

Python

# Python Generators

Elegant, Memory-Efficient Iterations A Powerful Python Feature

April 13, 2025



Source Code



# 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



**Alejandro Sánchez Yalí**

Software Developer | AI & Blockchain Enthusiast

 [www.asanchezyali.com](http://www.asanchezyali.com)

## 1.2. Generators vs. Lists

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

```
1 # List comprehension - loads all in memory
2 numbers_list = [x * 2 for x in range(1000000)]
3
4 # Generator expression - computes on-demand
5 numbers_gen = (x * 2 for x in range(1000000))
6
7 # Memory comparison
8 import sys
9 list_size = sys.getsizeof(numbers_list)
10 # ~8.06 MB
11 gen_size = sys.getsizeof(numbers_gen)
12 # ~200B
```

Other important differences include:

- **Memory Usage:** Generators consume significantly less memory than equiv-



**Alejandro Sánchez Yalí**

Software Developer | AI & Blockchain Enthusiast

 [www.asanchezyali.com](http://www.asanchezyali.com)

alent 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:

```
1 def countdown(n):  
2     """A simple generator function that counts down from n to 1"""
```



**Alejandro Sánchez Yalí**

Software Developer | AI & Blockchain Enthusiast

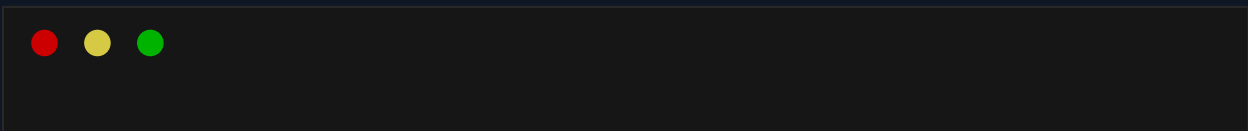
 [www.asanchezyali.com](http://www.asanchezyali.com)

```
3     print("Starting countdown!")
4     while n > 0:
5         yield n
6         n -= 1
7     print("Countdown complete!")
8
9 # Using the generator
10 counter = countdown(5)
11 print(next(counter)) # 5
12 print(next(counter)) # 4
13 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](http://www.asanchezyali.com)

```
1 # Method 1: Generator function with yield
2 def count_up_to(max):
3     count = 1
4     while count <= max:
5         yield count
6         count += 1
7
8 # Method 2: Generator expression
9 squares = (x**2 for x in range(10))
10
11 # Using generators
12 for num in count_up_to(5):
13     print(num) # Prints: 1, 2, 3, 4, 5
14
15 for num in squares:
16     print(num) # Prints: 0, 1, 4, 9, 16j
```


### 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](http://www.asanchezyali.com)

### 3.1. Basic Operations with Generators


Here are common ways to interact with generators:

```
1 def first_n_fibonacci(n):
2     """Generate first n Fibonacci numbers"""
3     a, b = 0, 1
4     count = 0
5     while count < n:
6         yield a
7         a, b = b, a + b
8         count += 1
9
10 # Iterating with a for loop
11 fib = first_n_fibonacci(10)
12 for num in fib:
13     print(num, end=' ') # 0 1 1 2 3 5 8 13 21 34
```



**Alejandro Sánchez Yalí**

Software Developer | AI & Blockchain Enthusiast

 [www.asanchezyali.com](http://www.asanchezyali.com)

## 3.2. Infinite Sequences

Generators are particularly useful for working with potentially infinite sequences:

```
1 # Creating an infinite sequence of Fibonacci numbers
2 def fibonacci():
3     a, b = 0, 1
4     while True:
5         yield a
6         a, b = b, a + b
7
8 # Using the infinite generator safely
9 fib_gen = fibonacci()
10 for _ in range(10):
11     print(next(fib_gen))
12
13 # Output: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34
```



**Alejandro Sánchez Yalí**

Software Developer | AI & Blockchain Enthusiast

 [www.asanchezyali.com](http://www.asanchezyali.com)



### 3.3. The yield from Statement


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

```
1 from collections.abc import Sequence
2
3 # Without yield from
4 def subgenerator(n):
5     for i in range(n):
6         yield i
7
8 def main_generator_old(n):
9     for val in subgenerator(n):
10         yield val
11
12 # With yield from - more elegant
13 def main_generator_new(n):
14     yield from subgenerator(n)
```



**Alejandro Sánchez Yalí**

Software Developer | AI & Blockchain Enthusiast

 [www.asanchezyali.com](http://www.asanchezyali.com)

```
15
16 def flatten(nested_list):
17     for item in nested_list:
18         if isinstance(item, Sequence) and not isinstance(item, (str,
19             bytes)):
20             yield from flatten(item)
21         else:
22             yield item
```

## 4. Generator Pipelines


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

```
1 def read_file(file_path):
2     with open(file_path, 'r') as f:
3         for line in f:
4             yield line.strip()
5
```



**Alejandro Sánchez Yalí**

Software Developer | AI & Blockchain Enthusiast

 [www.asanchezyali.com](http://www.asanchezyali.com)


```
6 def grep(lines, pattern):
7     for line in lines:
8         if pattern in line:
9             yield line
10
11 def uppercase(lines):
12     for line in lines:
13         yield line.upper()
14
15 # Usage
16 file_lines = read_file('data.txt')
17 filtered = grep(file_lines, 'python')
18 result = uppercase(filtered)
19
20 # Process results
21 for line in result:
22     print(line)
```

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



**Alejandro Sánchez Yalí**

Software Developer | AI & Blockchain Enthusiast

 [www.asanchezyali.com](http://www.asanchezyali.com)

## 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:

```
1 import tracemalloc
2
3 # Start memory monitoring
4 tracemalloc.start()
5
6 # Create a large list
7 large_list = [i * i for i in range(1000000)]
8 list_snapshot = tracemalloc.take_snapshot()
9 list_size = sum(stat.size for stat in
    list_snapshot.statistics('filename'))
10
11 # Reset monitoring
```



**Alejandro Sánchez Yalí**

Software Developer | AI & Blockchain Enthusiast


 [www.asanchezyali.com](http://www.asanchezyali.com)

```
12 tracemalloc.stop()
13 tracemalloc.start()
14
15 # Create an equivalent generator
16 large_gen = (i * i for i in range(1000000))
17 gen_snapshot = tracemalloc.take_snapshot()
18 gen_size = sum(stat.size for stat in
    gen_snapshot.statistics('filename'))
19
20 # Compare memory usage
21 print(f"List memory: {list_size / 1024 / 1024:.8f} MB")
22 print(f"Generator memory: {gen_size / 1024 / 1024:.8f} MB")
23 print(f"Memory ratio: {list_size / gen_size:.0f}x")
24
25 # Output:
26 # List memory: 38.57472229 MB
27 # Generator memory: 0.00038147 MB
28 # Memory ratio: 101121x
```



## Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 [www.asanchezyali.com](http://www.asanchezyali.com)

## 5.2. Processing Large Files

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

```
1 # Processing a large file with a list
2 def process_file_list(filename):
3     with open(filename) as f:
4         # All lines loaded in memory at once
5         return [line.upper() for line in f]
6
7 # Processing with a generator
8 def process_file_generator(filename):
9     with open(filename) as f:
10         for line in f:
11             # Process one line at a time
12             yield line.upper()
```

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



**Alejandro Sánchez Yalí**

Software Developer | AI & Blockchain Enthusiast

 [www.asanchezyali.com](http://www.asanchezyali.com)

## 6. The Iterator Protocol


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

```
1 # Generator functions implement this protocol:
2 class Counter:
3     def __init__(self, max_value):
4         self.max_value = max_value
5         self.current = 0
6
7     def __iter__(self):
8         return self
9
10    def __next__(self):
11        if self.current >= self.max_value:
12            raise StopIteration
13        self.current += 1
14        return self.current
```



**Alejandro Sánchez Yalí**

Software Developer | AI & Blockchain Enthusiast

 [www.asanchezyali.com](http://www.asanchezyali.com)

```
15
16 # Example usage
17 counter = Counter(5)
18 for number in counter:
19     print(number)
20
21 # Output: 1, 2, 3, 4, 5
```

This demonstrates how generators simplify the creation of iterators by handling the boilerplate code.

## 7. Real-World Applications

### 7.1. Log Processing

Efficiently process large log files without excessive memory usage:

```
1 # Processing a large log file efficiently
2 def parse_log_line(line):
```



**Alejandro Sánchez Yalí**

Software Developer | AI & Blockchain Enthusiast

 [www.asanchezyali.com](http://www.asanchezyali.com)



```
3     # Extract timestamp and message
4     parts = line.split(" ", 1)
5     return {"timestamp": parts[0], "message": parts[1]}
6
7 def filter_errors(log_entries):
8     for entry in log_entries:
9         if "ERROR" in entry["message"]:
10             yield entry
11
12 def process_logs(filename):
13     with open(filename) as f:
14         # Parse each line
15         entries = (parse_log_line(line) for line in f)
16         # Filter for errors
17         errors = filter_errors(entries)
18         # Group by hour
19         for error in errors:
20             yield error
```

**Alejandro Sánchez Yalí**

Software Developer | AI &amp; Blockchain Enthusiast

 [www.asanchezyali.com](http://www.asanchezyali.com)

## 7.2. Data Transformation Pipelines


Create efficient data processing workflows:

```
1 def csv_reader(file_path):
2     for line in open(file_path, 'r'):
3         yield line.strip().split(',')
4
5 def select_columns(data, indices):
6     for row in data:
7         yield [row[i] for i in indices]
8
9 def filter_rows(data, condition_func):
10    for row in data:
11        if condition_func(row):
12            yield row
13
14 # Usage example
15 data = csv_reader('large_dataset.csv')
```



**Alejandro Sánchez Yalí**

Software Developer | AI & Blockchain Enthusiast

 [www.asanchezyali.com](http://www.asanchezyali.com)

```
16 selected = select_columns(data, [0, 2, 3])
17 filtered = filter_rows(selected, lambda x: float(x[1]) > 100)
18
19 for row in filtered:
20     print(row)
```

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



**Alejandro Sánchez Yalí**

Software Developer | AI & Blockchain Enthusiast

 [www.asanchezyali.com](http://www.asanchezyali.com)

## 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

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](http://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](http://www.asanchezyali.com)

# Generators: The Future of Iteration

How will you optimize your code with generators?