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

* 1) `\_\_iter\_\_` = Implement this method to make your class iterable by defining the logic for iterating over its elements.
* 2) `\_\_next\_\_` = (or `\_\_getitem\_\_` for sequence-like iteration): Implement this method to define the behavior when fetching the next element in the iteration sequence.

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

* 1. `\_\_str\_\_`: This method is used to define the "informal" or user-friendly string representation of an object when it is printed using the `print()` function or when displayed as a string. It is intended for end-users and readability.
* 2. `\_\_repr\_\_`: This method is used to define the "formal" or developer-friendly string representation of an object. It is displayed when an object is entered directly into the interactive interpreter or when using the `repr()` function. It should provide a detailed and unambiguous representation for debugging and development purposes.

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

* To intercept slice operations in a class, we can implement the `\_\_getitem\_\_` method with appropriate handling for slicing. This method allows you to define the behavior when an instance of your class is accessed using slicing notation (e.g., `my\_instance[start:stop:step]`).

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

* To capture in-place addition in a class, we can implement the \_\_iadd\_\_ method in that class.

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

* Operator overloading is appropriate when we want to define custom behaviours for built-in operators like `+`, `-`, `\*`, `/`, `==`, `!=`, etc., to work with objects of custom class, providing more intuitive and meaningful operations. It's useful when the overloaded operator enhances code readability and maintains consistency in your class's interface**.**