1. What is the concept of an abstract superclass?

Answer:

A class which contains one or more abstract methods is called an abstract class. It is a class that create a blueprint for other classes. It allow us to create a set of methods that are needs to be created within any child classes built from the abstract class. An abstract method is a method without an implementation while an abstract class may or may not include abstract methods. Python language offer a module that allow us to define abstract classes since it doesn't directly support abstract classes. The module which is used to define an abstract class is abc (abstract base class) module. This module provide us with the infrastructure for defining abstract base classes. For example:

from abc import ABC

class AbstractClass1(ABC):

pass

To define an abstract method, we need to use the @abstractmethod decorator. Here is an example for that:

from abc import ABC, abstractmethod

class AbstractClass2(ABC):

@abstractmethod

def draw(self):

pass

Example:

|  |
| --- |
| # employee.py  from abc import ABC, abstractmethod  class Employee(ABC):  def \_\_init\_\_(self, first\_name, last\_name):  self.first\_name = first\_name  self.last\_name = last\_name  @property  def name(self):  return f"{self.first\_name} {self.last\_name}"  @abstractmethod  def salary(self):  pass    class PermanentEmployee(Employee):  def \_\_init\_\_(self, first\_name, last\_name, salary):  super().\_\_init\_\_(first\_name, last\_name)  self.salary = salary    # overriding abstract method  def salary(self):  return self.salary  class TempEmployee(Employee):  def \_\_init\_\_(self, first\_name, last\_name, worked\_hours, rate):  super().\_\_init\_\_(first\_name, last\_name)  self.worked\_hours = worked\_hours  self.rate = rate    # overriding abstract method  def salary(self):  return self.worked\_hours \* self.rate  class Payroll:  def \_\_init\_\_(self):  self.employee\_list = []  def add(self, employee):  self.employee\_list.append(employee)  def print\_all(self):  for e in self.employee\_list:  print(f"{e.name} \t ${e.salary()}")  # app.py  from employee import PermanentEmployee  from employee import TempEmployee  from employee import Payroll  payroll = Payroll()  payroll.add(PermanentEmployee('X', 'Y', 6500))  payroll.add(TempEmployee('X2', 'Y2', 200, 50))  payroll.print\_all() |

2. What happens when a class statement's top level contains a basic assignment statement?

Answer:

When a class statement’ top level contains a basic assignment statement, it is considered as class attribute or class level variable and any change in the value of class attribute will affect all the instances of the class.

|  |
| --- |
| class Test1:  val = 999    def \_\_init\_\_(self, val2):  self.val2 = val2    s = Test1(2)  print(s.val)  print(s.val2)  s.val = 5  # observe value is changed for both the objects  print(s.val) |

3. Why does a class need to manually call a superclass's \_\_init\_\_ method?

Answer:

The \_\_init\_\_ method of the child class overrides the \_\_init\_\_ method of the parent class, perhaps, we have to manually call a parent or superclass’s \_\_init\_\_ method using super() method.

|  |
| --- |
| class Person:  def \_\_init\_\_(self,name, age):  self.name = name  self.age = age    class Employee(Person):  def \_\_init\_\_(self,name, age, salary):  super().\_\_init\_\_(name, age)  self.salary = salary  emp = Employee('Debabrata', 28, 65000)  print(emp.\_\_dict\_\_) |

4. How can you augment, instead of completely replacing, an inherited method?

Answer:

We need to redefine the inherited method in a subclass and need to call back to the superclass’s version of the method manually from the new version of the method in the subclass by passing the self instance to the superclass’s version of the method manually to augment an inherited method instead of completely replacing it.

|  |
| --- |
| class Person:  def \_\_init\_\_(self,name, gender):  self.name = name  self.gender = gender  class Employee(Person):  def \_\_init\_\_(self,name, gender, salary):  super().\_\_init\_\_(name, gender)  self.salary = salary  emp1 = Employee('Debabrata', 'Male', 65000)  print(emp1.\_\_dict\_\_) |

5. How is the local scope of a class different from that of a function?

Answer:

A variable defined inside a function belongs to the local scope of that function, and can only be used inside that function and exists for as long as the function is executing. If the variable is declared without self inside of a class then it is called class level variables or class attributes and it also has a local variable scope. They are referenced by their names within the same scope but they can also be accessed from outside this scope if we use the attribute access operator on a class or an instance of the class.

|  |
| --- |
| def display\_name(name):  name = name  print(f'{name}')  print(display\_name('Debabrata Ghorai'))  try:  name  except NameError:  print('Name varible is not available outside display\_name function scope')  class Person:  name = "Debabrata"  def \_\_init\_\_(self):  pass  # Accessing species using class name  print(Person.name)  # Accessing species using instance of class  Male = Person()  print(Male.name) |