**Experiment 5.1**

**Aim:**  Python program to create and list of employees using Employee class. Program should also print total number of employees. Employee class should have

● empcount (class variable)

● id and name (instance variable)

● constructor to set id,

● set\_name( ), get\_name ( ), get\_id( ) methods (instance method)

● set\_emp\_count ( ) (class method)

**Theory:**

A class is a user-defined blueprint or prototype from which objects are created. Classes provide a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by their class) for modifying their state.

To understand the need for creating a class let’s consider an example, let’s say you wanted to track the number of dogs that may have different attributes like breed, age. If a list is used, the first element could be the dog’s breed while the second element could represent its age. Let’s suppose there are 100 different dogs, then how would you know which element is supposed to be which? What if you wanted to add other properties to these dogs? This lacks organization and it’s the exact need for classes.

Class creates a user-defined data structure, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

**Some points on Python class:**

* Classes are created by keyword class.
* Attributes are the variables that belong to a class.
* Attributes are always public and can be accessed using the dot (.) operator. Eg.: Myclass.Myattribute

**Class Definition Syntax:**

class ClassName:

# Statement-1

.

.

.

# Statement-N

An Object is an instance of a Class. A class is like a blueprint while an instance is a copy of the class with *actual values*. It’s not an idea anymore, it’s an actual dog, like a dog of breed pug who’s seven years old. You can have many dogs to create many different instances, but without the class as a guide, you would be lost, not knowing what information is required.  
An object consists of :

* **State:** It is represented by the attributes of an object. It also reflects the properties of an object.
* **Behavior:** It is represented by the methods of an object. It also reflects the response of an object to other objects.
* **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

**The self**

* Class methods must have an extra first parameter in the method definition. We do not give a value for this parameter when we call the method, Python provides it.
* If we have a method that takes no arguments, then we still have to have one argument.
* This is similar to this pointer in C++ and this reference in Java.

When we call a method of this object as myobject.method(arg1, arg2), this is automatically converted by Python into MyClass.method(myobject, arg1, arg2) – this is all the special self is about.

**\_\_init\_\_ method**

The \_\_init\_\_ method is similar to constructors in C++ and Java. Constructors are used to initializing the object’s state. Like methods, a constructor also contains a collection of statements(i.e. instructions) that are executed at the time of Object creation. It runs as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object.

## Class and Instance Variables

Instance variables are for data, unique to each instance and class variables are for attributes and methods shared by all instances of the class. Instance variables are variables whose value is assigned inside a constructor or method with self whereas class variables are variables whose value is assigned in the class.

Defining instance variable using a constructor.

**Class and instance Methods**

In [Object-oriented programming](https://pynative.com/python/object-oriented-programming/), when we design a class, we use the following three methods

* [Instance method](https://pynative.com/python-instance-methods/) 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.
* [Class method](https://pynative.com/python-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.

**Program:**

class Employee:

    empcount=0

    def \_\_init\_\_(self,id):

        self.id=id

    def set\_id(self,id):

        self.id = id

    def set\_emp\_count(self,empcount):

        self.empcount=empcount

    def set\_name(self,name):

        self.name = name

    def get\_name(self):

        return self.name

    def get\_id(self):

        return self.id

n=int(input("Enter the number of employees: \n"))

id = []

name = []

for i in range (n):

    id.append(input("Enter Employee-{} ID: \n".format(i+1)))

    name.append(input("Enter Employee-{} Name: \n".format(i+1)))

print()

print("Employee Id \t Employee Name")

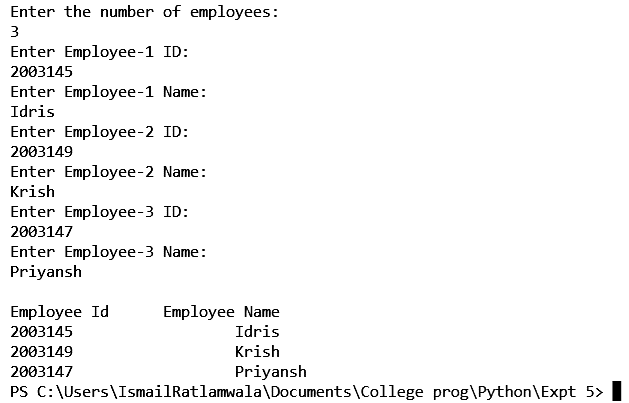
for j in range(n):

    emp=Employee(id[j])

    emp.set\_name(name[j])

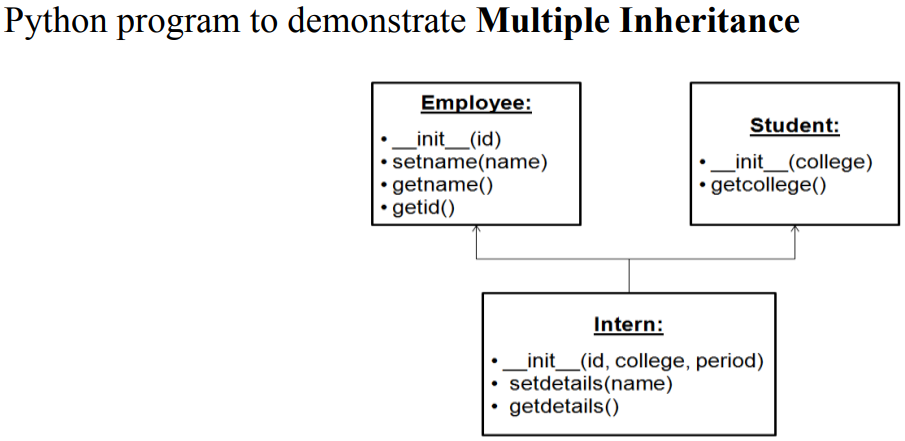
    print(emp.get\_id(),"\t\t",emp.get\_name())

**OUTPUT :**

****

# Experiment 5.2

**Aim:**



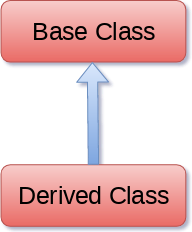
Python program to demonstrate Multiple Inheritance

# Python Inheritance

Inheritance is an important aspect of the object-oriented paradigm. Inheritance provides code reusability to the program because we can use an existing class to create a new class instead of creating it from scratch.

In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class. A child class can also provide its specific implementation to the functions of the parent class. In this section of the tutorial, we will discuss inheritance in detail.

In python, a derived class can inherit base class by just mentioning the base in the bracket after the derived class name. Consider the following syntax to inherit a base class into the derived class.



### Syntax:

class derived-class(baseclass):

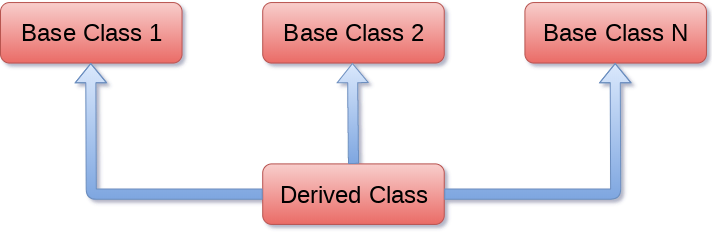
<class-suite>

class derive-class(base class1,baseclass2,.....baseclassn):

<class-suite>

## Python Multiple inheritance

Python provides us the flexibility to inherit multiple base classes in the child class.



### Syntax:

class Base1:

<class-suite>

class Base2:

<class-suite>

.

class BaseN:

<class-suite>

class Derived(Base1, Base2, ...… BaseN):

<class-suite>

**Code:**

class Employee:

def \_\_init\_\_(self, id):

self.id = id

def setname(self, name):

self.name = name

def getName(self):

return self.name

def getId(self):

return self.id

class Student:

def \_\_init\_\_(self, college):

self.college = college

def getCollege(self):

return self.college

class Intern(Employee, Student):

def \_\_init\_\_(self, id, college, period):

Employee.\_\_init\_\_(self,id)

Student.\_\_init\_\_(self,college)

self.period = period

def setDetails(self, name):

self.name = name

def getDetails(self):

return self.name

intern1 = Intern("1","TSEC","6 months")

intern1.setDetails("XYZ")

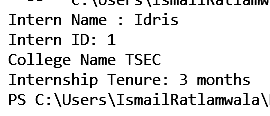
print("Intern Name : "+intern1.getName())

print("Intern ID: "+intern1.getId())

print("College Name "+intern1.getCollege())

print("Internship Tenure: "+intern1.period)

**OUTPUT:**

****

**Experiment 5.3**

**Aim :** WAP to demonstrate operator overloading in Python

**Theory:**

Operator overloading

Operator Overloading means giving extended meaning beyond their predefined operational meaning. For example operator + is used to add two integers as well as join two strings and merge two lists. It is achievable because ‘+’ operator is overloaded by int class and str class. You might have noticed that the same built-in operator or function shows different behavior for objects of different classes, this is called *Operator Overloading*.

How to overload the operators in Python?

Consider that we have two objects which are a physical representation of a class (user-defined data type) and we have to add two objects with binary ‘+’ operator it throws an error, because compiler don’t know how to add two objects. So we define a method for an operator and that process is called operator overloading. We can overload all existing operators but we can’t create a new operator. To perform operator overloading, Python provides some special function or magic function that is automatically invoked when it is associated with that particular operator. For example, when we use + operator, the magic method \_\_add\_\_ is automatically invoked in which the operation for + operator is defined.

Overloading binary + operator in Python :

When we use an operator on user defined data types then automatically a special function or magic function associated with that operator is invoked. Changing the behavior of operator is as simple as changing the behavior of method or function. You define methods in your class and operators work according to that behavior defined in methods. When we use + operator, the magic method \_\_add\_\_ is automatically invoked in which the operation for + operator is defined. There by changing this magic method’s code, we can give extra meaning to the + operator.

**Syntax for operator overloading:**

classA:

    def \_\_init\_\_(self, a):

        self.a = a

    # adding two objects

    def \_\_add\_\_(self, o):

        return self.a + o.a

ob1 = A(1)

ob2 = A(2)

ob3 = A("Geeks")

ob4 = A("For")

print(ob1 + ob2)

print(ob3 + ob4)

# Python magic method

Python magic method is defined as the special method which adds "magic" to a class. It starts and ends with double underscores, for example, \_init\_ or \_str\_.

The built-in classes define many magic methods. The dir() function can be used to see the number of magic methods inherited by a class. It has two prefixes, and suffix underscores in the method name.

Magic methods are also called **Dunder methods.**

\_\_str\_\_

This function computes "informal" or a nicely printable string representation of an object and must return a string object.

\_\_repr\_\_

This function is called by the repr() built-in function to compute the "official" string representation of an object and returns a machine-readable representation of a type. The goal of the \_repr\_ is to be unambiguous.

\_\_len\_\_

This function should return the length of an object.

#### Binary operators:

|  |  |
| --- | --- |
| Operator | Magic Method |
| **+** | \_\_add\_\_(self, other) |
| **–** | \_\_sub\_\_(self, other) |
| **\*** | \_\_mul\_\_(self, other) |
| **/** | \_\_truediv\_\_(self, other) |
| **//** | \_\_floordiv\_\_(self, other) |
| **%** | \_\_mod\_\_(self, other) |
| **\*\*** | \_\_pow\_\_(self, other) |
| >> | \_\_rshift\_\_(self, other) |
| << | \_\_lshift\_\_(self, other) |
| & | \_\_and\_\_(self, other) |
| | | \_\_or\_\_(self, other) |
| ^ | \_\_xor\_\_(self, other) |

**Program:**  
  
class Point:

def \_\_init\_\_(self, x=0, y=0):

self.x = x

self.y = y

def \_\_str\_\_(self):

return "({0},{1})".format(self.x, self.y)

def \_\_add\_\_(self, second):

x = self.x + second.x

y = self.y + second.y

return Point(x, y)

p1 = Point(1, 2)

p2 = Point(2, 3)

p3 = Point("Harsh ", "Yash ")

p4 = Point("Shah", "Savaliya")

print(p1+p2)

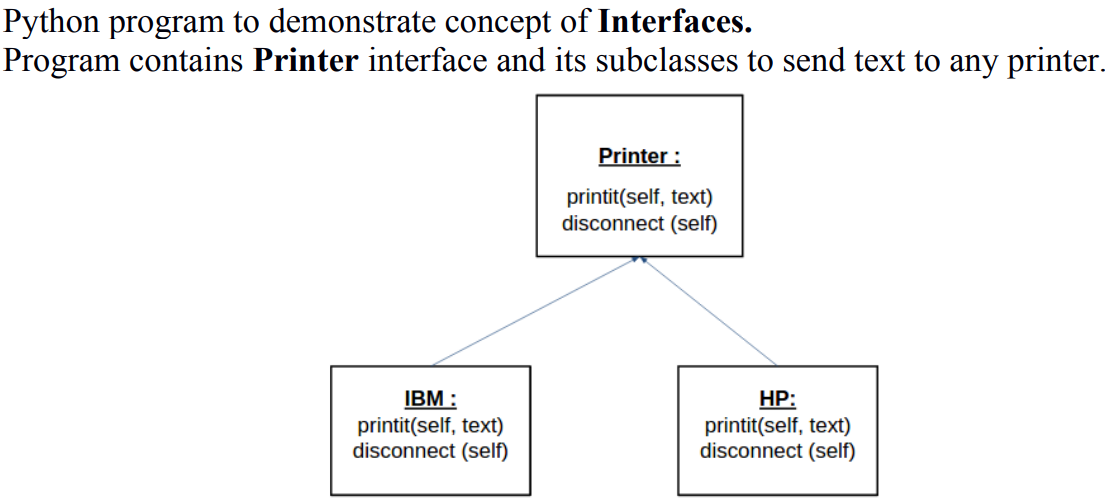
print(p3+p4)

**Output:**



**5.4 INTERFACES**

**Aim :**



**Theory :**

Interfaces play an important role in software engineering. As an application grows, updates and changes to the code base become more difficult to manage. More often than not, you wind up having classes that look very similar but are unrelated, which can lead to some confusion. In this tutorial, you’ll see how you can use a **Python interface** to help determine what class you should use to tackle the current problem.

**In this tutorial, you’ll be able to:**

* **Understand** how interfaces work and the caveats of Python interface creation
* **Comprehend** how useful interfaces are in a dynamic language like Python
* **Implement** an informal Python interface
* **Use** abc.ABCMeta and @abc.abstractmethod to implement a formal Python interface

Interfaces in Python are handled differently than in most other languages, and they can vary in their design complexity. By the end of this tutorial, you’ll have a better understanding of some aspects of Python’s data model, as well as how interfaces in Python compare to those in languages like Java, C++, and Go.

Interfaces are not necessary in Python. This is because Python has proper multiple inheritance, and also ducktyping, which means that the places where you must have interfaces in Java, you don't have to have them in Python.

That said, there are still several uses for interfaces. Some of them are covered by Pythons Abstract Base Classes, introduced in Python 2.6. They are useful, if you want to make base classes that cannot be instantiated, but provide a specific interface or part of an implementation.

Another usage is if you somehow want to specify that an object implements a specific interface, and you can use ABC's for that too by subclassing from them. Another way is zope.interface, a module that is a part of the Zope Component Architecture, a really awesomely cool component framework. Here you don't subclass from the interfaces, but instead mark classes (or even instances) as implementing an interface. This can also be used to look up components from a component registry.

**Implementing interface**

Interface acts as a blueprint for designing classes, so interfaces are implemented using **implementer** decorator on class. If a class implements an interface, then the instances of the class provide the interface. Objects can provide interfaces directly, in addition to what their classes implement.

**Methods**

* **implementedBy(class) –** returns a boolean value, True if class implements the interface else False
* **providedBy(object) –** returns a boolean value, True if object provides the interface else False
* **providedBy(class) –** returns False as class does not provide interface but implements it
* **list(zope.interface.implementedBy(class)) –** returns the list of interfaces implemented by a class
* **list(zope.interface.providedBy(object)) –** returns the list of interfaces provided by an object.
* **list(zope.interface.providedBy(class)) –** returns empty list as class does not provide interface but implements it.

**Python Interface Overview**

At a high level, an interface acts as a **blueprint** for designing classes. Like classes, interfaces define methods. Unlike classes, these methods are abstract. An **abstract method** is one that the interface simply defines. It doesn’t implement the methods. This is done by classes, which then **implement** the interface and give concrete meaning to the interface’s abstract methods.

Python’s approach to interface design is somewhat different when compared to languages like [Java](https://realpython.com/oop-in-python-vs-java/), Go, and [C++](https://realpython.com/python-vs-cpp/). These languages all have an interface keyword, while Python does not. Python further deviates from other languages in one other aspect. It doesn’t require the class that’s implementing the interface to define all of the interface’s abstract methods.

**Program :**

from abc import \*

class printer(ABC):

@abstractmethod

def \_print(self,text):

pass

@abstractmethod

def disconnect(self):

pass

class Ibm(printer):

def \_print(self, text):

print(text)

def disconnect(self):

print("bye")

class hp:

def \_print(self, text):

print(text)

def disconnect(self):

print("bye")

name = input("printer name")

text = input("enter text")

if name== 'HP':

p=hp()

elif name== 'IBM':

p=Ibm()

p.\_print(text)

p.disconnect()

**Output :**

