Q1. Which two operator overloading methods can you use in your classes to support iteration?

The two operator overloading methods that you can use in your classes to support iteration are:

1. \_\_iter\_\_: This method allows an object to be iterable. It should return an iterator object, which is an object that defines the \_\_next\_\_ method. The \_\_iter\_\_ method is called when the iter() function is called on the object. It is typically implemented in classes that represent a collection of items, such as lists or custom data structures.
2. \_\_next\_\_: This method is used by the iterator object to return the next item in the iteration sequence. It should raise the StopIteration exception when there are no more items to iterate over. The \_\_next\_\_ method is called each time the next() function is called on the iterator object.

Q2. In what contexts do the two operator overloading methods manage printing?

The two operator overloading methods that manage printing in different contexts are:

1. \_\_str\_\_: This method is responsible for defining a string representation of an object. It is called by the str() function and by the built-in print() function when an object is passed as an argument to it. The \_\_str\_\_ method should return a string that represents the object in a human-readable format.
2. \_\_repr\_\_: This method is responsible for defining a string representation of an object that can be used to recreate the object. It is called by the repr() function and by the interactive interpreter when the object is evaluated. The \_\_repr\_\_ method should return a string that represents the object in a format that can be evaluated as valid Python code.

By implementing these two methods in your class, you can control how the object is displayed or printed in different contexts. The \_\_str\_\_ method is used for a more user-friendly representation, while the \_\_repr\_\_ method is used for a more detailed and unambiguous representation of the object.

Q3. In a class, how do you intercept slice operations?

class MyClass:

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

self.data = data

def \_\_getitem\_\_(self, index):

if isinstance(index, slice):

# Handle slice operations

start, stop, step = index.indices(len(self.data))

sliced\_data = self.data[start:stop:step]

return sliced\_data

else:

# Handle single item access

return self.data[index]

# Create an instance of MyClass

my\_obj = MyClass([1, 2, 3, 4, 5, 6, 7, 8, 9])

# Accessing elements using indexing

print(my\_obj[2]) # Output: 3

# Accessing elements using slicing

print(my\_obj[2:7:2]) # Output: [3, 5, 7]

Q4. In a class, how do you capture in-place addition?

class MyClass:

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

self.value = value

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

if isinstance(other, MyClass):

self.value += other.value

elif isinstance(other, int):

self.value += other

else:

raise TypeError("Unsupported operand for +=: {}".format(type(other)))

return self

# Create instances of MyClass

obj1 = MyClass(5)

obj2 = MyClass(10)

# Perform in-place addition using the += operator

obj1 += obj2

print(obj1.value) # Output: 15

# Perform in-place addition with an integer

obj1 += 5

print(obj1.value) # Output: 20

Q5. When is it appropriate to use operator overloading?

It is appropriate to use operator overloading when you want to define custom behaviors for operators in your classes. Operator overloading allows you to redefine how operators such as +, -, \*, /, ==, <, >, and many others, work with instances of your custom classes.