Efficient Iteration in Python

How would you generate an infinite sequence of numbers starting from 8, stepping by 2?

Andres Pineda
Senior Developer @ AFL

OSS Contributor @ Uno Platform

RealPython | PythonMTL | PythonSDQ

@ajpinedam

Efficient Iteration in Python

Presentation Feedback

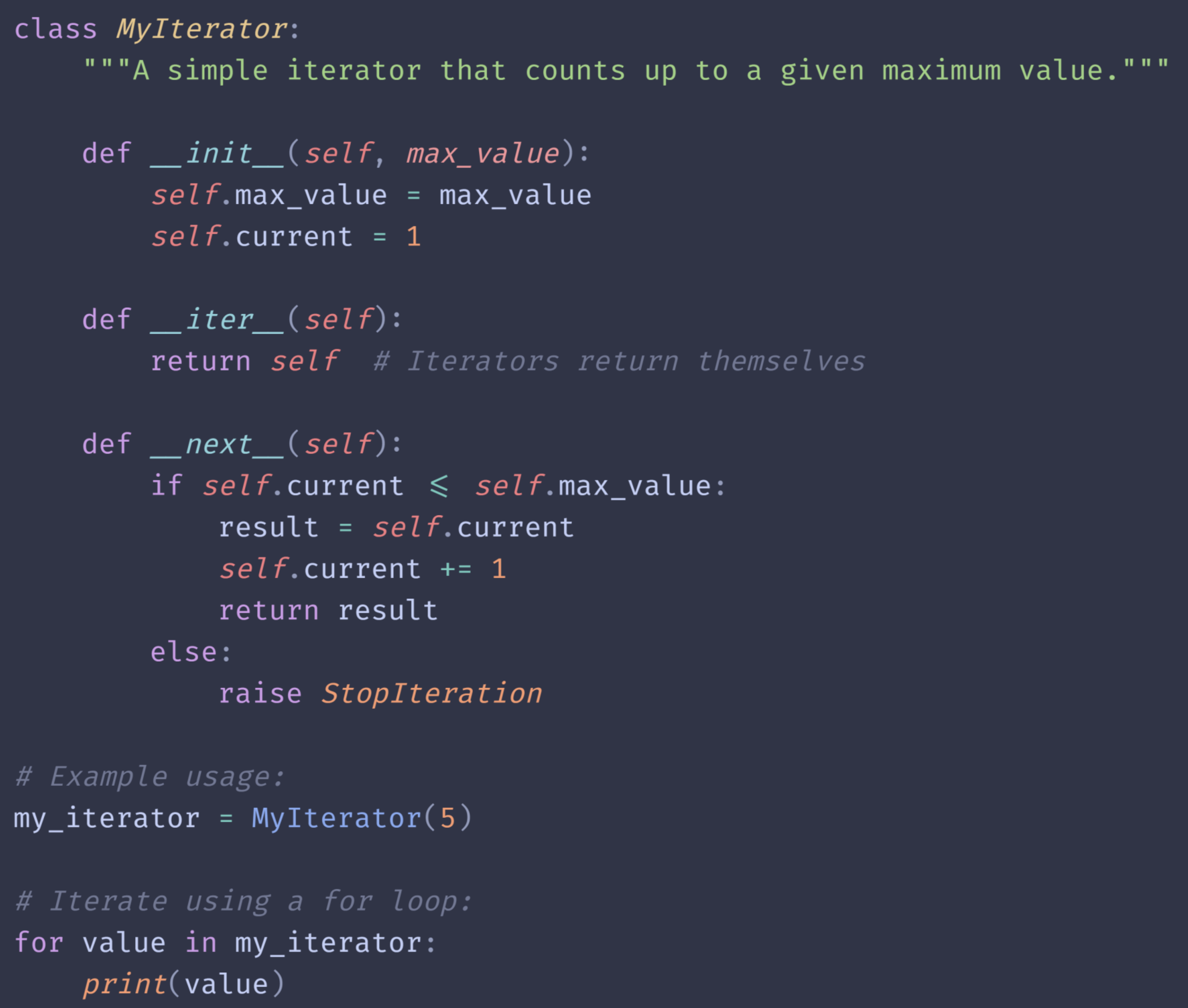
DON'T WORRY
I'VE GOT YOU
imgflip.com

What will we see?

- Python Generator and Iterators
- Introduction to Itertools
- Itertools for Iteration and Looping
- Combining Iterables with Itertools
- Filtering and Grouping Data
- Efficiency and Performance Considerations

Exploring Iterators

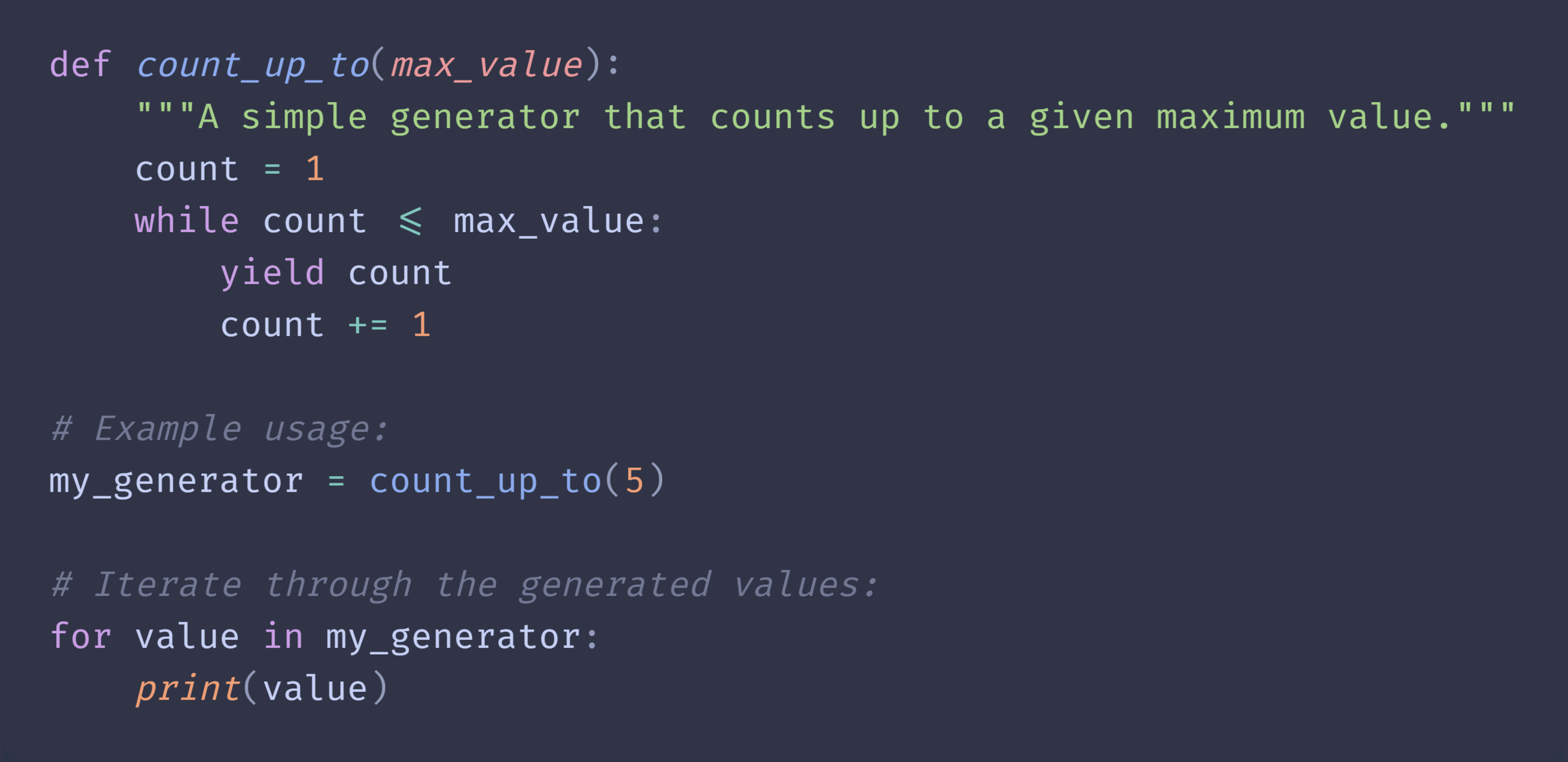
- Iterators are objects that implement the iterator protocol.
- They have two primary methods: __iter__() and __next__().
- Iterators enable traversal through collections without exposing the underlying structure.



```python
class MyIterator:
    """A simple iterator that counts up to a given maximum value."""

    def __init__(self, max_value):
        self.max_value = max_value
        self.current = 1

    def __iter__(self):
        return self  # Iterators return themselves

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

# Example usage:
my_iterator = MyIterator(5)

# Iterate using a for loop:
for value in my_iterator:
    print(value)
```

Understanding Generators

- Generators are special functions that yield values one at a time.
- They use the yield keyword to produce a value and maintain state.
- Generators are memory-efficient, ideal for large data sets.



```python
def count_up_to(max_value):
    """A simple generator that counts up to a given maximum value."""
    count = 1
    while count <= max_value:
        yield count
        count += 1

# Example usage:
my_generator = count_up_to(5)

# Iterate through the generated values:
for value in my_generator:
    print(value)
```

Key differences

- Iterators are classes that implement the __iter__() and __next__() methods.
- Generators are functions that use the yield keyword.
- Iterators are more flexible for complex iteration logic, while generators are often simpler for straightforward sequences.
- Iterators maintain state through instance variables, while generators maintain state through their execution context.

Introduction to Itertools

The **Itertools** module is a standard Python library essential for creating iterators for efficient looping.

Itertools History

Brought as Standard Library to Python 2.3 in 2003

Part of Python "Batteries Included", meaning it's no needed to be installed.

Raymond Hettinger is one of the most notable contributors

https://www.youtube.com/watch?v=OSGv2VnC0go

Overview of the Itertools Module

Efficient Iterator Tools

Itertools offers fast and memory-efficient tools that streamline the use of iterators in Python programming.

Creating Complex Iterators

The module provides functions that enable the creation of complex iterators, enhancing the functionality of your code.

Performance Enhancement

Using itertools can significantly enhance performance, allowing for more efficient data handling and iteration.

Why should we use Itertools?

Cleaner Code

Using Itertools allows developers to write cleaner and more readable code, enhancing overall code quality and maintainability.

Reduced Memory Usage

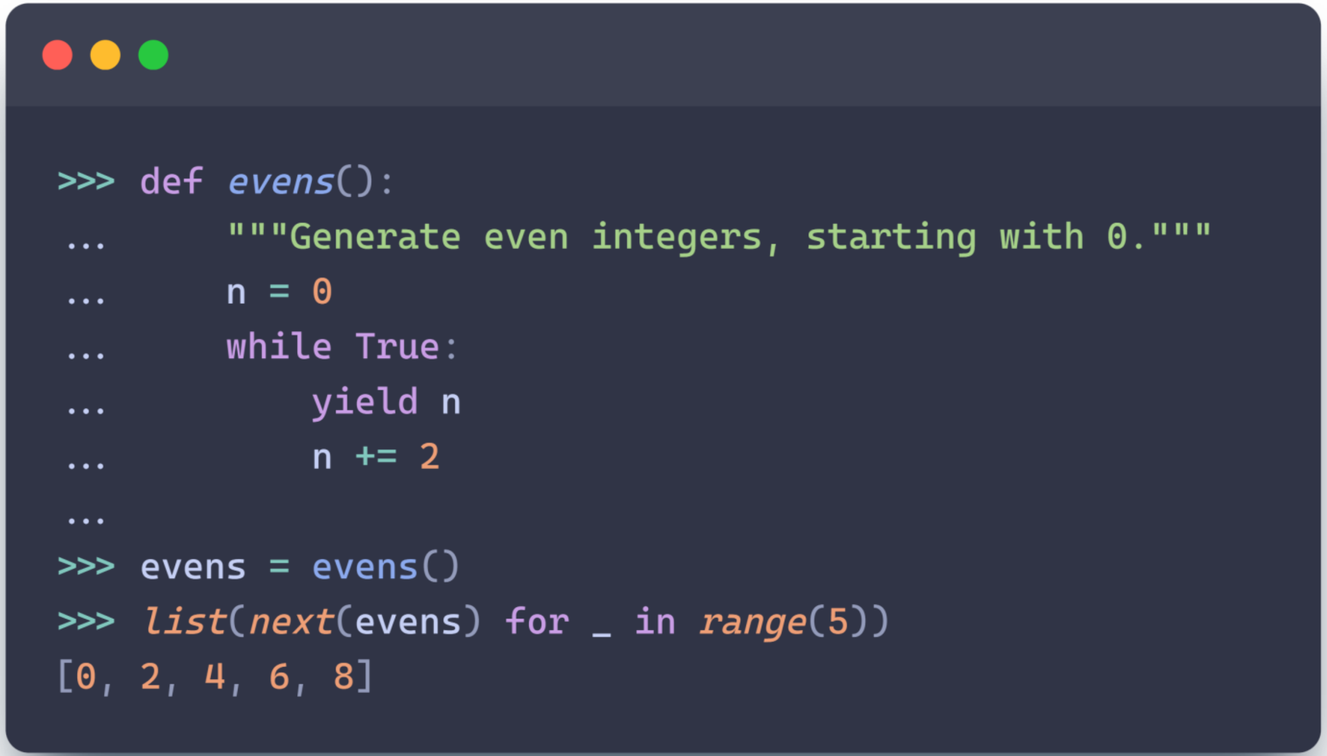
Itertools helps to efficiently manage memory usage, making it ideal for working with large datasets without overwhelming system resources.

Improved Performance

The use of Itertools can significantly improve performance, particularly when processing large volumes of data.

Itertools for Iteration and Looping

Even Numbers



```
>>> def evens():
...     """Generate even integers, starting with 0."""
...     n = 0
...     while True:
...         yield n
...         n += 2
...
>>> evens = evens()
>>> list(next(evens) for _ in range(5))
[0, 2, 4, 6, 8]
```

Infinite Iterators

`Count()`

The Count() function generates an infinite sequence of integers, starting from a specified number and increasing by one each time.

`Cycle()`

Cycle() allows you to repeat the elements of an iterable infinitely, cycling through them continuously without stopping.

`Repeat()`

The Repeat() function enables you to replicate a single value multiple times, creating an infinite sequence of that value.

Cycling through a list of options

```python
colors = ['red', 'blue', 'green']
for i in range(10):
    print(colors[i % len(colors)])
```

Useful for Round-robin scheduling.

itertools.cycle()

```python
from itertools import cycle

colors = cycle(['red', 'blue', 'green'])

for _ in range(10):
    print(next(colors))
```

Useful for Round-robin scheduling.

itertools.cycle()

Useful for Round-robin scheduling.

itertools.repeat()

```python
from itertools import repeat

for x in repeat('Confoo CA', 7):
    print(x)
```

Useful for preloading default values

itertools.repeat()

```python
import operator
from itertools import repeat

# List of product prices
prices = [100, 150, 200, 250]

# Fixed shipping cost to add to each price
shipping_cost = 10

# Use map with operator.add and repeat to add the shipping cost to each price
final_prices = list(map(operator.add, prices, repeat(shipping_cost)))
print(final_prices)  # Output: [110, 160, 210, 260]
```

Useful for preloading default values

Combining Iterables with Itertools

chain(), and chain.from_iterable()

`chain()`

The Chain() function is designed to combine several iterables, making data processing more efficient.

Chain() allows for seamless integration of different iterables, creating a unified iterable for further processing.

`chain.from_iterable()`

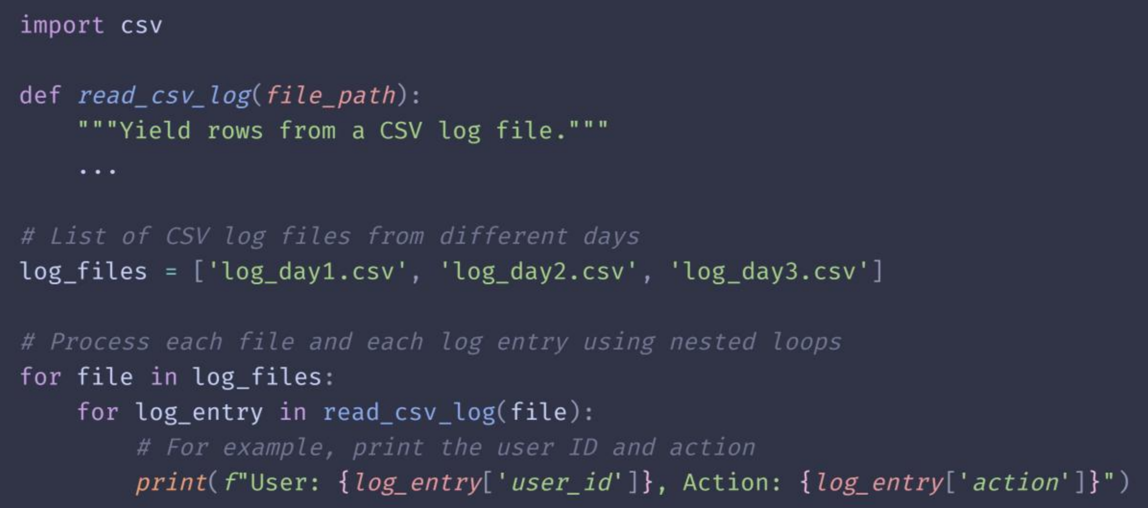
Like chain, this function helps flatten iterables, but this one receives a single iterable with nested iterables as parameter

itertools.chain

```python
from itertools import chain

print(list(chain([1, 2, 3], ['a', 'b', 'c'])))
# Output: [1, 2, 3,'a','b','c']

nested = [[1, 2, 3], [4, 5, 6]]
print(list(chain.from_iterable(nested)))
# Output: [1, 2, 3, 4, 5, 6]
```

Useful for flattening iterables

itertools.chain



```python
import csv

def read_csv_log(file_path):
    """Yield rows from a CSV log file."""
    ...

# List of CSV log files from different days
log_files = ['log_day1.csv', 'log_day2.csv', 'log_day3.csv']

# Process each file and each log entry using nested loops
for file in log_files:
    for log_entry in read_csv_log(file):
        # For example, print the user ID and action
        print(f"User: {log_entry['user_id']}, Action: {log_entry['action']}")
```

Useful for flattening iterables

Handling Uneven Iterables

Combining Iterables

`zip_longest()` allows the combination of multiple iterables of varying lengths into a single iterable without losing data.

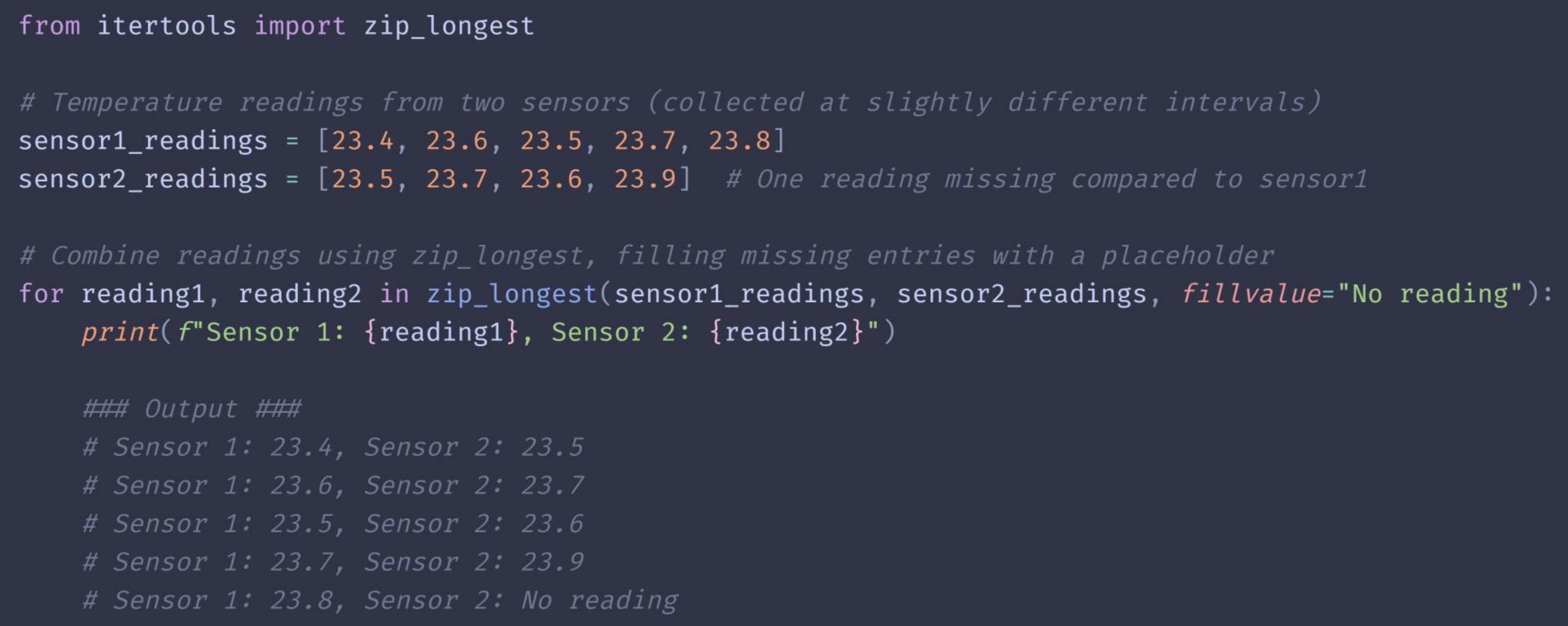
Filling Gaps

It fills gaps in shorter iterables with a specified value, ensuring a complete and accurate combination of data.

Data Integrity

Using **`zip_longest()`** enhances data integrity by preventing loss of information when working with uneven data sets.

itertools.zip_longest()



```python
from itertools import zip_longest

# Temperature readings from two sensors (collected at slightly different intervals)
sensor1_readings = [23.4, 23.6, 23.5, 23.7, 23.8]
sensor2_readings = [23.5, 23.7, 23.6, 23.9]  # One reading missing compared to sensor1

# Combine readings using zip_longest, filling missing entries with a placeholder
for reading1, reading2 in zip_longest(sensor1_readings, sensor2_readings, fillvalue="No reading"):
    print(f"Sensor 1: {reading1}, Sensor 2: {reading2}")

    ### Output ###
    # Sensor 1: 23.4, Sensor 2: 23.5
    # Sensor 1: 23.6, Sensor 2: 23.7
    # Sensor 1: 23.5, Sensor 2: 23.6
    # Sensor 1: 23.7, Sensor 2: 23.9
    # Sensor 1: 23.8, Sensor 2: No reading
```

Useful for handling uneven iterables

Product(), Permutations(), and Combinations()

`product()`

This function create all possible pairs or cartesian products, from two or more sets, enabling the combination of different data points.

`permutations()`

Permutations calculate all possible arrangements of a set, essential for understanding order and sequence in data management.

`combinations()`

Combinations select subsets from larger sets without considering order, useful for various applications in data analysis.

itertools.combinations()

```python
import itertools

students= ["Genvieve", "Guillaume", "Charlie", "Rose"]

# Generate all unique pairs of students
pairs = itertools.combinations(students, 2)

print("Scheduled meeting pairs:")
for pair in pairs:
    print(f"{pair[0]} meets {pair[1]}")
```

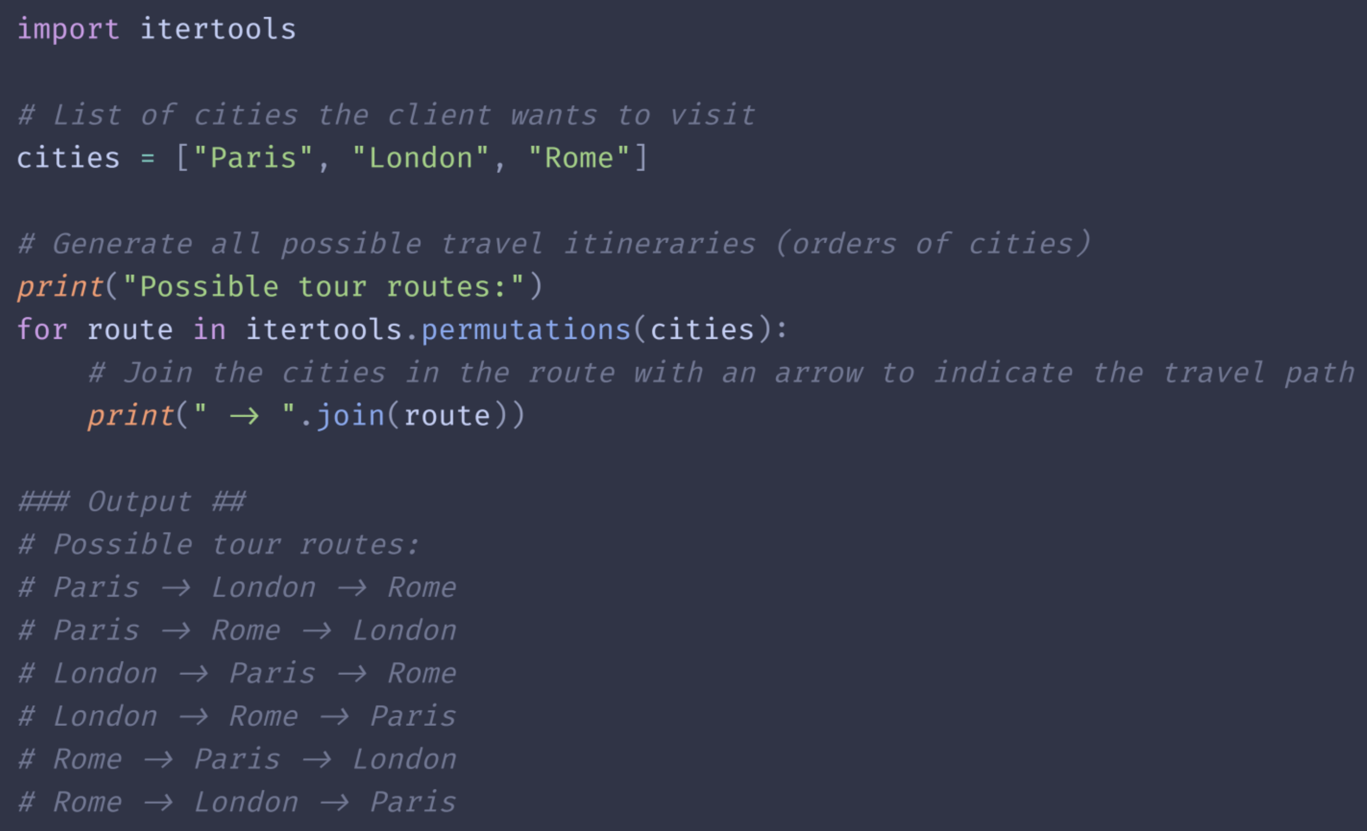
Useful for creating unique combinations

itertools.combinations()

```python
students= ["Genvieve", "Guillaume", "Charlie", "Rose"]

# Generate pairs using nested loops
pairs = []
for i in range(len(students)):
    for j in range(i + 1, len(students)):
        pairs.append((students[i], students[j]))

print("Scheduled meeting pairs:")
for pair in pairs:
    print(f"{pair[0]} meets {pair[1]}")
```

Useful for creating unique combinations

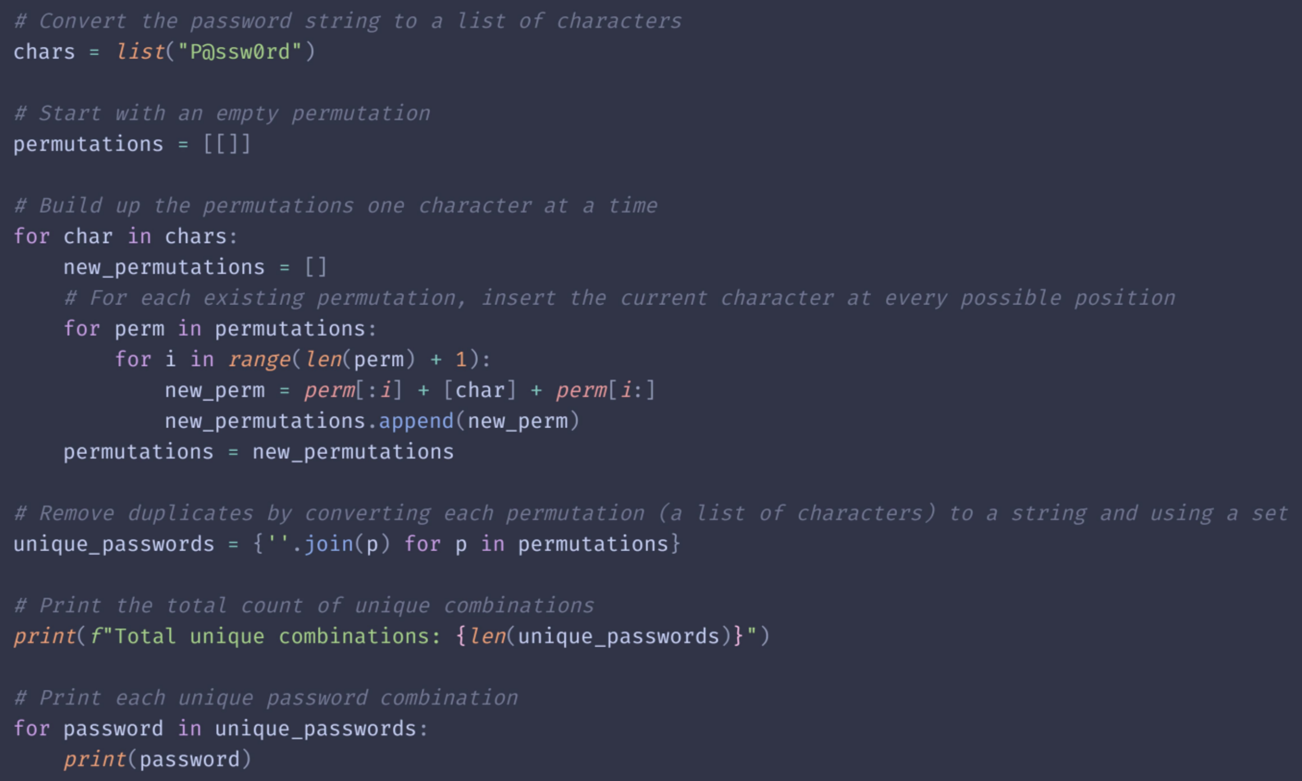
itertools.permutations()



```python
import itertools

# List of cities the client wants to visit
cities = ["Paris", "London", "Rome"]

# Generate all possible travel itineraries (orders of cities)
print("Possible tour routes:")
for route in itertools.permutations(cities):
    # Join the cities in the route with an arrow to indicate the travel path
    print(" → ".join(route))

### Output ##
# Possible tour routes:
# Paris → London → Rome
# Paris → Rome → London
# London → Paris → Rome
# London → Rome → Paris
# Rome → Paris → London
# Rome → London → Paris
```

Useful for listing all possible combinations

itertools.permutations()



```python
# Convert the password string to a list of characters
chars = list("P@ssw0rd")

# Start with an empty permutation
permutations = [[]]

# Build up the permutations one character at a time
for char in chars:
    new_permutations = []
    # For each existing permutation, insert the current character at every possible position
    for perm in permutations:
        for i in range(len(perm) + 1):
            new_perm = perm[:i] + [char] + perm[i:]
            new_permutations.append(new_perm)
    permutations = new_permutations

# Remove duplicates by converting each permutation (a list of characters) to a string and using a set
unique_passwords = {''.join(p) for p in permutations}

# Print the total count of unique combinations
print(f"Total unique combinations: {len(unique_passwords)}")

# Print each unique password combination
for password in unique_passwords:
    print(password)
```

Useful for brute-force

Filtering and Grouping Data

Using Compress() and Islice()

`compress()`

The **`compress()`** function filters elements from an iterable using a selector iterable, allowing selective data retrieval.

`Islice()`

islice() creates efficient slices of iterables, enabling access to specific sections of data without loading everything.

itertools.islice()

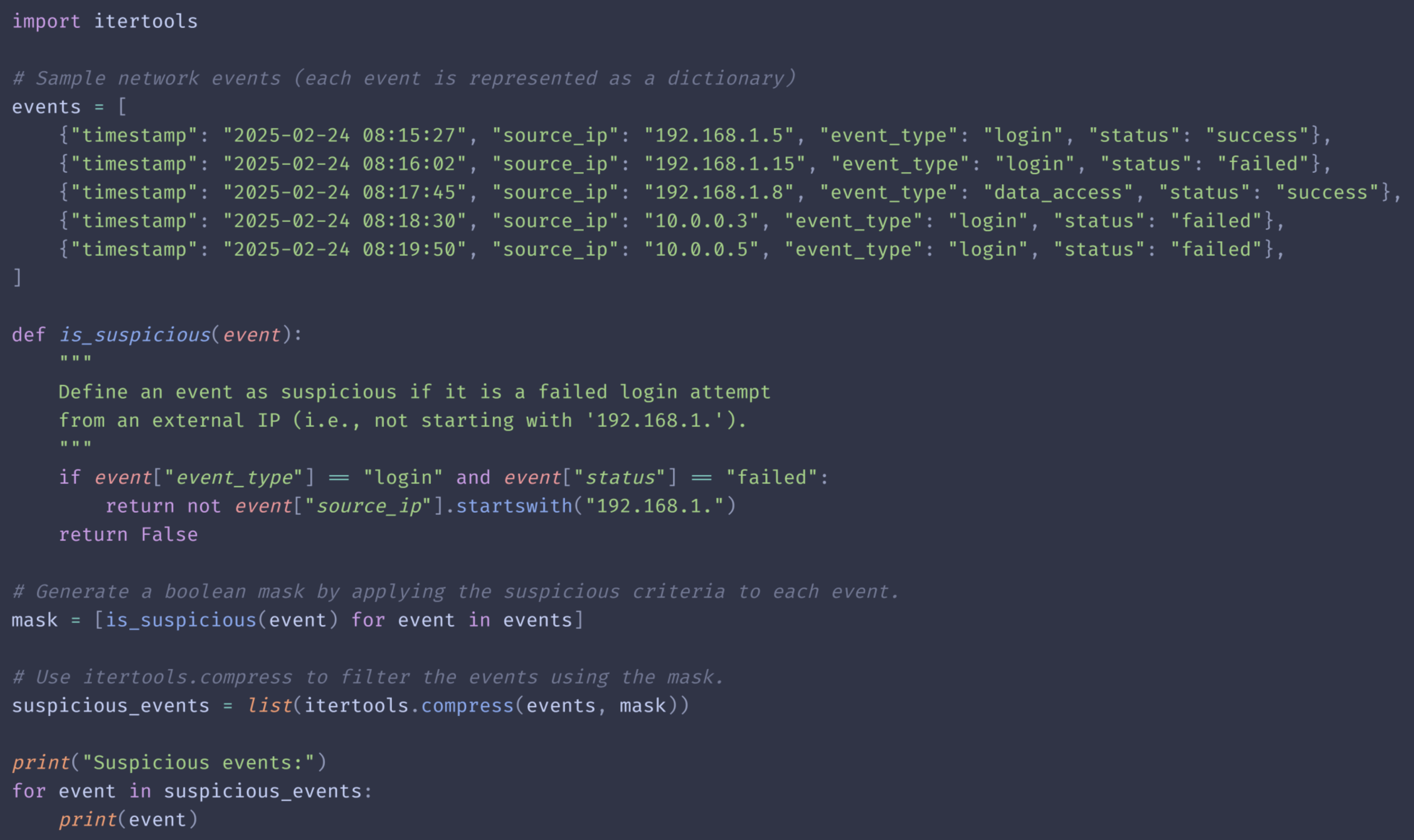
```
from itertools import islice, count

infinite_numbers = count(1)  # Infinite sequence: 1, 2, 3, 4, ...
first_five = islice(infinite_numbers, 5)  # Get first 5 elements

print(list(first_five))  # Output: [1, 2, 3, 4, 5]
```

Useful for slicing over iterators

itertools.compress()

```python
from itertools import compress

data = ['ios', 'Windows phone', 'Blackberry OS', 'android']

selectors = [True, False, False, True]

print(list(compress(data, selectors)))
```

itertools.compress()



```python
import itertools

# Sample network events (each event is represented as a dictionary)
events = [
    {"timestamp": "2025-02-24 08:15:27", "source_ip": "192.168.1.5", "event_type": "login", "status": "success"},
    {"timestamp": "2025-02-24 08:16:02", "source_ip": "192.168.1.15", "event_type": "login", "status": "failed"},
    {"timestamp": "2025-02-24 08:17:45", "source_ip": "192.168.1.8", "event_type": "data_access", "status": "success"},
    {"timestamp": "2025-02-24 08:18:30", "source_ip": "10.0.0.3", "event_type": "login", "status": "failed"},
    {"timestamp": "2025-02-24 08:19:50", "source_ip": "10.0.0.5", "event_type": "login", "status": "failed"},
]

def is_suspicious(event):
    """
    Define an event as suspicious if it is a failed login attempt
    from an external IP (i.e., not starting with '192.168.1.').
    """
    if event["event_type"] == "login" and event["status"] == "failed":
        return not event["source_ip"].startswith("192.168.1.")
    return False

# Generate a boolean mask by applying the suspicious criteria to each event.
mask = [is_suspicious(event) for event in events]

# Use itertools.compress to filter the events using the mask.
suspicious_events = list(itertools.compress(events, mask))

print("Suspicious events:")
for event in suspicious_events:
    print(event)
```

itertools.compress()

```python
from itertools import compress
from typing import List, Dict

def is_suspicious(event: Dict[str, str]) -> bool:
    """
    Define an event as suspicious if it is a failed login attempt
    from an external IP (i.e., not starting with '192.168.1.').
    """
    if event["event_type"] == "login" and event["status"] == "failed":
        return not event["source_ip"].startswith("192.168.1.")
    return False

# Sample network events (each event is represented as a dictionary)
events: List[Dict[str, str]] = [
    {"timestamp": "2025-02-24 08:15:27", "source_ip": "192.168.1.5", "event_type": "login", "status": "success"},
    {"timestamp": "2025-02-24 08:16:02", "source_ip": "192.168.1.15", "event_type": "login", "status": "failed"},
    {"timestamp": "2025-02-24 08:17:45", "source_ip": "192.168.1.8", "event_type": "data_access", "status": "success"},
    {"timestamp": "2025-02-24 08:18:30", "source_ip": "10.0.0.3", "event_type": "login", "status": "failed"},
    {"timestamp": "2025-02-24 08:19:50", "source_ip": "10.0.0.5", "event_type": "login", "status": "failed"},
]

# Create a boolean mask based on the suspicion criteria.
mask: List[bool] = [is_suspicious(event) for event in events]

# Use compress to filter events based on the mask.
suspicious_events: List[Dict[str, str]] = list(compress(events, mask))

print("Suspicious events (using itertools.compress):")
for event in suspicious_events:
    print(event)
```

Efficiency and Performance Considerations

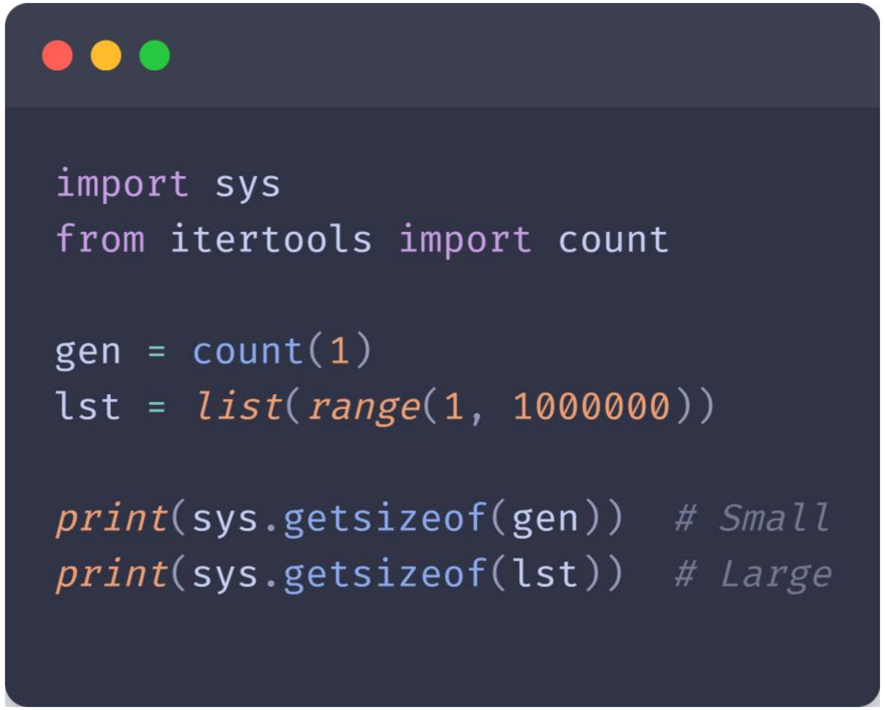
Memory Usage and Lazy Evaluation

Lazy Evaluation

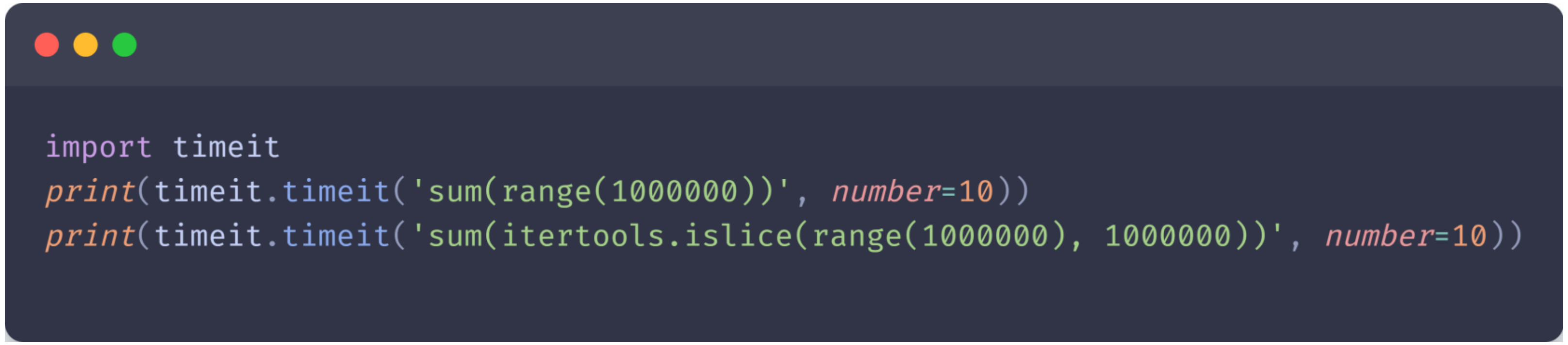
Lazy evaluation generates items on demand, which helps save memory and improve performance in code execution.

Memory Efficiency

Utilizing lazy evaluation minimizes memory usage by creating elements only when required, leading to more efficient code.



```python
import sys
from itertools import count

gen = count(1)
lst = list(range(1, 1000000))

print(sys.getsizeof(gen))  # Small
print(sys.getsizeof(lst))  # Large
```



```python
import timeit
print(timeit.timeit('sum(range(1000000))', number=10))
print(timeit.timeit('sum(itertools.islice(range(1000000), 1000000))', number=10))
```

Conclusion

Power of Itertools

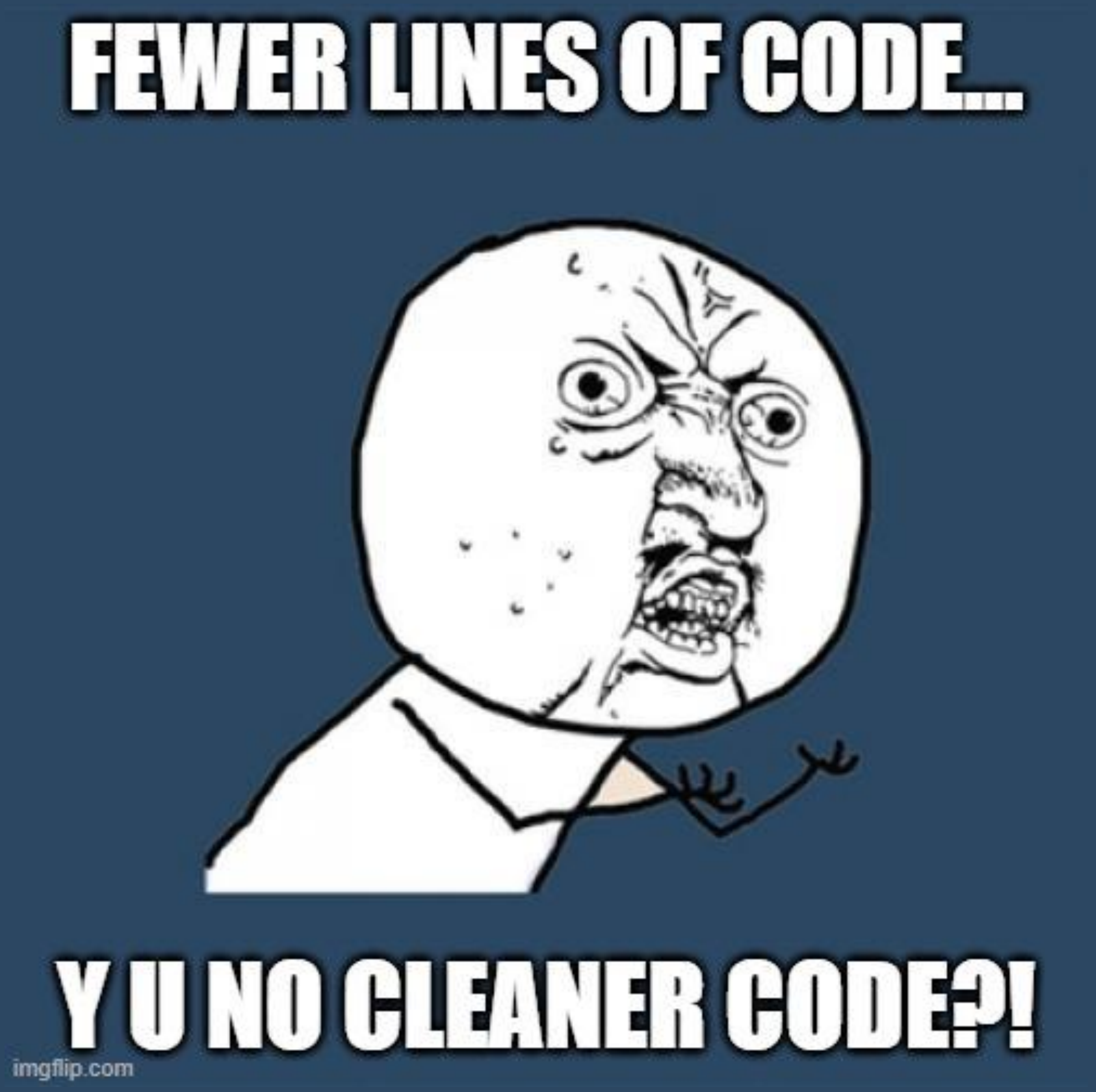
The Itertools module offers powerful functions that can optimize your Python code significantly, enhancing its capabilities.

Efficiency in Code

Using Itertools can lead to more efficient code, reducing the time complexity of operations in your programs.

Cleaner Code

By applying Itertools, you can write cleaner code that is easier to read and maintain, improving overall code quality.



FEWER LINES OF CODE...
Y U NO CLEANER CODE?!
imgflip.com

Learn more!

Itertools Docs
https://docs.python.org/3/library/itertools.html

Python Itertools By Example
https://realpython.com/python-itertools/

Iterators and Iterables in Python
https://realpython.com/python-iterators-iterables/

Transforming Code into beautiful, idiomatic Python
https://www.youtube.com/watch?v=OSGv2VnC0go

Questions?

Gracias!!!

Andres Pineda
Senior Developer @ AFL

Uno Platform Maintainer

RealPython | PythonSDQ | PyCascades

@ajpinedam