Clase 3: Control de Flujo en Python: Generadores e Iteradores

Introducción a los Iteradores 🌀

Los **iteradores** son objetos que permiten recorrer una colección (como listas, tuplas o diccionarios) sin necesidad de utilizar índices. Proporcionan una forma eficiente de acceder a los elementos de una colección, permitiendo trabajar con grandes cantidades de datos sin necesidad de almacenarlos todos en memoria al mismo tiempo.

Ejemplo de un Iterador

Para ilustrar el uso de iteradores, consideremos el siguiente ejemplo:

```
# Creando una lista de números del 1 al 4
numeros = [1, 2, 3, 4]

# Creando un iterador a partir de la lista
iterador = iter(numeros)
```

```
# Usando next() para obtener los elementos del iterador
print(next(iterador))  # Salida: 1
print(next(iterador))  # Salida: 2
print(next(iterador))  # Salida: 3
print(next(iterador))  # Salida: 4
# print(next(iterador))  # Genera StopIteration
```

Observaciones 📝

- Al utilizar `next()`, obtenemos el siguiente valor del iterador. Si no hay más elementos, se genera una excepción `StopIteration`.
- Los iteradores son útiles cuando se trabaja con grandes colecciones de datos, ya que permiten acceder a los elementos uno a uno, sin cargar toda la colección en memoria.

Generadores 🌱

Los **generadores** son una forma especial de crear iteradores en Python. Se definen como funciones que utilizan la palabra clave `yield` para devolver un valor en cada iteración. Esto permite que la función "recuerde" su estado entre cada llamada.

Ejemplo de un Generador

Aquí hay un ejemplo simple de un generador que produce números impares:

```
def generador_impares(limite):
    for numero in range(1, limite + 1, 2):
        yield numero

# Usando el generador
for impar in generador_impares(10):
    print(impar)  # Salida: 1, 3, 5, 7, 9
```

Serie de Fibonacci como Generador 🔢

Los generadores son ideales para crear secuencias, como la serie de Fibonacci. A continuación, se presenta un generador que produce la serie de Fibonacci hasta un límite determinado:

```
def generador_fibonacci(limite):
    a, b = 0, 1
    while a < limite:
        yield a
        a, b = b, a + b

# Usando el generador
for numero in generador_fibonacci(10):
    print(numero)  # Salida: 0, 1, 1, 2, 3, 5, 8
```

Ejercicios Prácticos 🧩

Ejercicio 1: Crear un Generador de Números Pares 🎉

Objetivo: Implementar un generador que devuelva números pares hasta un límite dado.

Código:

```
def generador_pares(limite):
    for numero in range(0, limite + 1, 2):
        yield numero

# Ejecución
for par in generador_pares(10):
    print(par)  # Salida: 0, 2, 4, 6, 8, 10
```

Ejercicio 2: Generador de Números Impares 🚀

Objetivo: Crear un generador que devuelva números impares desde un número inicial hasta un límite.

Código:

```
def generador_impares_inicial(inicial, limite):
    for numero in range(inicial, limite + 1, 2):
        yield numero

# Ejecución
for impar in generador_impares_inicial(1, 10):
    print(impar)  # Salida: 1, 3, 5, 7, 9
```

Ejercicio 3: Generador de la Serie de Fibonacci 🧱

Objetivo: Crear un generador para producir la serie de Fibonacci.

Código:

```
def generador_fibonacci(limite):
    a, b = 0, 1
    while a < limite:
        yield a
        a, b = b, a + b

# Ejecución
for numero in generador_fibonacci(20):
    print(numero)  # Salida: 0, 1, 1, 2, 3, 5, 8, 13
```

Conclusión 🎓

Los generadores e iteradores son herramientas poderosas en Python que permiten manejar colecciones de datos de manera eficiente. Su uso es fundamental para optimizar el rendimiento de las aplicaciones que manejan grandes volúmenes de datos.

¡Espero que estos apuntes te sean útiles y te ayuden a comprender mejor los generadores e iteradores en Python! 😊📚