Advanced Python: 반복처리와 함수형 프로그래밍

1. 반복 처리의 기초 개념

- **Iterable (이터러블)**
  - **__iter__() 메서드를 구현한 객체로, iter() 함수를 호출하면 이터레이터를 반환할 수 있는 객체임.**
  - **특징**: 이터러블은 이터레이터와 달리 __next__() 메서드를 가지지 않지만, 이터레이터를 통해 순회할 수 있는 객체임.
  - **예시**: 리스트(list), 튜플(tuple), 문자열(str), 딕셔너리(dict), 세트(set) 등이 이터러블에 해당됨.
  - **중요성**: 이터러블은 for 루프, map(), filter()와 같은 반복문 및 함수에서 기본적으로 사용되는 객체임.
- **Iterator (이터레이터)**
  - **이터러블을 반복 가능한 객체로 만들기 위해 __iter__()와 __next__() 메서드를 가진 객체임.**
  - **상태 관리**: next() 메서드를 호출할 때마다 이터레이터는 현재 상태를 기억하고, 그 다음 요소를 반환하며, 더 이상 반환할 값이 없을 때 StopIteration 예외를 발생시키는 방식임.
  - **장점**: 데이터를 한 번에 하나씩 처리하므로 메모리 사용을 줄이는 데 유리함. 특히 매우 큰 데이터셋을 처리할 때 적합함.
  - **예시**: 사용자 정의 클래스에서 __iter__()와 __next__()를 구현하여 커스텀 이터레이터를 만드는 방식임.
- **Generator (제너레이터)**
  - **함수 안에서 yield 키워드를 사용하여 값을 하나씩 생성하는 이터레이터임.**
  - **특징**: yield를 사용할 때마다 함수가 멈추고, 값이 반환되며 호출자가 그 값을 사용한 후 다음 next() 호출에서 이어서 실행됨.
  - **장점**: 메모리 효율성이 뛰어나며, 필요할 때마다 값을 생성하므로 큰 데이터를 처리할 때 매우 효과적임.
  - **예시**: 파일 처리, 데이터 스트림, 큰 리스트를 처리하는 등 메모리를 절약하는 상황에서 유용함.

2. 메모리 효율적 반복 처리

- **Lazy Evaluation (지연 평가)**
  - **필요한 순간에만 데이터를 계산하고 생성하는 기법임.**
  - **특징**: 제너레이터와 이터레이터는 지연 평가를 기본적으로 적용함. 예를 들어, range(1, 1000000)은 전체 숫자를 메모리에 저장하지 않고 필요할 때마다 값을 생성함.
  - **예시**: 파일에서 한 줄씩 읽어야 하는 경우, with open('large_file.txt') as f: for line in f와 같은 방식으로 처리하면 한 번에 한 줄씩 메모리에 로드되므로 효율적임.
  - **장점**: 모든 데이터를 미리 계산하지 않기 때문에 메모리 제한이 있을 때 유리하며, 스트리밍 데이터나 무한한 데이터 구조에 대해 작업할 때 적합함.
- **Generator Expressions (제너레이터 표현식)**
  - **리스트 컴프리헨션과 유사하게 작성하지만, 소괄호를 사용하여 제너레이터를 반환하는 표현식임.**
  - **특징**: 리스트 컴프리헨션은 전체 리스트를 메모리에 올리지만, 제너레이터 표현식은 필요할 때만 값을 계산하므로 메모리를 절약할 수 있음.
  - **예시**: (x * x for x in range(10**6))은 메모리에 큰 리스트를 올리지 않고 값이 필요할 때만 계산함.

3. 데이터 변환 및 필터링

- **Comprehensions (컴프리헨션)**
  - **리스트 컴프리헨션**: 반복 가능한 객체에서 특정 조건을 적용하여 새로운 리스트를 만드는 방식임.
    - **예시**: [x * 2 for x in range(10) if x % 2 == 0]은 짝수의 두 배 값을 담은 리스트를 생성함.
  - **딕셔너리 컴프리헨션**: 이터러블 객체에서 키와 값을 변환하여 딕셔너리를 만드는 방식임.
    - **예시**: {x: x**2 for x in range(5)}은 키가 숫자이고 값이 그 제곱인 딕셔너리를 생성함.
  - **세트 컴프리헨션**: 조건에 맞는 요소를 담은 세트를 생성하는 방식임.
    - **예시**: {x * 2 for x in range(10)}은 두 배 값을 담은 세트를 생성함.
  - **장점**: 컴프리헨션은 반복문보다 짧고 간결한 문법으로 새로운 컬렉션을 만들 수 있어 코드 가독성을 높임.
- **map(), filter(), reduce() 함수**
  - **map()**: 이터러블의 모든 요소에 함수를 적용하여 새로운 이터러블을 생성하는 함수임.
    - **사용법**: map(function, iterable) 형태로 사용됨.
    - **예시**: list(map(lambda x: x * 2, [1, 2, 3]))은 [2, 4, 6]을 반환함.

- **filter()**: 이터러블의 모든 요소에서 특정 조건을 만족하는 요소만 걸러내는 함수임.
  - **사용법**: filter(function, iterable) 형태로 사용됨.
  - **예시**: list(filter(lambda x: x % 2 == 0, [1, 2, 3, 4]))은 [2, 4]를 반환함.
- **reduce()**: 이터러블의 모든 요소를 하나의 값으로 누적하여 결합하는 함수임.
  - **사용법**: functools.reduce(function, iterable) 형태로 사용됨.
  - **예시**: reduce(lambda x, y: x + y, [1, 2, 3, 4])은 10을 반환함.

4. 함수형 프로그래밍 기법

- **람다 함수 (Lambda Functions)**
  - **정의**: lambda 키워드를 사용하여 익명 함수를 정의하는 방식임.
  - **특징**: 한 줄로 함수를 정의할 수 있으며, 간단한 계산이나 데이터 변환에 유용함.
  - **사용처**: map(), filter(), sorted()와 같은 함수에 함수 인자를 전달할 때 사용됨.
  - **예시**: sorted([(1, 'one'), (3, 'three'), (2, 'two')], key=lambda x: x[1])은 키로 문자열을 사용하여 정렬함.
- **함수형 프로그래밍**
  - **특징**: 데이터를 변화시키지 않고 함수 적용을 통해 처리하는 프로그래밍 패러다임임.
  - **장점**: 데이터의 상태 변화를 피함으로써 프로그램의 복잡성을 줄이고, 가독성과 테스트 가능성을 높임.
  - **고차 함수**: 함수를 인자로 받거나 결과로 반환하는 함수를 고차 함수라고 함.

5. 자원 관리와 코드 개선 기법

- **Context Manager (with 문)**
  - **정의**: 자원을 사용 후 안전하게 해제하도록 도와주는 구문임.
  - **__enter__()와 __exit__()**: with 블록에 들어갈 때 __enter__() 메서드를 호출하고, 블록이 끝나면 __exit__() 메서드를 호출하여 자원을 정리함.
  - **활용**: 파일 입출력, 데이터베이스 연결, 스레드 잠금 등 자원 관리를 자동화해야 하는 경우에 자주 사용됨.
  - **예시**: 커스텀 클래스에서 __enter__()와 __exit__()를 구현하여 자원을 자동으로 해제하는 예시임.
- **Decorator (데코레이터)**
  - **정의**: 기존 함수를 수정하지 않고 추가 기능을 부여할 수 있는 함수임.
  - **특징**: 함수 또는 메서드의 전후에 특정 작업을 수행할 때 유용함.
  - **사용 방법**: 데코레이터 함수는 다른 함수를 인자로 받아 내부에서 호출하거나 추가 기능을 수행하는 함수를 반환함.
  - **예시**: 함수 호출 전후에 특정 동작을 추가하는 데 사용됨.
  - **활용**: 웹 애플리케이션에서 인증을 처리하거나, 함수 호출을 기록하는 로깅, 실행 시간을 측정하는 성능 분석 등에 사용됨.

6. 반복 관련 고급 모듈

- **Itertools 모듈**
  - **정의**: 반복자를 효율적으로 처리할 수 있는 다양한 함수들을 제공하는 파이썬 내장 모듈임.
  - **주요 함수**:
    - **itertools.count(start, step)**: 무한히 증가하는 정수를 생성하는 이터레이터를 반환함.
      - **예시**: itertools.count(10, 2)는 10, 12, 14, ...와 같은 무한 시퀀스를 생성함.
    - **itertools.cycle(iterable)**: 주어진 이터러블을 무한히 반복하는 함수임.
      - **예시**: itertools.cycle('AB')는 'A', 'B', 'A', 'B', ...와 같은 무한 반복을 생성함.
    - **itertools.chain(*iterables)**: 여러 이터러블을 연결하여 하나의 이터러블로 반환하는 함수임.
      - **예시**: itertools.chain([1, 2], ['a', 'b'])는 [1, 2, 'a', 'b']를 반환함.
    - **itertools.islice(iterable, start, stop, step)**: 이터러블에서 슬라이스를 반환하는 함수임. 이터레이터에서 처음부터 필요 없는 요소를 무시하고 원하는 범위만 취할 때 유용함.
      - **예시**: itertools.islice(range(10), 2, 8, 2)는 2, 4, 6을 반환함.
    - **itertools.product(*iterables, repeat=1)**: 카테시안 곱을 생성하는 함수임.
      - **예시**: itertools.product('AB', [1, 2])는 ('A', 1), ('A', 2), ('B', 1), ('B', 2)를 반환함.

1. 반복 처리의 기초 개념

Iterable과 Iterator 예시 (Iterable 객체: 반복가능한 객체, Iterator 객체: 값을 순차적으로 반환하는 객체)

```
class CustomIterable:
    def __init__(self, start, end):
        self.start = start
        self.end = end

    def __iter__(self):
        return CustomIterator(self.start, self.end)

class CustomIterator:
    def __init__(self, start, end):
        self.current = start
        self.end = end

    def __iter__(self):
        return self

    def __next__(self):
        if self.current >= self.end:
            raise StopIteration
        else:
            self.current += 1
            return self.current - 1

# CustomIterable을 사용하여 순회하기
custom_iterable = CustomIterable(1, 5)
for value in custom_iterable:
    print(value)
```

Generator 예시

```
def countdown(start):
    while start > 0:
        yield start
        start -= 1
    yield "Liftoff!"

# 제너레이터 사용하기
for value in countdown(5):
    print(value)
```

2. 메모리 효율적 반복 처리

Lazy Evaluation과 Generator 표현식 예시

```
import sys

# 일반 리스트와 제너레이터 표현식의 메모리 사용 비교
list_comprehension = [x * x for x in range(10000)]
generator_expression = (x * x for x in range(10000))

print(f"List comprehension memory size: {sys.getsizeof(list_comprehension)} bytes")
print(f"Generator expression memory size: {sys.getsizeof(generator_expression)} bytes")

# 제너레이터 표현식의 사용 예
for value in generator_expression:
    if value > 100:
        break
    print(value)
```

3. 데이터 변환 및 필터링

Comprehensions 예시

```
# 리스트 컴프리헨션을 사용하여 짝수만 필터링하고 제곱 계산
numbers = range(1, 20)
even_squares = [x ** 2 for x in numbers if x % 2 == 0]
print(f"Even squares: {even_squares}")

# 딕셔너리 컴프리헨션으로 문자열 길이를 키로 하는 딕셔너리 생성
words = ["apple", "banana", "cherry", "date"]
word_lengths = {word: len(word) for word in words}
print(f"Word lengths: {word_lengths}")

# 세트 컴프리헨션으로 중복 제거
duplicates = [1, 2, 2, 3, 4, 4, 5]
unique_squares = {x ** 2 for x in duplicates}
print(f"Unique squares: {unique_squares}")
```

map(), filter(), reduce() 함수 예시

```
from functools import reduce

# map()을 사용하여 모든 숫자에 2를 곱하기
numbers = [1, 2, 3, 4, 5]
doubled_numbers = list(map(lambda x: x * 2, numbers))
print(f"Doubled numbers: {doubled_numbers}")

# filter()를 사용하여 짝수만 걸러내기
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(f"Even numbers: {even_numbers}")

# reduce()를 사용하여 모든 숫자의 합 계산하기
sum_numbers = reduce(lambda x, y: x + y, numbers)
print(f"Sum of numbers: {sum_numbers}")
```

4. 함수형 프로그래밍 기법

람다 함수와 고차 함수 예시

```
# 리스트를 정렬하면서 람다 함수 사용하기
pairs = [(1, 'one'), (3, 'three'), (2, 'two'), (4, 'four')]
# 숫자가 아닌 문자열을 기준으로 정렬
sorted_pairs = sorted(pairs, key=lambda pair: pair[1])
print(f"Sorted pairs by second element: {sorted_pairs}")

# 고차 함수 예제 - 함수를 인자로 전달하기
def apply_operation(numbers, operation):
    return [operation(num) for num in numbers]

# 숫자 리스트에 제곱을 적용
squared_numbers = apply_operation(numbers, lambda x: x ** 2)
print(f"Squared numbers: {squared_numbers}")
```

5. 자원 관리와 코드 개선 기법

Context Manager (with 문) 예시

```
class ManagedFile:
    def __init__(self, filename):
        self.filename = filename

    def __enter__(self):
        self.file = open(self.filename, 'w')
        print(f"Opening file {self.filename}")
        return self.file

    def __exit__(self, exc_type, exc_value, traceback):
```

```
        if self.file:
            self.file.close()
        print(f"Closing file {self.filename}")

# Context manager를 사용하여 파일 쓰기 작업 수행
with ManagedFile('example.txt') as f:
    f.write('Hello, world!₩n')
    f.write('ManagedFile is working correctly.₩n')
```

Decorator (데코레이터) 예시

```
import time

# 함수의 실행 시간을 측정하는 데코레이터
def timer_decorator(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"Function {func.__name__} took {end_time - start_time:.4f} seconds to complete")
        return result
    return wrapper

# 데코레이터를 사용하여 함수의 실행 시간 측정
@timer_decorator # is equivalent to "timer_decorator(slow_function)"
def slow_function():
    time.sleep(2)
    print("Function finished")

# 함수 호출
slow_function()
```

6. 반복 관련 고급 모듈

Itertools 모듈 예시

```
import itertools

# 1. itertools.count()를 사용한 무한 반복
print("itertools.count() 예시:")
for i in itertools.count(5, 5):
    if i > 20:
        break
    print(i)  # 5, 10, 15, 20

# 2. itertools.cycle()을 사용하여 이터러블 무한 반복
print("₩nitertools.cycle() 예시:")
count = 0
for item in itertools.cycle(['A', 'B', 'C']):
    if count > 5:
        break
    print(item, end=" ")  # A B C A B C
    count += 1

# 3. itertools.chain()을 사용하여 여러 이터러블 연결
print("₩n₩nitertools.chain() 예시:")
list1 = [1, 2, 3]
list2 = ['a', 'b', 'c']
for item in itertools.chain(list1, list2):
    print(item, end=" ")  # 1 2 3 a b c

# 4. itertools.islice()를 사용하여 이터러블 슬라이스
print("₩n₩nitertools.islice() 예시:")
infinite_count = itertools.count(10)
# 10부터 시작하여 5개의 숫자만 가져오기
```

```
for number in itertools.islice(infinite_count, 5):
    print(number, end=" ")  # 10 11 12 13 14

# 5. itertools.product()를 사용하여 카테시안 곱 생성
print("₩n₩nitertools.product() 예시:")
colors = ['red', 'blue']
sizes = ['S', 'M', 'L']
for combination in itertools.product(colors, sizes):
    print(combination)  # ('red', 'S'), ('red', 'M'), ..., ('blue', 'L')

# 6. itertools.permutations()를 사용하여 순열 생성
print("₩nitertools.permutations() 예시:")
items = [1, 2, 3]
for perm in itertools.permutations(items, 2):
    print(perm)  # (1, 2), (1, 3), (2, 1), (2, 3), ...

# 7. itertools.combinations()를 사용하여 조합 생성
print("₩nitertools.combinations() 예시:")
for comb in itertools.combinations(items, 2):
    print(comb)  # (1, 2), (1, 3), (2, 3)

# 8. itertools.groupby()를 사용하여 그룹화
print("₩nitertools.groupby() 예시:")
data = ['A', 'A', 'B', 'B', 'C', 'C', 'C']
# groupby는 반드시 정렬된 데이터에서만 유의미함
for key, group in itertools.groupby(data):
    print(f"{key}: {list(group)}")  # A: ['A', 'A'], B: ['B', 'B'], C: ['C', 'C', 'C']
```