**Iterators**

An iterator is an object that represents a stream of data. It allows you to traverse through a sequence of elements one at a time. You’ve probably encountered iterators when using for loops in Python. Behind the scenes, the for loop calls the iter() function on the iterable (like a list or a string) to create an iterator, and then it repeatedly calls next() on that iterator until it’s exhausted.

Here are some key points about iterators:

1. **Lazy Evaluation**: Iterators are lazy—they generate values on-the-fly as you request them. This is especially useful for large datasets because it avoids loading everything into memory at once.
2. **Custom Iterators**: You can create your own custom iterators by defining classes with \_\_iter\_\_() and \_\_next\_\_() methods. The \_\_iter\_\_() method returns the iterator object itself, and the \_\_next\_\_() method provides the next value.
3. **Built-in Iterators**: Python provides several built-in iterators, such as range(), enumerate(), and zip(). These allow you to iterate over sequences efficiently.

**Generators**

Generators take the concept of iterators a step further. They are a concise way to create iterators without explicitly defining a class. Instead of using return to produce a value, generators use yield. When a generator function is called, it doesn’t execute immediately; it returns a generator object. Each time you call next() on the generator, it resumes execution from where it left off, yielding the next value.

Here’s why generators are awesome:

1. **Memory Efficiency**: Since generators produce values on-the-fly, they don’t store the entire sequence in memory. This is perfect for large datasets or infinite sequences.
2. **Clean Code**: Generators allow you to write clean, readable code. You can express complex logic in a concise manner.
3. **Infinite Sequences**: You can create generators for infinite sequences (like an infinite stream of Fibonacci numbers) without worrying about memory constraints.

**Modules for Functional Programming**

You mentioned functools and itertools, and they’re indeed powerful tools for functional programming:

* functools:
  + Contains utility functions like reduce(), partial(), and wraps().
  + reduce() applies a binary function cumulatively to the items of an iterable.
  + partial() allows you to fix certain arguments of a function, creating a new function with fewer parameters.
  + wraps() is a decorator that helps preserve function metadata (like docstrings) when creating wrappers.
* itertools:
  + Provides a set of fast, memory-efficient tools for working with iterators.
  + Includes functions like chain(), cycle(), combinations(), and permutations().
  + chain() combines multiple iterables into one.
  + cycle() creates an infinite iterator by cycling through a sequence.
  + combinations() and permutations() generate combinations and permutations, respectively.