파이썬 프로그래밍

15. 고급함수

00 Contents

❖ 수업 목표

- 이터레이터 개념을 이해할 수 있다.
- 이터레이터의 값을 for문을 사용해 확인할 수 있다.
- 제너레이터를 활용하여 이터레이터를 생성할 수 있다.
- 제너레이터 생성 방법 2가지를 이해할 수 있다.

❖ 세부 목표

1. 이터레이터 개요

❖ 이터레이터

- 값을 하나씩 반환할 수 있는 객체
  - next 함수 호출 시 계속 그 다음 값을 리턴하는 객체

- 이터레이터 요구 조건
  - 다음 2개의 메서드가 구현되어야 함
    - __iter__() 메서드 구현
      - » 객체 자체를 반환
      - » 반복 가능한 객체(iterable)를 반복자(iterator)로 변환
      - » 즉, 객체를 이터러블로 만들어 for 루프나 iter() 함수에서 사용할 수 있도록 함
    - __next__() 메서드 구현
      - » 이터레이터에서 값을 순차적으로 반환
      - » 값이 없으면 StopIteration 예외를 발생시킴

1. 이터레이터 개요

❖ 이터레이터와 이터러블 구분

- 이터러블(iterable)
  - __iter__() 메서드가 있어야 하며, 이 메서드가 이터레이터(iterator) 반환
- 이터레이터(iterator)
  - __next__() 메서드가 있어야 하며, __iter__() 메서드는 자기 자신을 반환

1. 이터레이터 개요

❖ 이터레이터와 이터러블 구분

- 예) 리스트는 이터레이터?
  - a = [1, 2, 3]
  - next(a)

  - 실행 결과
    - » Traceback (most recent call last):
    - » File "<stdin>", line 1, in <module>
    - » TypeError: 'list' object is not an iterator

1. 이터레이터 개요

❖ 이터레이터와 이터러블 구분

- 리스트는 이터러블이지만 이터레이터는 아님: 오류 발생
  - 즉, 반복 가능하다고 해서 이터레이터는 아님
  - 하지만 반복 가능하다면 iter 함수를 이용해 이터레이터로 변환 가능
    - a = [1, 2, 3]
    - iter_a = iter(a)
    - # 리스트는 이터러블이므로 __iter__()를 호출하여 이터레이터 생성 가능
    - print(type(iter_a))

    - 실행 결과
      - » <class 'list_iterator'>

1. 이터레이터 개요

❖ 이터레이터

- next 함수 호출
  - >>> next(ia)
  - 1
  - >>> next(ia)
  - 2
  - >>> next(ia)
  - 3
  - >>> next(ia)
  - Traceback (most recent call last):
  - File "<stdin>", line 1, in <module>
  - StopIteration

  - 더 이상 리턴할 값이 없다면 StopIteration 예외 발생

1. 이터레이터 개요

❖ 이터레이터

- for 문을 이용한 자동 값 호출
  - next 함수의 별도 사용 필요 없음
  - StopIteration 예외 미 발생
  - 예)
  - a = [1, 2, 3]
  - ia = iter(a)
  - for i in ia:
  - 　print(i)
  - for i in ia:
  - 　print(i)

    » 실행 결과
    » 1
    » 2
    » 3

  - 이터레이터는 for문이나 next로 값을 한 번 읽으면, 다시 값을 읽을 수 없음

2. 이터레이터 생성

❖ 이터레이터 만들기

- iter 함수 사용
- 클래스로 이터레이터 생성

2. 이터레이터 생성

- 클래스로 이터레이터 생성
  - 클래스에 __iter__와 __next__라는 2개의 메서드를 구현하여 생성 가능
    - 예) # iterator.py

```
» class MyItertor:
»     def __init__(self, data):
»         self.data = data
»         self.position = 0
»     def __iter__(self):
»         return self
»     def __next__(self):
»         if self.position >= len(self.data):
»             raise StopIteration
»         result = self.data[self.position]
»         self.position += 1
»         return result

» if __name__ == "__main__":
»     i = MyItertor([1,2,3])
»     for item in i:
»         print(item)
```

2. 이터레이터 생성

- 클래스로 이터레이터 생성
  - 클래스에 __iter__와 __next__라는 2개의 메서드를 구현하여 생성 가능
    - __iter__ 메서드와 __next__ 메서드
      - » 생성자 __init__ 메서드와 마찬가지로 클래스에서 특별한 의미를 갖는 메서드
    - __iter__ 메서드를 구현하면 해당 클래스로 생성한 객체는 반복 가능한 객체가 됨
    - __iter__ 메서드는 반복 가능한 객체를 리턴해야 하며 보통 클래스의 객체를 의미하는 self를 리턴
    - 클래스에 __iter__ 메서드를 구현할 경우, 반드시 __next__ 함수를 구현해야 함
    - __next__ 메서드는 반복 가능한 객체의 값을 차례대로 반환하는 역할을 수행
    - __next__ 메서드는 for 문을 수행하거나 next 함수 호출 시 수행
    - MyIterator 객체를 생성할 때 전달한 data를 하나씩 리턴하고 더 이상 리턴할 값이 없으면 StopIteration 예외를 발생시키도록 구현
    - 코드 실행 결과
      - » 1
      - » 2
      - » 3

2. 이터레이터 생성

- 입력 데이터를 역순으로 출력하는 ReverseIterator 클래스 생성
  - 예) # reviterator.py

```
class ReverseItertor:
    def __init__(self, data):
        self.data = data
        self.position = len(self.data) -1
    def __iter__(self):
        return self
    def __next__(self):
        if self.position < 0:
            raise StopIteration
        result = self.data[self.position]
        self.position -= 1
        return result
if __name__ == "__main__":
    i = ReverseItertor([1,2,3])
    for item in i:
        print(item)
```

  - 코드 실행 결과 : 입력받은 데이터를 역순으로 출력

```
3
2
1
```

3. 제너레이터 개요

❖ 제너레이터(generator)

- 이터레이터를 생성하는 함수
  - 제너레이터로 생성한 객체는 이터레이터와 마찬가지로 next 함수 호출 시 그 값을 차례대로 획득 가능
- 제너레이터에서는 차례대로 결과를 반환하고자 return 대신 yield 키워드 사용 (yield: 생성하다)
  - 예)

```
def simple_generator():
    yield 'a'
    yield 'b'
    yield 'c'

g = simple_generator()
```

  - mygen 함수는 yield 구문을 포함하므로 제너레이터
  - 제너레이터 객체는 g = mygen()과 같이 제너레이터 함수를 호출하여 생성

3. 제너레이터 개요

❖ 제너레이터(generator)

- type 명령어로 g 변수가 제너레이터 타입의 객체라는 것 확인 가능
  - >>> type(g)
  - <class 'generator'>

- 제너레이터의 값을 차례대로 획득하기
  - >>> next(g)
  - 'a'

  - 제너레이터 객체 g로 next 함수 실행 시, mygen 함수의 첫 번째 yield 문에 따라 'a' 값을 리턴
  - 제너레이터는 yield라는 문장을 만나면 그 값을 리턴하되 현재 상태를 그대로 기억
    - » 마치 음악을 재생하다가 일시 정지 버튼으로 멈춘 것과 유사

3. 제너레이터 개요

❖ 제너레이터(generator)

- 제너레이터의 값을 차례대로 획득하기
  - 다시 next() 함수 실행
    - » >>> next(g)
    - » 'b'
  - 두 번째 yield 문에 따라 'b' 값을 리턴

  - 계속해서 next 함수를 호출 시, 다음과 같은 결과 출력
    - » >>> next(g)
    - » 'c'
    - » >>> next(g)
    - » Traceback (most recent call last):
    - » File "<stdin>", line 1, in <module>
    - » StopIteration

  - mygen 함수에는 총 3개의 yield 문이 있으므로 네 번째 next를 호출하면 추가 리턴값이 없으므로 StopIteration 예외가 발생

4. 제너레이터 생성

❖ 제너레이터 생성 방법

- def를 이용한 함수 생성
  - 예) # generator.py
    - def mygen():
    - for i in range(1, 1000):
    - result = i * i
    - yield result

    - gen = mygen()
    - print(next(gen))
    - print(next(gen))
    - print(next(gen))

    - mygen 함수는 1부터 1,000까지 각 숫자의 제곱 값을 순서대로 리턴하는 제너레이터
    - 예제 실행 결과 : 총 3번의 next를 호출
      - » 1
      - » 4
      - » 9

4. 제너레이터 생성

❖ 제너레이터 생성 방법

- 제너레이터 표현식(generator expression) 활용 생성
  - 제너레이터 컴프리핸션 형태로 생성
    - 컴프리헨션(comprehension): 파이썬의 자료구조(list, dictionary)에 데이터를 좀 더 쉽고 간결하게 담기 위한 문법
    - 예)
      - » gen = (i * i for i in range(1, 100))
      - » mygen 함수로 만든 제너레이터와 동일하게 동작
      - » 리스트 컴프리헨션(list comprehension), 즉 간단하게 구현하는 구문과 유사

4. 제너레이터 생성

❖ 컴프리헨션(Comprehension)

- 간결하고 효율적인 방식으로 새로운 시퀀스를 생성할 수 있는 문법
- 반복문과 조건문을 활용하여 리스트, 딕셔너리 등을 생성하는 데 사용

▪ 리스트 컴프리헨션

- 기존 리스트나 다른 이터러블 객체로부터 새로운 리스트를 생성
- 문법
  - [expression for item in iterable if condition]

    » expression: 새 리스트의 각 요소를 정의
    » item: 이터러블의 각 요소
    » iterable: 반복 가능한 객체 (리스트, 튜플, 문자열 등)
    » condition (선택적): 조건문이 참인 경우에만 요소를 포함

4. 제너레이터 생성

❖ 컴프리헨션(Comprehension)

- 리스트 컴프리헨션 예제
  - 기본 사용

```
# 기존 리스트의 각 요소를 제곱
numbers = [1, 2, 3, 4]
squared = [x**2 for x in numbers]
print(squared)                    # [1, 4, 9, 16]
```

  - 조건 추가

```
# 짝수만 필터링
even_numbers = [x for x in numbers if x % 2 == 0]
print(even_numbers)               # [2, 4]
```

4. 제너레이터 생성

❖ 컴프리헨션(Comprehension)

- 딕셔너리 컴프리헨션
  - 키-값 쌍을 생성하여 새로운 딕셔너리를 생성
  - 문법
    - {key_expression: value_expression for item in iterable if condition}

      » key_expression: 새 딕셔너리의 key 요소를 정의
      » value_expression: 새 딕셔너리의 value 요소를 정의
      » item: 이터러블의 각 요소
      » iterable: 반복 가능한 객체 (리스트, 튜플, 문자열 등)
      » condition (선택적): 조건문이 참인 경우에만 요소를 포함

4. 제너레이터 생성

❖ 컴프리헨션(Comprehension)

- 딕셔너리 컴프리헨션 예제
  - 기본 사용
    - # 숫자를 키로, 숫자의 제곱을 값으로 하는 딕셔너리 생성
    - squared_dict = {x: x**2 for x in range(5)}
    - print(squared_dict) # {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
  - 조건 추가
    - # 짝수만 포함하는 딕셔너리 생성
    - even_squared_dict = {x: x**2 for x in range(10) if x % 2 == 0}
    - print(even_squared_dict) # {0: 0, 2: 4, 4: 16, 6: 36, 8: 64}

4. 제너레이터 생성

❖ 제너레이터와 이터레이터

- 이터레이터 생성 방법 선택
  - 클래스를 이용해 작성하면 좀 더 복잡한 행동을 구현 가능
  - 간단한 경우, 제너레이터로 이터레이터를 생성

    - 예) (i * i for i in range(1, 100)) => 제너레이터를 이터레이터 클래스로 구현

```
» # gen_iter_class.py
» class MyIterator:
»     def __init__(self):
»         self.data = 1
»
»     def __iter__(self):
»         return self
»
»     def __next__(self):
»         result = self.data * self.data
»         if self.data >= 100:
»             raise StopIteration
»         self.data += 1
»         return result
```

4. 제너레이터 생성

❖ 제너레이터 활용하기

- 리스트 사용 예제
  - 예) longtime_job 함수를 5번 실행해 리스트에 그 결과값을 담고 첫 번째 값 출력

```
» # generator1.py
» import time
» def longtime_job():
»     print("job start")
»     time.sleep(1)                # 1초 지연
»     return "done"
» list_job = [longtime_job() for i in range(5)]
» print(list_job[0])
```

  - 실행 결과

```
» job start
» job start
» job start
» job start
» job start
» job start
» done
```

  - longtime_job 함수는 실행 시간이 약 1초
  - 리스트를 만들 때 5번 함수를 실행하므로 시간이 5초 소요

4. 제너레이터 생성

❖ 제너레이터 활용하기

- 제너레이터 사용 예제
- 예) 제너레이터를 적용하도록 앞선 프로그램을 수정(제너레이터 컴프리핸션 활용)

```
» # generator2.py
» import time
» def longtime_job():
»     print("job start")
»     time.sleep(1)
»     return "done"
» list_job = (longtime_job() for i in range(5))
» print(next(list_job))
```

- 출력 결과

```
» job start
» done
```

- 실행 시 next 함수 1회 실행에 따른 약 1초의 시간만 소요
  - » 제너레이터 표현식에서 longtime_job() 함수는 호출 단계에서 미리 실행 5회가 아닌 next 함수에 따른 결과값이 필요할 때, 1회만 호출
  - » 이렇듯 결과값이 필요할 때까지 연산을 늦추는 방식 => 느긋한 계산법(lazy evaluation)
- 시간이 오래 걸리는 작업을 한꺼번에 처리하기보다는 필요한 경우에만 호출하여 사용할 때 제너레이터가 유용

5. 연습 문제

❖ 다음 리스트를 이터레이터로 변환하고 동작을 확인하시오.

- food = ["김밥", "만두", "양념치킨", "족발", "피자", "쫄면", "라면"]

5. 연습 문제

❖ 파일을 다음과 같이 생성 후, 파일에서 한 줄씩 읽는 제너레이터를 생성하고 사용하시오.

```
def write_file(file_path):
    with open(file_path, 'w') as file:
        file.write(data)
```

❖ 과제

- 1. 이터레이터와 리스트 비교 코드 작성하기
- 2. 이터레이터 값 2가지 방식으로 출력 하기
- 3. 클래스로 이터레이터 생성 코드 작성하기
- 4. 제너레이터 생성 방법 2가지 코드 작성하기

❖ 다음 수업 내용

- 알고리즘(1)
  - 스택(Stack)
  - 큐(Queue)