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: In object-oriented programming, a class is a blueprint for creating objects, while an instance is an object created from that class. The relationship between a class and its instances is a one-to-many partnership, where one class can have many instances created from it.

When a new instance is created from a class, it inherits all the attributes and methods defined by that class. Each instance has its own set of values for the attributes defined by the class, which can be manipulated and accessed independently of other instances.

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

Ans:

a. **State data**: Data that represents the current state of an instance, such as its position or velocity.

b. **Configuration data**: Data that specifies how an instance should behave, such as the size of a window or the number of threads to use.

c. **User-specific data**: Data that is specific to a user of an application or system, such as their preferences or settings.

d. **Temporary data**: Data that is needed only temporarily during the execution of a method or function, such as intermediate results or loop counters.

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

Ans: Attributes or properties, Methods or functions, Constructors, Inheritance, Polymorphism, Access modifiers

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

Ans: Access to class attributes, Inheritance, Polymorphism, Code organization

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

Ans: Yes, inheritance is supported in Python. Inheritance allows a subclass to inherit attributes and methods from a parent or superclass, and then modify or extend them as needed. In Python, the syntax for creating a subclass that inherits from a superclass is as follows:

class Superclass:

# define attributes and methods of the superclass

class Subclass(Superclass):

# define attributes and methods of the subclass

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

Ans: Python supports a limited form of encapsulation for instance and class variables using naming conventions. Specifically, it supports a convention for indicating that a variable is intended to be private by prefixing its name with two underscores (\_\_).

In Python, when a variable is prefixed with two underscores, its name is "mangled" to prevent accidental access from outside the class. This means that the variable's name is changed to include the name of the class as a prefix, which makes it more difficult to access the variable from outside the class. However, the variable can still be accessed from outside the class if its mangled name is known.

Here is an example of using double underscores to indicate private variables in Python:

class MyClass:

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

self.\_\_private\_var = 10

self.\_semi\_private\_var = 20

obj = MyClass()

# Accessing private variable using its mangled name

print(obj.\_MyClass\_\_private\_var) # Output: 10

# Accessing semi-private variable directly

print(obj.\_semi\_private\_var) # Output: 20

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

Ans: In Python, a class variable is a variable that is shared by all instances of a class, while an instance variable is a variable that is unique to each instance of a class.

A class variable is defined inside the class definition but outside any method definition. It is accessed using the class name rather than an instance of the class. A class variable is shared by all instances of the class, and any changes made to it will affect all instances of the class. Here is an example of a class variable:

class MyClass:

class\_variable = 10

obj1 = MyClass()

obj2 = MyClass()

print(obj1.class\_variable) # Output: 10

print(obj2.class\_variable) # Output: 10

MyClass.class\_variable = 20

print(obj1.class\_variable) # Output: 20

print(obj2.class\_variable) # Output: 20

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

Ans: In Python, self is a special variable that refers to the instance of a class, and it is used to access the instance variables and methods of the class.

self should always be included as the first parameter in a class's method definitions, with the exception of class methods and static methods (which use cls and @staticmethod, respectively).

For example, consider the following class definition:

class MyClass:

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

self.x = x

self.y = y

def method1(self):

print("Instance method called")

@classmethod

def method2(cls):

print("Class method called")

@staticmethod

def method3():

print("Static method called")

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

Ans: So, the difference between x.\_\_add\_\_(y) and x.\_\_radd\_\_(y) is that the former calculates x + y whereas the latter calculates y + x — both calling the respective method defined on the object x.

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: It is necessary to use a reflection method when you want to perform some operation on an object or class, but you don't know the type of the object or class at compile time. Reflection methods allow you to determine the type of an object or class at runtime and then perform the appropriate operation.

For example, if you have a function that accepts any object as an argument, you can use the isinstance reflection method to determine the type of the object and then perform the appropriate operation based on its type.

However, in some cases, it may not be necessary to use a reflection method even though you support the operation in question. For example, if you know the type of an object at compile time, you can call its methods directly, without using reflection methods. This can be more efficient and easier to read than using reflection methods.

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

Ans: In Python, the \_\_iadd\_\_ method is called when the "+=" operator is used with an instance of a class. It is used to define the behavior of the "+=" operator for that class.

The \_\_iadd\_\_ method takes two arguments: self and other, where self is the instance of the class on the left-hand side of the "+=" operator, and other is the value on the right-hand side of the "+=" operator.

Here is an example:

class MyClass:

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

self.value = value

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

self.value += other.value

return self

a = MyClass(10)

b = MyClass(20)

a += b

print(a.value) # Output: 30

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

Ans: Yes, the \_\_init\_\_ method is inherited by subclasses in Python. When a subclass is created, it inherits all the methods and attributes of its parent class, including the \_\_init\_\_ method.

If we need to customize the behavior of the \_\_init\_\_ method within a subclass, we can define a new \_\_init\_\_ method in the subclass with the desired behavior. This new method will override the \_\_init\_\_ method inherited from the parent class, and the subclass instances will use the new method instead.

Here is an example:

class Animal:

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

self.name = name

self.species = species

class Cat(Animal):

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

super().\_\_init\_\_(name, species='Cat')

self.breed = breed

my\_cat = Cat('Kitty', 'Siamese')

print(my\_cat.name) # Output: Kitty

print(my\_cat.species) # Output: Cat

print(my\_cat.breed) # Output: Siamese