Tutorial 2: Iterators, Generators, and Decorators in Python

1. Introduction

This tutorial introduces Python iterators, generators, and decorators. You will learn how to use them to write efficient programs for data streaming, memory-efficient computation, and modifying function behavior without changing the original code.

2. Iterators

2.1 What is an Iterator?

An iterator is an object that allows you to traverse through all elements of a collection (like a list or tuple) one at a time. It implements two main methods: `__iter__()` and `__next__()`.

2.2 Example: Using an Iterator

```
numbers = [1, 2, 3, 4]
iter_numbers = iter(numbers)

print(next(iter_numbers))  # Output: 1
print(next(iter_numbers))  # Output: 2
print(next(iter_numbers))  # Output: 3
print(next(iter_numbers))  # Output: 4
```

2.3 Custom Iterator

```
class CountUp:
    def __init__(self, limit):
        self.limit = limit
        self.current = 0

    def __iter__(self):
        return self

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

counter = CountUp(5)
for num in counter:
    print(num)
```

Output: 1, 2, 3, 4, 5

3. Generators

3.1 What is a Generator?

A generator is a special function that returns an iterator using `yield`. Unlike normal functions that return all data at once, generators produce values on the fly, which is memory-efficient.

3.2 Example: Simple Generator

```
def squares(n):
    for i in range(n):
        yield i * i

for val in squares(5):
    print(val)
```

Output: 0, 1, 4, 9, 16

3.3 Example: Infinite Generator

```
def natural_numbers():
    num = 1
    while True:
        yield num
        num += 1

counter = natural_numbers()
for _ in range(5):
    print(next(counter))
```

Output: 1, 2, 3, 4, 5

3.4 Use Case: Large Data Streaming

```
def read_large_file(file_path):
    with open(file_path) as f:
        for line in f:
            yield line.strip()

for line in read_large_file('large_file.txt'):
    print(line)
```

This approach reads line by line without loading the entire file into memory.

3.5 Generator Expressions

```
squares = (x * x for x in range(10))
for sq in squares:
    print(sq)
```

4. Decorators

4.1 What is a Decorator?

A decorator is a function that takes another function and extends or modifies its behavior without changing its code.

4.2 Example: Logging Decorator

```
def logger(func):
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__} with {args} {kwargs}")
        result = func(*args, **kwargs)
        print(f"Result: {result}")
        return result
    return wrapper

@logger
def add(a, b):
    return a + b

add(5, 7)
```

4.3 Example: Timing Decorator

```
import time

def timing(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"{func.__name__} took {end - start:.4f} seconds")
        return result
    return wrapper

@timing
def compute_sum(n):
    total = sum(range(n))
    return total

compute_sum(1000000)
```

4.4 Chaining Decorators

```
@logger
@timing
def multiply(a, b):
    return a * b

multiply(5, 6)
```

Decorators are applied from bottom to top.

Exercises

1. Write an iterator that returns the first `n` even numbers.
2. Write a generator that yields the Fibonacci sequence up to `n` terms.
3. Implement a decorator that prints “Start” before a function and “End” after it.
4. Write a generator to read a CSV file line by line.
5. Implement an iterator class that iterates over a range in reverse order.
6. Write a generator expression to create a list of cubes of numbers from 1 to 20.
7. Create a decorator to measure memory usage of a function.
8. Write a generator that yields prime numbers indefinitely.
9. Create a decorator to cache results of a function (memoization).
10. Implement a generator that filters only even numbers from a list.
11. Write a generator to simulate a stream of sensor readings.
12. Implement a decorator to log execution time and input arguments.
13. Write an iterator that yields only unique items from a list.
14. Implement a decorator to check if inputs of a function are positive numbers.
15. Write a generator to simulate an infinite sequence of random numbers.