Python

Python Generators

Elegant, Memory-Efficient Iterations A Powerful Python Feature

March 14, 2025

1. Introduction to Python Generators

Python generators provide an elegant way to create iterators with minimal memory footprint. Unlike lists that store all values in memory, generators produce values on-the-fly, making them ideal for handling large datasets or infinite sequences.

1.1. What Are Generators?

Generators are special functions that return an iterator using the **yield** statement instead of **return**. This allows the function to pause execution and later resume from where it left off.

- **Memory Efficiency:** Values are generated one at a time, not stored in memory
- **Lazy Evaluation:** Values are computed only when needed
- **Simplicity:** Cleaner code compared to implementing iterators manually

1.2. Generators vs. Lists

When comparing generators to traditional data structures like lists, we find several key differences:

- **Memory Usage:** Generators consume significantly less memory than equivalent lists
- **Computation:** Lists compute all values at once; generators compute values on-demand
- **Access Patterns:** Lists allow random access; generators only permit sequential access

---

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

- **Reusability:** Lists can be iterated multiple times; generators are exhausted after one iteration

2. Creating Python Generators

There are two primary ways to create generators in Python: generator functions and generator expressions.

2.1. Generator Functions

Generator functions look like regular functions but use the **yield** keyword to return values:

```python
def countdown(n):
    """A simple generator function that counts down from n to 1"""
    print("Starting countdown!")
    while n > 0:
        yield n
        n -= 1
    print("Countdown complete!")

# Using the generator
counter = countdown(5)
print(next(counter))  # 5
print(next(counter))  # 4
print(next(counter))  # 3
```


Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

The state of the function is preserved between yields, allowing it to resume execution from where it left off.

2.2. Generator Expressions

Generator expressions provide a concise way to create generators, similar to list comprehensions but with parentheses instead of square brackets:

```
# List comprehension (creates entire list in memory)
squares_list = [x*x for x in range(1000000)]  # Uses more memory

# Generator expression (creates generator object)
squares_gen = (x*x for x in range(1000000))   # Uses minimal memory

# Using the generator expression
print(next(squares_gen))  # 0
print(next(squares_gen))  # 1
print(next(squares_gen))  # 4
```

3. Working with Python Generators

Generators can be used in many contexts where iterables are expected.


Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

3.1. Basic Operations with Generators

Here are common ways to interact with generators:

```python
def first_n_fibonacci(n):
    """Generate first n Fibonacci numbers"""
    a, b = 0, 1
    count = 0
    while count < n:
        yield a
        a, b = b, a + b
        count += 1

# Iterating with a for loop
fib = first_n_fibonacci(10)
for num in fib:
    print(num, end=’ ’)  # 0 1 1 2 3 5 8 13 21 34
```

4. Memory Efficiency with Generators

One of the main advantages of generators is their memory efficiency.

4.1. Memory Comparison: Lists vs. Generators

Let’s compare memory usage between lists and generators:


Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```python
import tracemalloc

# Start memory monitoring
tracemalloc.start()

# Create a large list
large_list = [i * i for i in range(1000000)]
list_snapshot = tracemalloc.take_snapshot()
list_size = sum(stat.size for stat in
    list_snapshot.statistics(’filename’))

# Reset monitoring
tracemalloc.stop()
tracemalloc.start()

# Create an equivalent generator
large_gen = (i * i for i in range(1000000))
gen_snapshot = tracemalloc.take_snapshot()
gen_size = sum(stat.size for stat in
    gen_snapshot.statistics(’filename’))

# Compare memory usage
print(f"List memory: {list_size / 1024 / 1024:.2f} MB")
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```
print(f"Generator memory: {gen_size / 1024 / 1024:.2f} MB")
print(f"Memory ratio: {list_size / gen_size:.0f}x")

# Create a large list
large_list = [i * i for i in range(1000000)]
list_snapshot = tracemalloc.take_snapshot()
list_size = sum(stat.size for stat in
    list_snapshot.statistics(’filename’))

# Reset monitoring
tracemalloc.stop()
tracemalloc.start()

# Create an equivalent generator
large_gen = (i * i for i in range(1000000))
gen_snapshot = tracemalloc.take_snapshot()
gen_size = sum(stat.size for stat in
    gen_snapshot.statistics(’filename’))

# Compare memory usage
print(f"List memory: {list_size / 1024 / 1024:.2f} MB")
print(f"Generator memory: {gen_size / 1024 / 1024:.2f} MB")
print(f"Memory ratio: {list_size / gen_size:.0f}x")
```

The memory savings can be substantial, especially when processing large datasets.

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

5. Conclusion

Python generators provide an elegant, memory-efficient way to work with data sequences and iterative computations. They excel in scenarios involving large datasets, stream processing, and computational pipelines.

5.1. Key Takeaways

- **Memory Efficiency:** Generators calculate values on-demand, avoiding memory overhead
- **Lazy Evaluation:** Computation happens only when needed, improving performance
- **Elegant APIs:** Create clean, readable code for data processing pipelines
- **Infinite Sequences:** Work with potentially infinite data without memory concerns
- **Foundation for Async:** Generators provided the foundation for Python's async/await syntax

Mastering generators is an essential skill for writing efficient, elegant Python code, especially when dealing with large data processing tasks.

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

Generators: The Future of Iteration

How will you optimize your code with generators?