Python

Python Generators

Efficient Iteration and Memory Management

April 29, 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
- **State Preservation:** Generators maintain their state between calls
- **Sequence Creation:** Easily model complex or infinite sequences

When using **yield**, the internal state of the function (local variables and execution point) is saved, allowing the function to resume from where it left off on the next call to `next()`.

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

1.2. Generators vs. Lists

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

```python
# List comprehension - loads all in memory
numbers_list = [x * 2 for x in range(1000000)]

# Generator expression - computes on-demand
numbers_gen = (x * 2 for x in range(1000000))

# Memory comparison
import sys
list_size = sys.getsizeof(numbers_list)
# ~8.06 MB
gen_size = sys.getsizeof(numbers_gen)
# ~200B
```

Other important differences include:

- **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
- **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"""
```

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

```python
    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
```

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:

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

```python
# Method 1: Generator function with yield
def count_up_to(max):
    count = 1
    while count <= max:
        yield count
        count += 1

# Method 2: Generator expression
squares = (x**2 for x in range(10))

# Using generators
for num in count_up_to(5):
    print(num)  # Prints: 1, 2, 3, 4, 5

for num in squares:
    print(num)  # Prints: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81
```

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

# Alternatively with next()
```

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

```python
fib = first_n_fibonacci(5)
print(next(fib))  # 0
print(next(fib))  # 1
print(next(fib))  # 1
```

3.2. Infinite Sequences

Generators are particularly useful for working with potentially infinite sequences:

```python
# Creating an infinite sequence of Fibonacci numbers
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

# Using the infinite generator safely
fib_gen = fibonacci()
```


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

```python
for _ in range(10):
    print(next(fib_gen))

# Output: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34
```

3.3. The yield from Statement

Python 3.3 introduced the **yield from** statement, which simplifies delegation to sub-generators:

```python
from collections.abc import Sequence

# Without yield from
def subgenerator(n):
    for i in range(n):
        yield i

def main_generator_old(n):
    for val in subgenerator(n):
```

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

```python
        yield val

# With yield from - more elegant
def main_generator_new(n):
    yield from subgenerator(n)

def flatten(nested_list):
    for item in nested_list:
        if isinstance(item, Sequence) and not isinstance(item, (str, bytes)):
            yield from flatten(item)
        else:
            yield item
```

4. Generator Pipelines

Generators can be chained together to create powerful data processing pipelines:


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

PYTHON | **GENERATORS**

```python
def read_file(file_path):
    with open(file_path, 'r') as f:
        for line in f:
            yield line.strip()

def grep(lines, pattern):
    for line in lines:
        if pattern in line:
            yield line

def uppercase(lines):
    for line in lines:
        yield line.upper()

# Usage
file_lines = read_file('data.txt')
filtered = grep(file_lines, 'python')
result = uppercase(filtered)

# Process results
```

Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

www.asanchezyali.com

```python
for line in result:
    print(line)
```

This approach is memory-efficient because each line is processed one at a time through the entire pipeline.

5. Memory Efficiency with Generators

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

5.1. Memory Comparison: Lists vs. Generators

Let’s compare memory usage between lists and generators:

```python
import tracemalloc

# Start memory monitoring
tracemalloc.start()

# Create a large list
```

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

PYTHON | **GENERATORS**

```python
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:.8f} MB")
print(f"Generator memory: {gen_size / 1024 / 1024:.8f} MB")
print(f"Memory ratio: {list_size / gen_size:.0f}x")

```

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

```
# Output:
# List memory: 38.57472229 MB
# Generator memory: 0.00038147 MB
# Memory ratio: 101121x
```

While generators save memory, they can be slower than lists for operations requiring repeated access, as values must be recalculated each time.

5.2. Processing Large Files

Generators are particularly useful when working with files that would be too large to fit in memory:

```
# Processing a large file with a list
def process_file_list(filename):
    with open(filename) as f:
        # All lines loaded in memory at once
        return [line.upper() for line in f]

# Processing with a generator
```

```python
def process_file_generator(filename):
    with open(filename) as f:
        for line in f:
            # Process one line at a time
            yield line.upper()
```

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

6. The Iterator Protocol

Under the hood, generators implement Python's iterator protocol, which requires **__iter__** and **__next__** methods:

```python
# Generator functions implement this protocol:
class Counter:
    def __init__(self, max_value):
        self.max_value = max_value
        self.current = 0

```

```python
    def __iter__(self):
        return self

    def __next__(self):
        if self.current >= self.max_value:
            raise StopIteration
        self.current += 1
        return self.current

# Example usage
counter = Counter(5)
for number in counter:
    print(number)

# Output: 1, 2, 3, 4, 5
```

Generators eliminate the need to manually implement `__iter__` and `__next__`, as Python automatically generates them when you use `yield`.

7. Real-World Applications

PYTHON | GENERATORS

7.1. Log Processing

Efficiently process large log files without excessive memory usage:

```python
# Processing a large log file efficiently
def parse_log_line(line):
    # Extract timestamp and message
    parts = line.split(" ", 1)
    return {"timestamp": parts[0], "message": parts[1]}

def filter_errors(log_entries):
    for entry in log_entries:
        if "ERROR" in entry["message"]:
            yield entry

def process_logs(filename):
    with open(filename) as f:
        # Parse each line
        entries = (parse_log_line(line) for line in f)
```

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

```
        # Filter for errors
        errors = filter_errors(entries)
        # Group by hour
        for error in errors:
            yield error
```

7.2. Data Transformation Pipelines

Create efficient data processing workflows:

```
def csv_reader(file_path):
    for line in open(file_path, ’r’):
        yield line.strip().split(’,’)

def select_columns(data, indices):
    for row in data:
        yield [row[i] for i in indices]

```

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

```python
def filter_rows(data, condition_func):
    for row in data:
        if condition_func(row):
            yield row

# Usage example
data = csv_reader(’large_dataset.csv’)
selected = select_columns(data, [0, 2, 3])
filtered = filter_rows(selected, lambda x: float(x[1]) > 100)

for row in filtered:
    print(row)
```

7.3. Generators with itertools

Combining generators with Python’s itertools library provides powerful data manipulation capabilities:

```python
from itertools import islice
```

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

```python

def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

fib_slice = islice(fibonacci(), 10)
print(list(fib_slice))  # [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

8. Best Practices

To get the most from generators in Python:

- **Use generator expressions** for simple transformations
- **Use generator functions** for complex logic or when state is needed
- **Chain generators together** to create processing pipelines
- **Remember generators are single-use** — create new ones if needed
- **Use yield from** to delegate to sub-generators


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

- **Add type hints** with typing.Generator for clarity
- **Consider contextlib.contextmanager** for resource management

9. 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.

9.1. 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


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

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

9.2. Further Reading

To dive deeper into Python generators, check these resources:

- PEP 255 - The original Python Generator proposal
- *Fluent Python* by Luciano Ramalho, which dedicates an entire chapter to iterators and generators
- Python's official documentation on generators and the itertools module
- Remember this document was translated, edited and written in collaboration with AI.


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

10. Explore My Other Posts

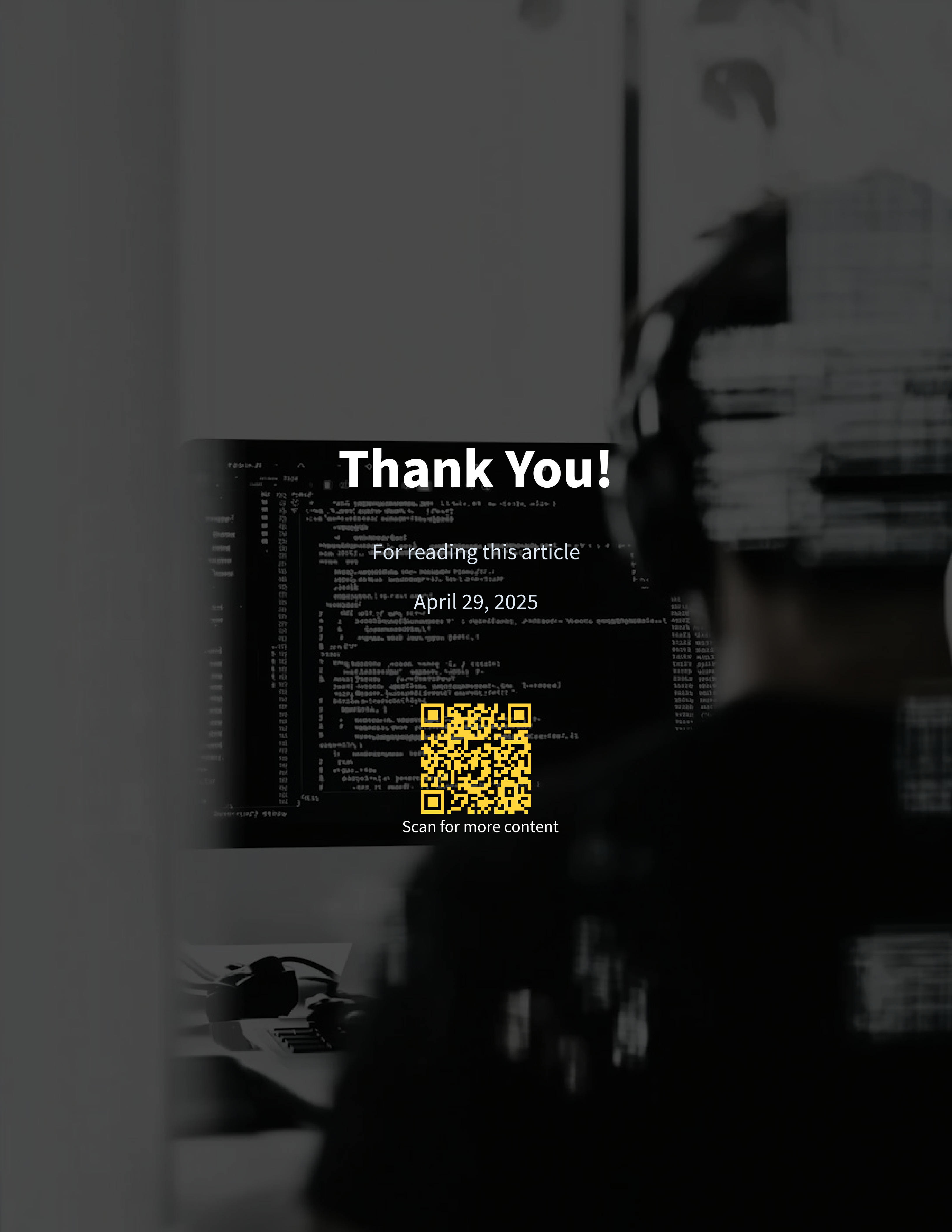
Enjoyed This Content?

Don’t miss my previous post about:

Python Context Managers: Elegant Resource Management with the «with» Statement

Discover how Python Context Managers can help you elegantly manage resources, prevent memory leaks, and write cleaner code.

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



Thank You!

For reading this article

April 29, 2025

Scan for more content