**Assignment 6 Generators and Iterators**

**Understanding how generators work in Python**

Generators in Python are a special type of iterable that allow you to create an iterator in a very efficient way. Unlike regular functions that return a single value and then exit, generators use the yield keyword to return a value, but they can "pause" and "resume" their execution, which makes them memory-efficient for working with large data sets**.**

**1. Basic Concept:**

When a function contains the yield keyword, it becomes a **generator function**. Instead of returning a value and terminating, a generator function "yields" a series of values, one at a time, and it can continue from where it left off when the next value is requested.

### 2. ****How a Generator Works:****

* **yield Keyword:** When the generator function is called, it returns a **generator object**. This object does not immediately execute the function code. Instead, it begins execution only when iterated over (e.g., in a loop or via next()).

The yield statement suspends the function’s execution and sends a value to the caller. The state of the function is saved so that when the generator is resumed (next time next() is called), it continues from where it left off.

### 3. ****Key Properties of Generators:****

* **Memory Efficient:** Since generators yield values one by one, they do not store the entire sequence in memory. This is especially useful when working with large datasets.
* **Lazy Evaluation:** Generators compute values on-the-fly, meaning they only generate values when requested.
* **State Persistence:** Generators maintain their state between iterations, so the function doesn’t restart from the beginning.

### 4. ****Creating a Generator:****

A generator can be created in two ways:

1. **Generator Functions** – Using yield inside a function.
2. **Generator Expressions** – Similar to list comprehensions but using parentheses () instead of square brackets [].

### Example 1: Generator Function

def count\_up\_to(max):

count = 1

while count <= max:

yield count

count += 1

counter = count\_up\_to(5)

# The generator object does not start executing yet.

# You need to iterate over it or call next().

print(next(counter)) # 1

print(next(counter)) # 2

print(next(counter)) # 3

print(next(counter)) # 4

print(next(counter)) # 5

print(next(counter)) # StopIteration error, as the generator is exhausted

In this example:

* The function count\_up\_to generates numbers from 1 to max.
* Each time next() is called, the function resumes execution and yields the next number.
* After all values are yielded, the generator raises a StopIteration exception.

### Example 2: Generator Expression

You can create generators using a syntax similar to list comprehensions.

Code:

squares = (x \* x for x in range(1, 6))

for square in squares:

print(square)

**Output:**

1

4

9

16

25

Here, (x \* x for x in range(1, 6)) creates a generator expression that produces squares of numbers from 1 to 5. Like list comprehensions, but more memory-efficient.

### 5. ****Iterating Over Generators:****

You can iterate over a generator just like any other iterable object using a for loop or by using next() explicitly.

**Example:**

gen = count\_up\_to(3)

for num in gen:

print(num)

**output:**

1

2

3

The generator will be exhausted after this iteration and cannot be reused. You would need to create a new generator to get the values again.

### 6. ****Using**** yield ****for Infinite Sequences:****

Generators are particularly useful for generating infinite sequences, as they can continue generating values indefinitely without consuming excessive memory.

**Example:**

def infinite\_counter():

count = 1

while True:

yield count

count += 1

counter = infinite\_counter()

# You can limit the number of values produced using slicing or iteration limits

for i in range(5):

print(next(counter)) # Prints 1, 2, 3, 4, 5

### 7. ****Handling Generator States with**** send() ****and**** throw() ****Methods:****

Generators can also communicate with the code that is calling them, using the send() method to send values into the generator or the throw() method to raise exceptions inside the generator.

* **send(value)** – Sends a value into the generator, which resumes execution from the point of the last yield.
* **throw(exception)** – Raises an exception inside the generator function.

**Example**:

def echo():

while True:

value = (yield)

print(f"Received: {value}")

gen = echo()

next(gen) # Prime the generator (to start execution)

gen.send("Hello")

gen.send("World”)

**Difference between yield and return in tabular form**

| **Feature** | **yield** | **return** |
| --- | --- | --- |
| Definition | Used in a generator to produce values one at a time without terminating the function. | Exits the function and returns a value. |
| Execution | Function execution is paused and state is remembered. | Function execution stops immediately. |
| Return Type | Returns a generator object. | Returns the actual value or object. |
| Iteration | Can yield multiple values over time. | Can return only once per call. |
| Use Case | Suitable for large datasets or infinite sequences. | Suitable for immediate computation. |
| Memory Usage | Efficient, uses less memory (values are produced on demand). | May use more memory (stores entire result). |
| Syntax | yield value | return value |
| Result Access | Access using a loop or next(). | Access directly by calling the function. |
| State Retention | Retains local state between calls. | Does not retain state after returning. |

**Understanding iterators and creating custom iterators**

### Understanding Iterators in Python

In Python, **iterators** are objects that represent a stream of data. They allow you to traverse through a collection (like lists, tuples, etc.) one element at a time. An **iterator** must implement two methods:

1. **\_\_iter\_\_()**: This method should return the iterator object itself. It is required for an object to be iterable.
2. **\_\_next\_\_()**: This method returns the next item from the collection. When there are no more items, it raises the StopIteration exception, signaling that the iteration is complete.

### How Iterators Work in Python

* You can create an iterator object by calling the iter() function on any iterable (like a list or a tuple). This will return an iterator that allows you to step through the elements.

**Example:**

numbers = [1, 2, 3]

iter\_obj = iter(numbers) # Create an iterator object

print(next(iter\_obj)) # Output: 1

print(next(iter\_obj)) # Output: 2

print(next(iter\_obj)) # Output: 3

print(next(iter\_obj)) # Raises StopIteration exception

### Creating Custom Iterators

You can create a custom iterator by defining a class that implements the \_\_iter\_\_() and \_\_next\_\_() methods.

#### Steps to Create a Custom Iterator:

1. Define a class and implement the \_\_iter\_\_() method. This method must return self as the iterator object.
2. Implement the \_\_next\_\_() method. This method defines how to retrieve the next item in the sequence.
3. When there are no more items to iterate over, \_\_next\_\_() should raise StopIteration.

### Example of a Custom Iterator:

class SquareIterator:

def \_\_init\_\_(self, limit):

self.limit = limit

self.current = 1

def \_\_iter\_\_(self):

return self # The iterator object is the same as the object itself

def \_\_next\_\_(self):

if self.current <= self.limit:

result = self.current \*\* 2

self.current += 1

return result

else:

raise StopIteration # End of iteration

# Create an iterator instance with a limit of 5

squares = SquareIterator(5)

# Iterate through the squares

for square in squares:

print(square)

9

16

25

### Explanation:

* **\_\_init\_\_(self, limit)**: Initializes the iterator with a limit (the maximum number for which squares should be calculated).
* **\_\_iter\_\_(self)**: Returns the iterator object itself (in this case, the same object).
* **\_\_next\_\_(self)**: Calculates the next square, checks if the limit is reached, and raises StopIteration when the iteration is over.

### Why Use Iterators?

* **Memory Efficient**: Iterators don't store the entire collection in memory at once. They produce items one at a time, making them more memory-efficient.
* **Lazy Evaluation**: The next item is calculated only when needed (lazily), so you don't waste resources on unnecessary calculations.
* **Convenience**: They work well with loops and Python’s built-in functions like map(), filter(), and reduce().

### Use Cases for Custom Iterators:

* Generating an infinite sequence of data (e.g., Fibonacci numbers, prime numbers).
* Reading large files line by line without loading the entire file into memory.
* Custom data structures that require specific iteration behavior.