📝 BÀI GIẢNG: GENERATOR, LIST COMPREHENSION VÀ DECORATOR

✍️ **Tác giả: Đặng Kim Thi**

📌 MỤC LỤC

🔷 1. GENERATOR TRONG PYTHON

1.1. Iterator vs Generator

- **Iterator** là một đối tượng có thể duyệt từng phần tử bằng phương thức `__iter__()` và `__next__()`.
- **Generator** là một loại iterator đặc biệt, tạo ra dữ liệu **một cách lười biếng (lazy evaluation)** bằng từ khóa `yield`.

💡 **Khác biệt chính:**

Thuộc tính	Iterator	Generator
Cách tạo	Sử dụng class và `__iter__()` + `__next__()`	Sử dụng `def` với `yield`
Bộ nhớ	Lưu toàn bộ dữ liệu	Sinh dữ liệu từng phần
Hiệu suất	Có thể tốn nhiều RAM	Tiết kiệm RAM, tối ưu hơn

1.2. Cách tạo Generator

Generator được tạo bằng cách sử dụng **hàm bình thường nhưng có `yield` thay vì `return`**.

🔷 **Ví dụ 1: Tạo generator sinh số chẵn**

```
def even_numbers(n):
    for i in range(0, n, 2):
        yield i

gen = even_numbers(10)
print(next(gen))  # Output: 0
print(next(gen))  # Output: 2
```

1.3. Ví dụ minh họa

💡 **Tạo generator sinh dãy Fibonacci vô hạn**

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

gen = fibonacci()
for _ in range(5):
    print(next(gen))  # Output: 0, 1, 1, 2, 3
```

1.4. Lợi ích của Generator

✅ **Tiết kiệm bộ nhớ**: Không lưu trữ toàn bộ dữ liệu.
✅ **Hiệu suất cao**: Chỉ tạo dữ liệu khi cần.
✅ **Dễ đọc, dễ viết hơn iterator thông thường**.

🔷 2. LIST COMPREHENSION

2.1. Khái niệm List Comprehension

List Comprehension là cách rút gọn để tạo danh sách trong Python.

🔷 **Cấu trúc tổng quát:**

```
[biểu_thức for phần_tử in danh_sách if điều_kiện]
```

2.2. Cách sử dụng

💡 **Ví dụ 1: Tạo danh sách bình phương của số chẵn từ 0 đến 10**

```
squares = [x**2 for x in range(11) if x % 2 == 0]
print(squares)  # Output: [0, 4, 16, 36, 64, 100]
```

2.3. So sánh với vòng lặp thông thường

Cách	Vòng lặp for	List Comprehension
Viết code	Dài hơn	Ngắn gọn hơn
Hiệu suất	Chậm hơn	Nhanh hơn

🔷 3. DECORATOR TRONG PYTHON

3.1. Khái niệm Decorator

Decorator là **hàm bọc quanh một hàm khác** để mở rộng chức năng mà không sửa đổi mã gốc.

3.2. Cách tạo Decorator

🔷 **Ví dụ 1: Tạo decorator để in log trước khi gọi hàm**

```
def log_decorator(func):
    def wrapper(*args, **kwargs):
        print(f"Đang gọi hàm {func.__name__}")
        return func(*args, **kwargs)
    return wrapper

@log_decorator
def say_hello():
    print("Hello!")

say_hello()
# Output:
# Đang gọi hàm say_hello
# Hello!
```

🎯 TỔNG KẾT

- **Generator** giúp tối ưu bộ nhớ khi duyệt dữ liệu lớn.
- **List Comprehension** giúp viết mã ngắn gọn, hiệu suất cao.

- **Decorator** mở rộng chức năng của hàm mà không cần chỉnh sửa mã gốc.

✍️ **Tác giả: Đặng Kim Thi**