**6.Generators and Iterators**

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

Generators are a powerful feature in Python that allow you to create iterators in a more memory-efficient way.

**What are Generators?**

Generators are special functions that:

* Produce a sequence of values over time
* Maintain their state between iterations
* Don't store the entire sequence in memory at once

**Key Characteristics**

1. **Lazy Evaluation**: Values are generated on-the-fly as needed
2. **Memory Efficient**: Only one value exists in memory at a time
3. **Stateful**: Remembers where it left off between calls

**How Generators Work**

1. When you call a generator function, it returns a generator object (but doesn't execute the function yet)
2. The first next() call runs the function until it hits yield
3. The generator pauses at the yield statement, remembering its state **4.** The next next() call resumes execution right after the yield

1. **Difference between yield and return.**

The key distinction between yield and return lies in how they affect function execution and state preservation

**return Statement:**

1. Terminates the function immediately
2. Returns a single value to the caller
3. Forgets all local state (variables are discarded)
4. Can only be used once per function call
5. Standard function behavior - what you see in most functions

**yield Statement:**

1. Pauses the function temporarily
2. Returns a value but remembers where it left off
3. Preserves all local state between calls
4. Can be used multiple times in a generator function
5. Creates a generator instead of returning a single value

1. **Understanding iterators and creating custom iterators**

Iterators are objects that allow you to traverse through all elements of a collection (like lists, tuples, dictionaries, etc.) one at a time.

1. **Iterable** 
   * An object capable of returning its elements one at a time.
   * Must implement the \_\_iter\_\_() method.
   * Examples: lists, tuples, strings, dictionaries, sets, etc.
2. **Iterator** 
   * An object used to iterate over an iterable.
   * Must implement both:
     + \_\_iter\_\_() → returns the iterator object itself.
     + \_\_next\_\_() → returns the next item or raises StopIteration when no items are left.

A custom iterator allows you to define your own logic for iteration, especially useful when:

* You want to iterate over a custom data structure.
* You want to create controlled or lazy iteration (e.g., one item at a time, on demand).
* You want to implement infinite sequences or filtered views of data.

To create a custom iterator:

1. Define a class.
2. Implement the \_\_iter\_\_() method to return the iterator object.
3. Implement the \_\_next\_\_() method to define how the next value is calculated and when to stop (using StopIteration).