	This function checks if an object is a function
help()	It checks what other functions do

So, this was all about object introspection in Python. I hope you are enjoying this course. If yes, then please share this course with your friends and family also. I will see you in the