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

Sol. Classes can support iteration by defining (or inheriting) \_\_getitem\_\_ or \_\_iter\_\_.In all iteration contexts, Python tries to use \_\_iter\_\_ (which returns an object that supports the iteration protocol with a \_\_next\_\_ method) first: if no \_\_iter\_\_ is found by inheritance search, Python falls back on the \_\_getitem\_\_ indexing method (which is called repeatedly, with successively higher indexes).

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

Sol. The \_\_str\_\_ and \_\_repr\_\_ methods implement object print displays. The former is called by the print and str built-in functions; the latter is called by print and str if there is no \_\_str\_\_, and always by the repr built-in, interactive echoes, and nested appearances. That is, \_\_repr\_\_ is used everywhere, except by print and str when a \_\_str\_\_ is defined. A \_\_str\_\_ is usually used for user-friendly displays; \_\_repr\_\_ gives extra details or the object’s as-code form.

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

Sol. Slicing is caught by the \_\_getitem\_\_ indexing method: it is called with a slice object, instead of a simple index. \_\_getslice\_\_ may be used as well.

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

Sol. In-place addition tries \_\_iadd\_\_ first, and \_\_add\_\_ with an assignment second. The same pattern holds true for all binary operators. The \_\_radd\_\_ method is also available for right-side addition

Q5. When is it appropriate to use operator overloading?

Sol. When a class naturally matches, or needs to emulate, a built-in type’s interfaces. For example, collections might imitate sequence or mapping interfaces. You generally shouldn’t implement expression operators if they don’t naturally map to your objects, though—use normally named methods instead.