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

**Answer:**  
The two methods are:

1. \_\_iter\_\_() → Returns an iterator object (usually self).
2. \_\_next\_\_() → Defines how to fetch the next value in iteration.

**Example:**

class Counter:

def \_\_init\_\_(self, start, end):

self.current = start

self.end = end

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

return self # The class itself is the iterator

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

if self.current >= self.end:

raise StopIteration

self.current += 1

return self.current - 1

counter = Counter(1, 5)

for num in counter:

print(num) # Output: 1, 2, 3, 4

* \_\_iter\_\_() makes the class iterable.
* \_\_next\_\_() defines the iteration logic.

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

**Answer:**  
The two methods are:

1. \_\_str\_\_() → Used by print() and str(), meant for user-friendly string representation.
2. \_\_repr\_\_() → Used by repr() and debugging, meant for an official representation of the object.

**Example:**

class Person:

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

self.name = name

def \_\_str\_\_(self):

return f"Person: {self.name}" # User-friendly

def \_\_repr\_\_(self):

return f"Person('{self.name}')" # Debug-friendly

p = Person("Alice")

print(p) # Output: Person: Alice (\_\_str\_\_)

print(repr(p)) # Output: Person('Alice') (\_\_repr\_\_)

* \_\_str\_\_() is for end-users.
* \_\_repr\_\_() is for developers.

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

**Answer:**

* Use \_\_getitem\_\_() to handle slicing (obj[start:stop:step]).
* Use \_\_setitem\_\_() to modify slices (obj[start:stop] = values).

**Example:**

class MyList:

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

self.items = items

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

return self.items[index] # Supports slicing

def \_\_setitem\_\_(self, index, value):

self.items[index] = value # Modify slice

lst = MyList([0, 1, 2, 3, 4, 5])

print(lst[1:4]) # Output: [1, 2, 3]

lst[1:3] = [9, 9]

print(lst.items) # Output: [0, 9, 9, 3, 4, 5]

* \_\_getitem\_\_() retrieves slice values.
* \_\_setitem\_\_() modifies slice values.

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

**Answer:**

* Implement \_\_iadd\_\_() for += (in-place addition).
* If \_\_iadd\_\_() is not defined, Python falls back to \_\_add\_\_().

**Example:**

class Number:

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

self.value = value

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

self.value += other

return self

n = Number(5)

n += 10 # Calls \_\_iadd\_\_

print(n.value) # Output: 15

* \_\_iadd\_\_() modifies the object in place.

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

**Answer:**  
Use operator overloading when:

1. **Enhancing readability** → E.g., using + for vector addition.
2. **Working with custom objects** → E.g., defining \_\_eq\_\_() for comparisons.
3. **Mimicking built-in behavior** → E.g., overloading \_\_getitem\_\_() for index access.
4. **Providing a natural interface** → E.g., \_\_str\_\_() for user-friendly printing.

**Example:**

class Vector:

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

self.x, self.y = x, y

def \_\_add\_\_(self, other): # Overload +

return Vector(self.x + other.x, self.y + other.y)

v1 = Vector(2, 3)

v2 = Vector(4, 5)

v3 = v1 + v2 # Calls \_\_add\_\_

print(v3.x, v3.y) # Output: 6 8

* Overloading + makes vector addition intuitive.

**Summary:**

| **Feature** | **Method** |
| --- | --- |
| **Iteration** | \_\_iter\_\_(), \_\_next\_\_() |
| **Printing** | \_\_str\_\_(), \_\_repr\_\_() |
| **Slicing** | \_\_getitem\_\_(), \_\_setitem\_\_() |
| **In-place Addition** | \_\_iadd\_\_() |
| **Operator Overloading Usage** | Enhances readability & usability |