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

In object-oriented programming, a class is a blueprint or template for creating objects. Each object that is created from a class is called an instance of that class. In this sense, a class can be thought of as a "one-to-many" relationship, because one class can be used to create many objects or instances.

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

In object-oriented programming, an instance is a specific realization of a class. This means that each instance has its own unique state, which is determined by the values of its attributes or properties. The values of an instance's attributes are specific to that particular instance, and are not shared with other instances of the same class. Therefore, data that is specific to a single instance of a class is held only in that instance.

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

In object-oriented programming, a class is a blueprint or template for creating objects. A class defines the attributes and behaviors that are common to all objects that are created from that class. This means that a class stores knowledge about the properties or characteristics that are shared by all objects that are instances of that class. For example, a class called "Dog" might define attributes such as "breed" and "color", as well as behaviors such as "bark" and "wag tail". This knowledge is stored in the class and is used to create specific instances of dogs with unique values for these attributes.

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

In object-oriented programming, a method is a function that is associated with a particular class. A method is similar to a regular function in that it takes some input, performs some computation, and returns some output. However, a method is different from a regular function in that it has access to the attributes and behaviors of the class to which it belongs. This means that a method can use the data stored in an instance of the class to perform its computation, and can also modify the state of the instance.

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

Yes, inheritance is supported in Python. Inheritance is a mechanism in object-oriented programming that allows a class to inherit the attributes and behaviors of another class, known as the superclass or base class. This allows the class that is inheriting, known as the subclass or derived class, to reuse the functionality of the superclass and modify or extend it as needed.

In Python, inheritance is specified using the syntax **class SubClassName(SuperClassName),** where SubClassName is the name of the subclass and SuperClassName is the name of the superclass.

For example, the following code defines a Dog class that inherits from a Mammal class:

class Mammal:

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

self.name = name

class Dog(Mammal):

def bark(self):

print("Woof! Woof!")

In this example, the Dog class inherits from the Mammal class, which means that it has access to the name attribute and the \_\_init\_\_ method of the Mammal class. The Dog class also defines its own bark method, which is specific to instances of the Dog class.

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

Python supports encapsulation by allowing you to make a class or instance variable private. To make a class or instance variable private in Python, you can simply prefix its name with an underscore (\_). This indicates that the variable should not be accessed directly from outside the class.

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

In Python, a class variable is a variable that is shared among all instances of a class. An instance variable, on the other hand, is a variable that belongs to a specific instance of a class.

To distinguish between a class variable and an instance variable, you need to consider where the variable is defined and how it is accessed. A class variable is defined directly within the class definition, and it can be accessed either through an instance of the class or through the class itself. An instance variable, on the other hand, is defined within a method of the class, and it can only be accessed through a specific instance of the class.

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

By using the “self” we can access the attributes and methods of the class in python. It binds the attributes with the given arguments.

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

\_\_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?**

Reflection refers to the ability for code to be able to examine attributes about objects that might be passed as parameters to a function.

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

\_\_add\_\_ magic method is used to add the attributes of the class instance. For example, let's say object1 is an instance of a class A and object2 is an instance of class B and both of these classes have an attribute called 'a', that holds an integer

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

Yes, the \_\_init\_\_ method is inherited by subclasses in Python. If you need to customize the \_\_init\_\_ method's behavior within a subclass, you can override it by defining a new \_\_init\_\_ method in the subclass that includes the customized behavior you want.

For example:

***class BaseClass:***

***def \_\_init\_\_(self, arg1, arg2):***

***# Initialize base class attributes here***

***self.arg1 = arg1***

***self.arg2 = arg2***

***class SubClass(BaseClass):***

***def \_\_init\_\_(self, arg1, arg2, arg3):***

***# Initialize subclass attributes here***

***self.arg3 = arg3***

***# Call the base class's \_\_init\_\_ method to initialize***

***# base class attributes***

***super().\_\_init\_\_(arg1, arg2)***

In the SubClass example above, the \_\_init\_\_ method is overridden to accept an additional argument (arg3) and to initialize a new attribute (self.arg3) with this argument. The super().\_\_init\_\_(arg1, arg2) call at the end of the method is used to call the \_\_init\_\_ method of the base class (BaseClass), which initializes the attributes arg1 and arg2 in the SubClass instance.