1. What is the concept of an abstract superclass?

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

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

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

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

1. What is the concept of an abstract superclass?

Ans:In object-oriented programming, an abstract class is a [class](https://www.pythontutorial.net/python-oop/python-class/) that cannot be instantiated. However we can create classes that inherit from an abstract class.

An abstract class can be considered as a blueprint for other classes.

It allows you to create a set of methods that must be created within any child classes built from the abstract class.

A class which contains one or more abstract methods is called an abstract class.

An abstract method is a method that has a declaration but does not have an implementation.

While we are designing large functional units we use an abstract class. When we want to provide a common interface for different implementations of a component, we use an abstract class.

Python doesn’t directly support abstract classes. But it does offer a [module](https://www.pythontutorial.net/python-basics/python-module/) that allows you to define abstract classes.

To define an abstract class, you use the abc (abstract base class) module.

The abc module provides you with the infrastructure for defining abstract base classes.

## example

## **Python abstract class example for** developing a payroll program for a company.

The company has two groups of employees: full-time employees and hourly employees.

The full-time employees get a fixed salary while the hourly employees get paid by hourly wages for their services.

The payroll program needs to print out a payroll that includes employee names and their monthly salaries.

We may have the following classes: Employee, FulltimeEmployee, HourlyEmployee, and Payroll.

To structure the program, we’ll use [modules](https://www.pythontutorial.net/python-basics/python-module/), where each class is placed in a separate module (or file).

### **The Employee class**

The Employee class represents an employee, either full-time or hourly. The Employee class should be an abstract class because there’re only full-time employees and hourly employees, no general employees exist.

The Employee class should have a property that returns the full name of an employee. In addition, it should have a method that calculates salary. The method for calculating salary should be an abstract method.

The following defines the Employee abstract class:

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 full\_name(self):

return f"{self.first\_name} {self.last\_name}"

@abstractmethod

def get\_salary(self):

pass

### **The FulltimeEmployee class**

The FulltimeEmployee class inherits from the Employee class. It’ll provide the implementation for the get\_salary() method.

Since full-time employees get fixed salaries, you can initialize the salary in the constructor of the class.

The following illustrates the FulltimeEmployee class:

class FulltimeEmployee(Employee):

def \_\_init\_\_(self, first\_name, last\_name, salary):

super().\_\_init\_\_(first\_name, last\_name)

self.salary = salary

def get\_salary(self):

return self.salary

### **The HourlyEmployee class**

The HourlyEmployee also inherits from the Employee class. However, hourly employees get paid by working hours and their rates. Therefore, you can initialize this information in the constructor of the class.

To calculate the salary for the hourly employees, you multiply the working hours and rates.

The following shows the HourlyEmployee class:

class HourlyEmployee(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

def get\_salary(self):

return self.worked\_hours \* self.rate

## **The Payroll class**

The Payroll class will have a method that adds an employee to the employee list and print out the payroll.

Since fulltime and hourly employees share the same interfaces (full\_time property and get\_salary() method). Therefore, the Payroll class doesn’t need to distinguish them.

The following shows the Payroll class:

## **The Payroll class**

The Payroll class will have a method that adds an employee to the employee list and print out the payroll.

Since fulltime and hourly employees share the same interfaces (full\_time property and get\_salary() method). Therefore, the Payroll class doesn’t need to distinguish them.

The following shows the Payroll class:

class Payroll:

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

self.employee\_list = []

def add(self, employee):

self.employee\_list.append(employee)

def print(self):

for e in self.employee\_list:

print(f"{e.full\_name} \t ${e.get\_salary()}")

### **The main program**

The following app.py uses the FulltimeEmployee, HourlyEmployee, and Payroll classes to print out the payroll of five employees.

from fulltimeemployee import FulltimeEmployee

from hourlyemployee import HourlyEmployee

from payroll import Payroll

payroll = Payroll()

payroll.add(FulltimeEmployee('Tan', 'Dow', 6000))

payroll.add(FulltimeEmployee('Wan', 'De', 6500))

payroll.add(HourlyEmployee('Joseph', 'Smith', 200, 50))

payroll.add(HourlyEmployee('David', 'Wilson', 150, 100))

payroll.add(HourlyEmployee('Kevin', 'Miller', 100, 150))

payroll.print()

Output: Tan Dow $6000

Wan De $6500

Jenifer Smith $10000

David Wilson $15000

Kevin Miller $15000