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

\_\_iter\_\_ - to convert not iterable value to iterable ex : **string** ,

\_\_next\_\_ is used to move on to the next value inside the iterable value

Ex :

class Counter:

def \_\_init\_\_(self, low, high):

self.current = low

self.high = high

def \_\_iter\_\_(self):

return self

def \_\_next\_\_(self):

if self.current > self.high:

raise StopIteration

else:

self.current += 1

return self.current - 1

for num in Counter(5, 15):

print(num)

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

The \_\_str\_\_**method** in Python represents the class objects as a string – it can be used for classes, The \_\_str\_\_ method is called when the functions print() or str() are invoked on the object and return a string, If we have not defined the \_\_str\_\_, then it will call the \_\_repr\_\_ method. The \_\_repr\_\_**method** returns a string that describes the pointer of the object by default

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

**The slice() method returns a portion of an iterable as an object of the slice class based on the specified range**. It can be used with string, list, tuple, set, bytes, or range objects or custom class object that implements sequence methods \_\_getitem\_\_() and \_\_len\_\_() methods.

|  |
| --- |
| class Demo:      def \_\_getitem\_\_(self, key):                   print(key)            return key  a = Demo()    a[1]   a[1, 2]   a[1, 2, 3] |

Output

1

(1, 2)

(1, 2

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

To use the in-place addition operator += on custom objects, you need to define the \_\_iadd\_\_() method (*“dunder method”, “magic method”*) that takes two arguments self and other, updates the first argument self with the result of the addition, and returns the updated object.

class Data:

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

self.data = data

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

self.data += other.data

return self

x = Data(40)

y = Data(2)

x += y

print(x.data)

**Q5. When is it appropriate to use operator overloading?**

Operator overloading in Python is the ability of a single operator to perform more than one operation based on the class (type) of operands , so when we need to give extended meaning beyond their predefined operational meaning we can go with Operator overloading.