**Que 1: Understanding how generators work in Python.**

A **generator** is a **function** that:

* **Yields** values one at a time using the yield keyword
* **Remembers its state** between each call
* Is **used to create iterators** — but in a memory-efficient way

**Why use Generators?**

1. **Saves memory** – doesn’t store all values in memory at once
2. **Fast for large data** – generates data when needed ("lazy evaluation")
3. **Clean syntax** – easier to write and read for streaming data

**Que 2: Difference between yield and return.**

**return:**

* Ends the function.
* Gives **one final value**.
* Can't continue after return.
* Used in **normal functions**.

**yield:**

* Pauses the function.
* Gives **one value at a time**.
* Can continue from where it stopped.
* Used in **generator functions**.

**Que 3 : Understanding iterators and creating custom iterators.**

In Python, an **iterator** is an object that helps you loop through a collection, like a list or string, one item at a time. To be an iterator, the object must have two special methods: \_\_iter\_\_() and \_\_next\_\_(). The \_\_iter\_\_() method returns the iterator object itself, and the \_\_next\_\_() method returns the next value in the sequence. When there are no more items, \_\_next\_\_() should raise a StopIteration error to stop the loop. Many Python objects like lists and tuples are already iterable, so you can use them in a for loop directly. But if you want to create your own custom iterator, you can do it using a class. Inside the class, you define \_\_iter\_\_() to return self, and \_\_next\_\_() to give the next value each time. This is useful when you want full control over how data is returned, like counting numbers, skipping values, or stopping at a certain point.