**Q1. What is the relationship between classes and modules?**

Modules are collections of methods and constants. They cannot generate instances. Classes may generate instances (objects), and have per-instance state (instance variables).

Modules may be mixed in to classes and other modules. The mixed in module’s constants and methods blend into that class’s own, augmenting the class’s functionality. Classes, however, cannot be mixed in to anything.

A class may inherit from another class, but not from a module.

A module may not inherit from anything.

This does not generate a subclass (which would require inheritance), but does generate an is\_a? relationship between the class and the module.

class Animal

include Comparable

attr\_reader :legs

def initialize(name, legs)

@name, @legs = name, legs

end

def <=>(other)

legs <=> other.legs

end

def inspect

@name

end

end

c = Animal.new("cat", 4)

s = Animal.new("snake", 0)

p = Animal.new("parrot", 2)

c < s # => false

s < c # => true

p >= s # => true

p.between?(s, c) # => true

[p, s, c].sort # => [snake, parrot, cat]

**Q2. How do you make instances and classes?**

Creating Instance Objects

To create instances of a class, you call the class using class name and pass in whatever arguments its *\_\_init\_\_* method accepts.

"This would create first object of Employee class"

emp1 = Employee("Zara", 2000)

"This would create second object of Employee class"

emp2 = Employee("Manni", 5000)

It can access the attributes

class Employee:

'Common base class for all employees'

empCount = 0

def \_\_init\_\_(self, name, salary):

self.name = name

self.salary = salary

Employee.empCount += 1

def displayCount(self):

print "Total Employee %d" % Employee.empCount

def displayEmployee(self):

print "Name : ", self.name, ", Salary: ", self.salary

"This would create first object of Employee class"

emp1 = Employee("Zara", 2000)

"This would create second object of Employee class"

emp2 = Employee("Manni", 5000)

emp1.displayEmployee()

emp2.displayEmployee()

print "Total Employee %d" % Employee.empCount

**Q3. Where and how should be class attributes created?**

**Class attributes:** are the variables defined directly in the class that are shared by all objects of the class.

**Instance attributes:** are attributes or properties attached to an instance of a class. Instance attributes are defined in the constructor.

Instance attributes are owned by the specific instances of a class. That is, for two different instances, the instance attributes are usually different.

**class** **A**:

a = "I am a class attribute!"

x = A()

y = A()

x.a

#### OUTPUT:

'I am a class attribute!'

y.a

**Q4. Where and how are instance attributes created?**

**Class attributes** are the variables defined directly in the class that are shared by all objects of the class.

**Instance attributes** are attributes or properties attached to an instance of a class. Instance attributes are defined in the constructor.

Example: Passing Instance Attribute Values in Constructor

 Copy

>>> std = Student('Bill',25)

>>> std.name

'Bill'

>>> std.age

25

>>> std.name = 'Steve'

>>> std.age = 45

>>> std.name

'Steve'

>>> std.age

45

**Q5. What does the term "self" in a Python class mean?**

The self parameter is **a reference to the current instance of the class**, and is used to access variables that belongs to the class.

### **Example**

Use the words mysillyobject and abc instead of self:

class Person:  
  def \_\_init\_\_(mysillyobject, name, age):  
    mysillyobject.name = name  
    mysillyobject.age = age  
  
  def myfunc(abc):  
    print("Hello my name is " + abc.name)  
  
p1 = Person("John", 36)  
p1.myfunc()

**Q6. How does a Python class handle operator overloading?**

The operator overloading in Python means provide extended meaning beyond theirpredefined operational meaning. Such as, we use the "+" operator for adding two integers as well as joining two strings or merging two lists. We can achieve this as the "+" operator is overloaded by the "int" class and "str" class.

**Example:**

1. print (14 + 32)
3. # Now, we will concatenate the two strings
4. print ("Java" + "Tpoint")
6. # We will check the product of two numbers
7. print (23 \* 14)
9. # Here, we will **try** to repeat the String
10. print ("X Y Z " \* 3)

**Q7. When do you consider allowing operator overloading of your classes?**

The user can notice that the same inbuilt operator or function is showing different behaviour for objects of different classes. This process is known as operator overloading. Example: print (14 + 32)

**Example:**

1. **class** example:
2. def \_\_init\_\_(self, X):
3. self.X = X
5. # adding two objects
6. def \_\_add\_\_(self, U):
7. **return** self.X + U.X
8. object\_1 = example( **int**( input( print ("Please enter the value: "))))
9. object\_2 = example( **int**( input( print ("Please enter the value: "))))
10. print (": ", object\_1 + object\_2)
11. object\_3 = example(str( input( print ("Please enter the value: "))))
12. object\_4 = example(str( input( print ("Please enter the value: "))))
13. print (": ", object\_3 + object\_4)

**Q8. What is the most popular form of operator overloading?**

A very popular and convenient example is the Addition (+) operator. Just think how the '+' operator operates on two numbers and the same operator operates on two strings. It performs “Addition” on numbers whereas it performs “Concatenation” on strings.

**class** bubble:

**def** \_\_init\_\_(self, volume):

self.volume = volume

We have defined a class bubble which has an attribute volume. Let’s see what happens when we combine two bubbles. We should get the volume of the combined bubble.

>>> b1 = bubble(20)

>>> b2 = bubble(30)

And now let’s add b1 and b2 to merge the bubbles.

>>> b1 + b2

**Q9. What are the two most important concepts to grasp in order to comprehend Python OOP code?**

we will elaborate on two key concepts of OOP which are **inheritance and polymorphism**. Both inheritance and polymorphism are key ingredients for designing robust, flexible, and easy-to-maintain software. These concepts are best explained via examples. Let's start with creating a simple class

class Employee(): def \_\_init\_\_(self, emp\_id, salary):  
 self.emp\_id = emp\_id  
 self.salary = salary def give\_raise(self):  
 self.salary = self.salary \* 1.05

We have created a class called Employee. It has two data attributes which are employee id (emp\_id) and salary. We have also defined a method called give\_raise. It applies a 5-percent increase on the salary of an employee.

emp1 = Employee(1001, 56000)print(emp1.salary)  
56000emp1.give\_raise()print(emp1.salary)  
58800.0

OOP allows us to create a class based on another class. For instance, we can create the Manager class based on the Employee class.

class Manager(Employee):  
 pass

mgr1 = Manager(101, 75000)  
print(mgr1.salary)  
75000

class Manager(Employee): def give\_raise(self):  
 self.salary = self.salary \* 1.10mgr1 = Manager(101, 75000)  
print(mgr1.salary)  
75000mgr1.give\_raise()  
print(mgr1.salary)  
82500

class Director(Employee): def give\_raise(self):  
 self.salary = self.salary \* 1.20

def bulk\_raise(list\_of\_emps):  
for emp in list\_of\_emps:  
emp.give\_raise()

emp1 = Employee(101, 45000)  
emp2 = Manager(103, 60000)  
emp3 = Director(105, 70000)list\_of\_emps = [emp1, emp2, emp3]

bulk\_raise(list\_of\_emps)print(emp1.salary)  
47250.0print(emp2.salary)  
66000.0print(emp3.salary)  
84000.0

## Conclusion

Both inheritance and polymorphism are fundamental concepts of object oriented programming. These concepts help us to create code that can be extended and easily maintainable.

Inheritance is a great way to eliminate unnecessary repetitive code. A child class can inherit from the parent class partially or entirely. Python is quite flexible with regards to inheritance. We can add new attributes and methods as well as modify the existing ones.

Polymorphism contributes to Python’s flexibility as well. An object with a particular type can be used as if it belonged to a different type. We have seen an example of it with the give\_raise method.