Understanding Python Generators: Memory Efficiency and Implementation

What is a Generator

Python generators are a simple way of creating iterators.

Custom range iterator implementation using separate iterable and iterator classes

Explanation

* The code defines a custom range implementation with two separate classes: mera_range (iterable) and mera_iterator (iterator) that follow Python's iteration protocol
* The mera_range class stores start and end values and implements __iter__ to return a new mera_iterator instance for the given range
* The mera_iterator class maintains a reference to the iterable object and implements __next__ to yield values one by one until reaching the end condition
* When the start value exceeds or equals the end value, the iterator raises StopIteration to signal completion of iteration
* This design demonstrates how to create custom iterable objects by separating the iteration logic from the data structure using the iterator pattern

```
# iterable
                                                                                class mera
```

The Why

Comparing Memory Usage Between Lists and Ranges in Python

Explanation

* The code initializes a list `L` containing integers from 0 to 99,999 and measures its memory size using `sys.getsizeof()`.
* It then creates a range object `x` for the same range of integers and checks its memory size, demonstrating that `range` uses significantly less memory than a list.
* The code further extends the range to 10 million integers and again measures its memory size, confirming that the memory usage remains minimal with `range`.
* The comments highlight the efficiency of iterators, as `range` generates elements on-the-fly rather than storing them all in memory like a list.
* This illustrates the advantage of using `range` for large sequences, especially in memory-constrained environments.

```
L = [x for x in range(100000)]
```

Output

```
800984
```

48

A Simple Example

This Python code defines a generator function that yields multiple statements sequentially.

Explanation

* The function `gen_demo` is defined as a generator using the `yield` keyword, which allows it to produce a series of values over time.
* Each call to the generator will return the next value in the sequence, allowing for efficient memory usage as values are generated on-the-fly.
* The three `yield` statements provide the strings 'first statement', 'second statement', and 'third statement' in that order when the generator is iterated over.
* This approach is useful for handling large datasets or streams of data where you do not want to load everything into memory at once.

```
def gen_demo():
```

This code snippet demonstrates the creation and printing of a generator object in Python.

Explanation

* The variable `gen` is assigned the result of calling the function `gen_demo()`, which is expected to return a generator.
* The `print(gen)` statement outputs the representation of the generator object, which typically includes its memory address.
* Generators are a type of iterable that allow you to iterate through a sequence of values without storing them all in memory at once.
* This code snippet does not execute the generator; it merely creates it and prints its

reference.

```
gen = gen_demo()
print(gen)
```

Output

```
<generator object gen_demo at 0x000002657740A8D0>
```

Sequentially retrieving values from a Python generator object

Explanation

* The code snippet calls the `next()` function four times on a generator object named `gen`.
* Each call to `next(gen)` retrieves the next value produced by the generator, advancing its internal state.
* If the generator is exhausted (i.e., all values have been yielded), a `StopIteration` exception will be raised on subsequent calls to `next()`.
* This approach is useful for processing items one at a time without loading all values into memory at once.

```
print(next(gen))
print(next
```

Output

```
first statement
second sta
```

Python generator function iteration and output demonstration

Explanation

* The code initializes a generator object by calling the gen_demo() function which yields values

one at a time
* The for loop iterates through each yielded value from the generator sequence
* Each value is printed to the console during each iteration of the loop
* This demonstrates lazy evaluation where values are produced on-demand rather than storing all values in memory at once
* The generator continues producing values until it's exhausted, at which point the loop terminates automatically

```
gen = gen_demo()
```

Output

```
first statement
                                                                                second sta
```

Python Tutor Demo (yield vs return)

Understanding the difference between yield and return in Python generators

Explanation

* The `yield` statement allows a function to produce a series of values over time, maintaining its state between calls.
* Unlike `return`, which exits the function and loses all local state, `yield` pauses the function and saves its context for the next invocation.
* This behavior makes `yield` suitable for creating iterators and handling large datasets efficiently without loading everything into memory at once.
* Generators can be iterated over, allowing for lazy evaluation and improved performance in scenarios where not all data is needed immediately.

```
# Yield remembers the previous value with the help of generators and continues from that
value. Where as return forgets the previous value altogether
```

Example 2

This code defines a generator function that yields the squares of numbers from 1 to a specified limit.

Explanation

* The function `square` takes a single argument `num`, which specifies the upper limit for generating squares.
* It uses a `for` loop to iterate through numbers starting from 1 up to and including `num`.
* The `yield` statement produces the square of each number `i` during each iteration, allowing for lazy evaluation.
* This generator can be used to produce a sequence of squared numbers without storing them all in memory at once.
* To retrieve the squared values, the function can be called in a loop or converted to a list.

```
def square(num):
                                                    for i
```

This code demonstrates the use of a generator to produce square numbers sequentially.

Explanation

* The generator `gen` is created by calling the `square` function with an argument of 10, which is expected to yield square numbers starting from 0 up to 9.
* The `next(gen)` function retrieves the next value from the generator, allowing for sequential access to the generated square numbers.
* The first three calls to `next(gen)` print the squares of 0, 1, and 2, respectively.
* The `for` loop continues to iterate over the remaining values from the generator, starting from the last yielded value (which would be 3), and prints the subsequent squares until the generator is exhausted.

```
gen = square(10)
```

Output

```
1
                                                    4
```

Range function using Generator

This code defines a generator function that produces a range of numbers from start to end.

Explanation

* The function `my_range` takes two parameters, `start` and `end`, which define the range of numbers to be generated.
* It uses a `for` loop to iterate over the range created by Python's built-in `range` function.
* The `yield` statement allows the function to return a value and pause its state, making it a generator that can produce values one at a time.
* This approach is memory efficient, as it generates numbers on-the-fly instead of storing them all in memory at once.
* The generator can be iterated over using a loop or converted to a list, providing flexibility in how the generated numbers are used.

```
def my_range(start,end):
                                                    for i
```

This code snippet demonstrates how to create and iterate over a custom range generator in Python.

Explanation

* The variable `gen` is assigned the result of calling the `my_range` function with arguments 15 and 26, which presumably generates a sequence of numbers from 15 to 25.
* The `for` loop iterates over each number produced by the `gen` generator.
* Inside the loop, each number `i` is printed to the console, resulting in the output of numbers from 15 to 25, inclusive.
* This approach is memory efficient as it generates numbers on-the-fly rather than storing them all in memory at once.

```
gen = my_range(15,26)
```

Output

```
15
```

```
16
```

This code snippet iterates through a custom range and prints each value.

Explanation

* The `my_range` function is called with parameters 2 and 6, indicating the start and end of the range.
* The loop iterates over each integer from 2 up to, but not including, 6.
* The `print(i)` statement outputs each integer in the specified range to the console.
* This code effectively demonstrates how to create a loop that processes a specific range of numbers.

```
for i in my_range(2,6):
```

```
    print(
```

Output

```
2
```

```
3
```

Generator Expression

This code snippet demonstrates how to create a list of squared numbers using list comprehension in Python.

Explanation

* The code initializes a list `L` that will store the squared values.
* It uses a list comprehension to iterate over a range of numbers from 1 to 5 (inclusive).

* For each number `i` in the range, it calculates `i**2`, which is the square of `i`.
* The resulting list `L` will contain the values `[1, 4, 9, 16, 25]`.
* This approach is concise and efficient for generating lists based on existing iterables.

```
# list comprehension
```

This code snippet demonstrates the use of a generator expression to produce squares of numbers from 1 to 5.

Explanation

* A generator expression is created that computes the square of each integer from 1 to 5 using `i**2 for i in range(1,6)`.
* The generator is stored in the variable `gen`, which allows for lazy evaluation, meaning values are generated on-the-fly as needed.
* A for loop iterates over the generator, retrieving and printing each squared value sequentially.
* This approach is memory efficient since it does not store all squared values in memory at once, but generates them one by one.

```
gen = (i**2 for i in range(1,6))
```

Output

```
1
4
```

Benefits of using a Generator

1. Ease of Implementation

Custom iterable class implementation for generating a range of numbers in Python

Explanation

* Defines a class `my_range` that initializes with a `start` and `end` value.

* Implements the `__iter__` method to return an instance of `my_iterator`, allowing iteration over the range.
* This structure enables the creation of custom iterable objects that can be used in loops and comprehensions.
* The class can be expanded with additional methods to enhance functionality, such as supporting slicing or length checks.

```
class my_range:
```

Custom iterator class for iterating over a range of numbers

Explanation

* Defines a class `my_iterator` that implements the iterator protocol in Python.
* The `__init__` method initializes the iterator with an iterable object containing `start` and `end` attributes.
* The `__iter__` method returns the iterator object itself, allowing it to be used in loops.
* The `__next__` method retrieves the current value, increments the `start` attribute, and raises `StopIteration` when the end of the range is reached.

```
# iterator
```

```
class my_i
```

This code defines a generator function to create a custom range of numbers.

Explanation

* The function `my_range` takes two parameters, `start` and `end`, which define the range of numbers to generate.
* It uses a `for` loop to iterate over the range created by the built-in `range` function, which generates numbers from `start` to `end - 1`.
* The `yield` statement allows the function to return one number at a time, making it a generator that can be iterated over without storing all values in memory at once.
* This approach is memory efficient, especially for large ranges, as it produces values on-the-fly.

```
def my_range(start,end):
```

2. Memory Efficient

Comparing memory usage between a list and a generator in Python

Explanation

* A list `L` is created using a list comprehension, containing integers from 0 to 99,999.
* A generator `gen` is created using a generator expression, which generates integers from 0 to 99,999 on-the-fly.
* The `sys.getsizeof()` function is used to measure the memory size of both the list and the generator.
* The printed output shows that the list consumes significantly more memory than the generator, illustrating the memory efficiency of generators.

```
L = [x for x in range(100000)]
```

```
gen = (x f
```

Output

```
Size of L in memory 800984
```

```
Size of ge
```

3. Representing Infinite Streams

Generator function that produces infinite sequence of even numbers starting from zero

Explanation

* Defines a generator function using the yield keyword that creates an infinite sequence of even numbers
* Initializes variable n to 0 and enters an infinite loop that never terminates
* Yields the current value of n (starting at 0) before incrementing it by 2 each iteration
* The generator can be iterated over to get successive even numbers (0, 2, 4, 6, 8...)
* This approach efficiently generates values on-demand without storing the entire sequence in memory

```
def all_even():
    n = 0
```

This code snippet demonstrates the usage of a generator to produce even numbers sequentially.

Explanation

* The variable `even_num_gen` is assigned a generator object created by calling the `all_even()` function, which is expected to yield even numbers.
* The `next()` function is called three times on the generator, which retrieves the next value from the generator each time it is invoked.
* Each call to `next(even_num_gen)` advances the generator to the next even number, effectively skipping the previous ones.
* This approach allows for efficient memory usage since only one even number is generated at a time rather than storing all even numbers in a list.

```
even_num_gen = all_even()
next(even_
```

Output

```
4
```

4. Chaining Generators

Calculate the sum of squares of the first ten Fibonacci numbers using generators in Python

Explanation

* The `fibonacci_numbers` function generates Fibonacci numbers up to a specified count using a generator.
* The `square` function takes an iterable and yields the square of each number from that iterable.
* The `print` statement computes the sum of the squares of the first ten Fibonacci numbers by chaining the two generator functions.
* The use of `yield` allows for efficient memory usage, as values are generated on-the-fly rather than stored in a list.

```
def fibonacci_numbers(nums):
    x, y =
```

Output

```
4895
```