Q1. Define the relationship between a class and its instances. Is it a one-to-one or a one-to-many partnership, for example?

Ans:- *Class is abstraction of an real world entity. It consists of attributes and methods. Instance is an object of a class. It one to many relationship between class and its instances.*

Q2. What kind of data is held only in an instance?

Ans:- *Instance objects contains the Instance variables which are specific to that specific Instance object.*

Q3. What kind of knowledge is stored in a class?

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

Q4. What exactly is a method, and how is it different from a regular function?

Ans:- *The methods with a class can be used to access the insatnce variables of its instance. So, the object's state can be modified by its method.Function cant access the attributes of an instance of a class or cant modify the state of the object.*

Q5. Is inheritance supported in Python, and if so, what is the syntax?

Ans:- *: Inheritance is supported by python*

*# Example of Inheritance:*

**class** A:

var**=**1

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

**pass**

**class** B(A): *# class B is detived from class A*

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

super()**.**\_\_init\_\_()

c**=**B()

print("Class of Instance:",c**.**\_\_class\_\_)

print("Base class:",c**.**\_\_class\_\_**.**\_\_bases\_\_)

Class of Instance: <class '\_\_main\_\_.B'>

Base class: (<class '\_\_main\_\_.A'>,)

Q6. How much encapsulation (making instance or class variables private) does Python support?

Ans:- *Encapsulation prevents from accessing accidentally, but not intentionally. The private attributes and methods are not really hidden. The private attributes can be accessed within the object method.*

Q7. How do you distinguish between a class variable and an instance variable?

Ans:- *The class attribute is available to all the instance objects of that class. Instance variable is accessible only to the object or Instance of that class.*

Q8. When, if ever, can self be included in a class's method definitions?

Ans:- *self can included to access the class variables and instance variiables.*

Q9. What is the difference between the \_ \_add\_ \_ and the \_ \_radd\_ \_ methods?

Ans:- *When you add two numbers using the + operator, internally, the \_\_add\_\_() method will be called. We can overload this method to perform*

Q10. When is it necessary to use a reflection method? When do you not need it, even though you support the operation in question?

ANS:- *Suppose we are implementing a class that you want to act like a number via operator overloading.So we implement*

*# \_\_add\_\_ in your class, and now expressions like obj + 10 is acceptable.This is because obj + 10 is interpreted*

*# as obj.\_\_add\_\_(10), and the custom method \_\_add\_\_ can do whatever it means to add 10 to custom class.*

*# However, what about an expression like 10 + obj which is really (10).\_\_add\_\_(myobj)?*

*# The 10 is an instance of a Python built-in type and its \_\_add\_\_ method doesn't know anything*

*# about the new type,obj, so it will return a error NotImplemented.*

*# To handle such scenarios, \_\_radd\_\_ is used. Python will first try (10).\_\_add\_\_(myobj),*

*# and if that returns NotImplemented, Python will check if the right-hand operand implements*

*# \_\_radd\_\_, and if it does, it will call obj.\_\_radd\_\_(10) rather than raising a TypeError.*

Q11. What is the \_ \_iadd\_ \_ method called?

Ans:- *\_\_iadd\_\_ method is called when we use implementation like a+=b which is a.\_\_iadd\_\_(b)*

**class** A:

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

self**.**x**=**x

**def** \_\_iadd\_\_(self,other):

self**.**x **+=** other**.**x

**return** self**.**x

obj1**=**A(2)

obj2**=**A(3)

obj1**+=**obj2

print(obj1)

5

Q12. Is the \_ \_init\_ \_ method inherited by subclasses? What do you do if you need to customize its behavior within a subclass?

Ans:- *\_\_init\_\_ method is inherited by its subclass. But it can be overloaded, to customize it*

**class** A:

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

self**.**x**=**x

**class** B(A):

**pass**

obj**=**B(2)

obj**.**x

*# here the value x is accessible to instance of class B which is subclass of class A.This means*

*# \_\_init\_\_ of class A is inherited in sub class B*

**class** C(A):

**def** \_\_init\_\_(self,x,y): *# Here we are overloading the \_\_init\_\_ inherited from class A*

self**.**x**=**x

self**.**y**=**y

**def** func(self):

**return**(self**.**x **+** self**.**y)

obj1**=**C(3,4)

obj1**.**func()

7