UNIT 40

제너레이터 사용하기

40 제너레이터 사용하기

제너레이터 사용하기

- 제너레이터는 이터레이터를 생성해주는 함수임
- 이터레이터는 클래스에 __iter__, __next__ 또는 __getitem__ 메서드를 구현해야 하지만 제너레이터는 함수 안에서 yield라는 키워드만 사용하면 끝임
- 제너레이터는 이터레이터보다 훨씬 간단하게 작성할 수 있음
- 제너레이터는 발생자라고 부르기도 함

40.1 제너레이터와 yield 알아보기

제너레이터와 yield 알아보기

- 함수 안에서 yield를 사용하면 함수는 제너레이터가 되며 yield에는 값(변수)을 지정함

• **yield 값**

yield.py

```
def number_generator():
    yield 0
    yield 1
    yield 2

for i in number_generator():
    print(i)
```

실행 결과

```
0
1
2
```

40.1 제너레이터와 yield 알아보기

제너레이터 객체가 이터레이터인지 확인하기

- 다음과 같이 dir 함수로 메서드 목록을 확인해보자

```
>>> g = number_generator()
>>> g
<generator object number_generator at 0x03A190F0>
>>> dir(g)
['__class__', '__del__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute
__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__lt__', '__name__', '__ne_
_', '__new__', '__next__', '__qualname__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof_
_', '__str__', '__subclasshook__', 'close', 'gi_code', 'gi_frame', 'gi_running', 'gi_yieldfrom', 'send', 'thro
w']
```

- number_generator 함수를 호출하면 제너레이터 객체(generator object)가 반환됨
- 이 객체를 dir 함수로 살펴보면 이터레이터에서 볼 수 있는 __iter__, __next__ 메서드가 들어있음

40.1 제너레이터와 yield 알아보기

제너레이터 객체가 이터레이터인지 확인하기

```
>>> g.__next__()
0
>>> g.__next__()
1
>>> g.__next__()
2
>>> g.__next__()
Traceback (most recent call last):
  File "<pyshell#29>", line 1, in <module>
    g.__next__()
StopIteration
```

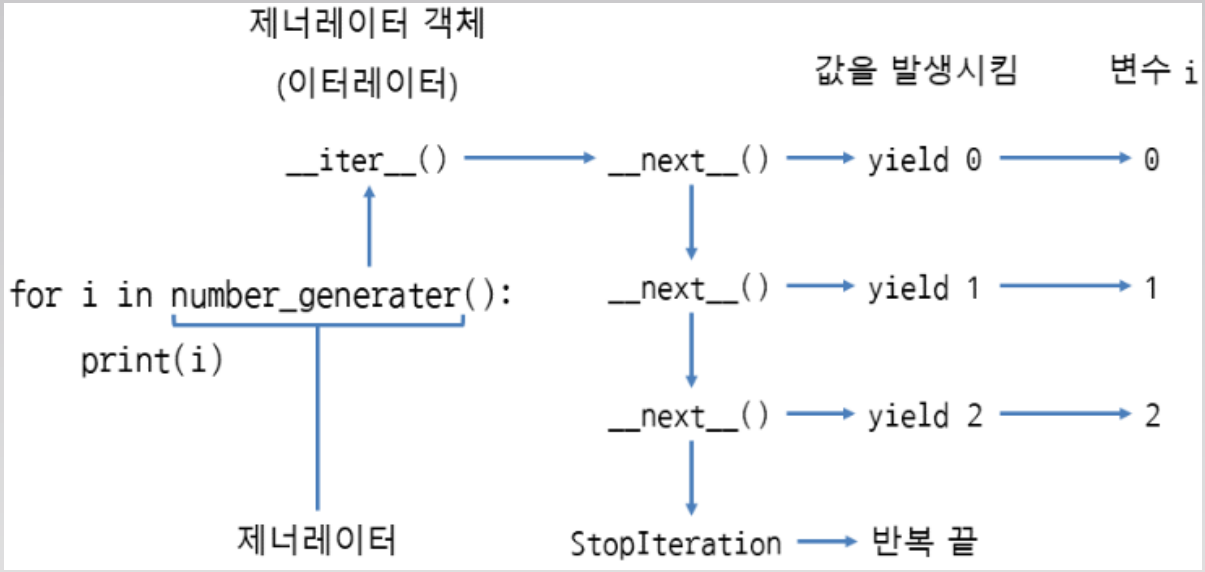
- 함수에 yield만 사용해서 간단하게 이터레이터를 구현할 수 있음
- 이터레이터는 __next__ 메서드 안에서 직접 return으로 값을 반환했지만 제너레이터는 yield에 지정한 값이 __next__ 메서드(next 함수)의 반환값으로 나옴
- 이터레이터는 raise로 StopIteration 예외를 직접 발생시켰지만 제너레이터는 함수의 끝까지 도달하면 StopIteration 예외가 자동으로 발생함
- 제너레이터는 제너레이터 객체에서 __next__ 메서드를 호출할 때마다 함수 안의 yield까지 코드를 실행하며 yield에서 값을 발생시킴(generate)

40.1 제너레이터와 yield 알아보기

for와 제너레이터

- 다음과 같이 for 반복문은 반복할 때마다 __next__를 호출하므로 yield에서 발생시킨 값을 가져옴

▼ 그림 for 반복문과 제너레이터



40.1 제너레이터와 yield 알아보기

for와 제너레이터

- 제너레이터 객체에서 __iter__를 호출하면 self를 반환하므로 같은 객체가 나옴(제너레이터 함수 호출 〉 제너레이터 객체 〉 __iter__는 self 반환 〉 제너레이터 객체)
- yield를 사용하면 값을 함수 바깥으로 전달하면서 코드 실행을 함수 바깥에 양보함
- yield는 현재 함수를 잠시 중단하고 함수 바깥의 코드가 실행되도록 만듦

40.1 제너레이터와 yield 알아보기

yield의 동작 과정 알아보기

- 그럼 yield의 동작 과정을 알아보기 위해 for 반복문 대신 next 함수로 __next__ 메서드를 직접 호출해보자

- 변수 = next(제너레이터객체)

yield_next.py

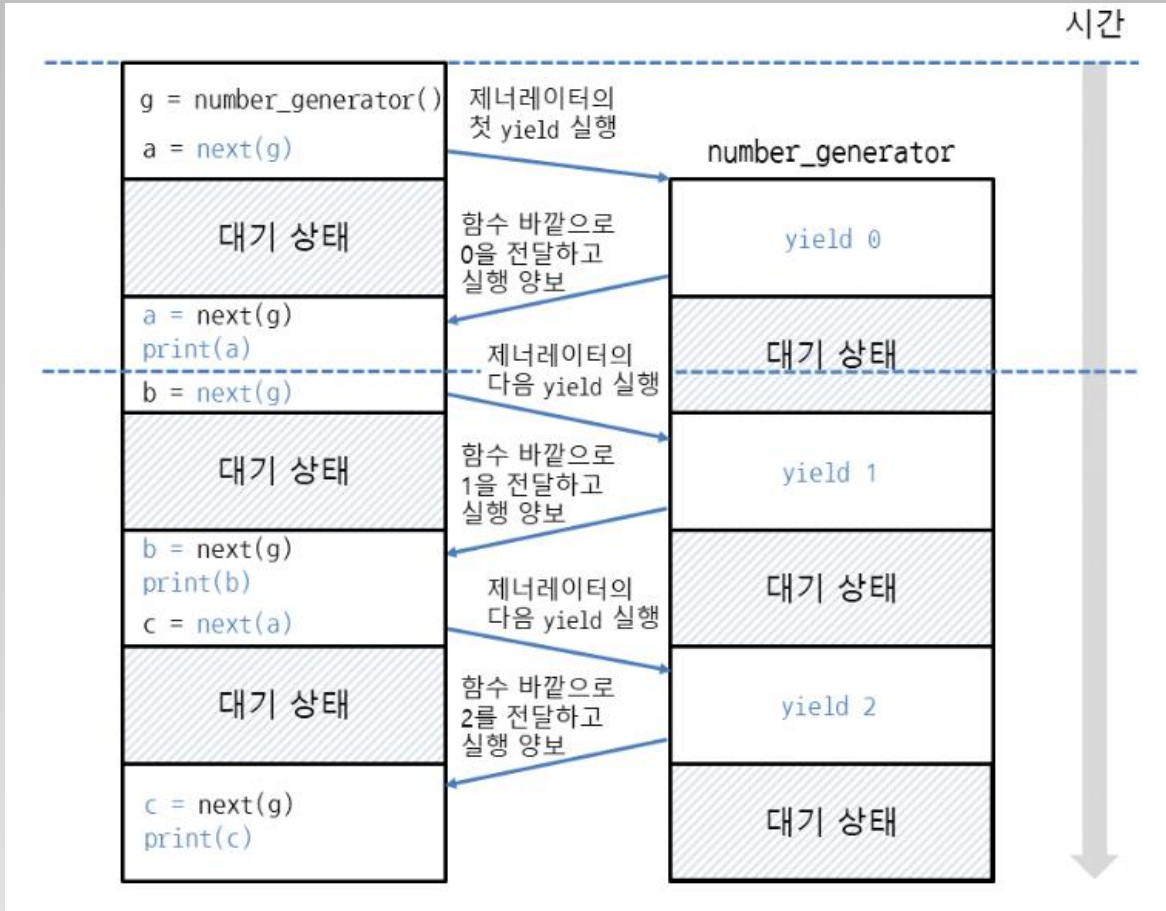
```
def number_generator():
    yield 0    # 0을 함수 바깥으로 전달하면서 코드 실행을 함수 바깥에 양보
    yield 1    # 1을 함수 바깥으로 전달하면서 코드 실행을 함수 바깥에 양보
    yield 2    # 2를 함수 바깥으로 전달하면서 코드 실행을 함수 바깥에 양보

g = number_generator()

a = next(g)    # yield를 사용하여 함수 바깥으로 전달한 값은 next의 반환값으로 나옴
print(a)       # 0

b = next(g)
print(b)       # 1

c = next(g)
print(c)       # 2
```

실행 결과

```
0
1
2
```

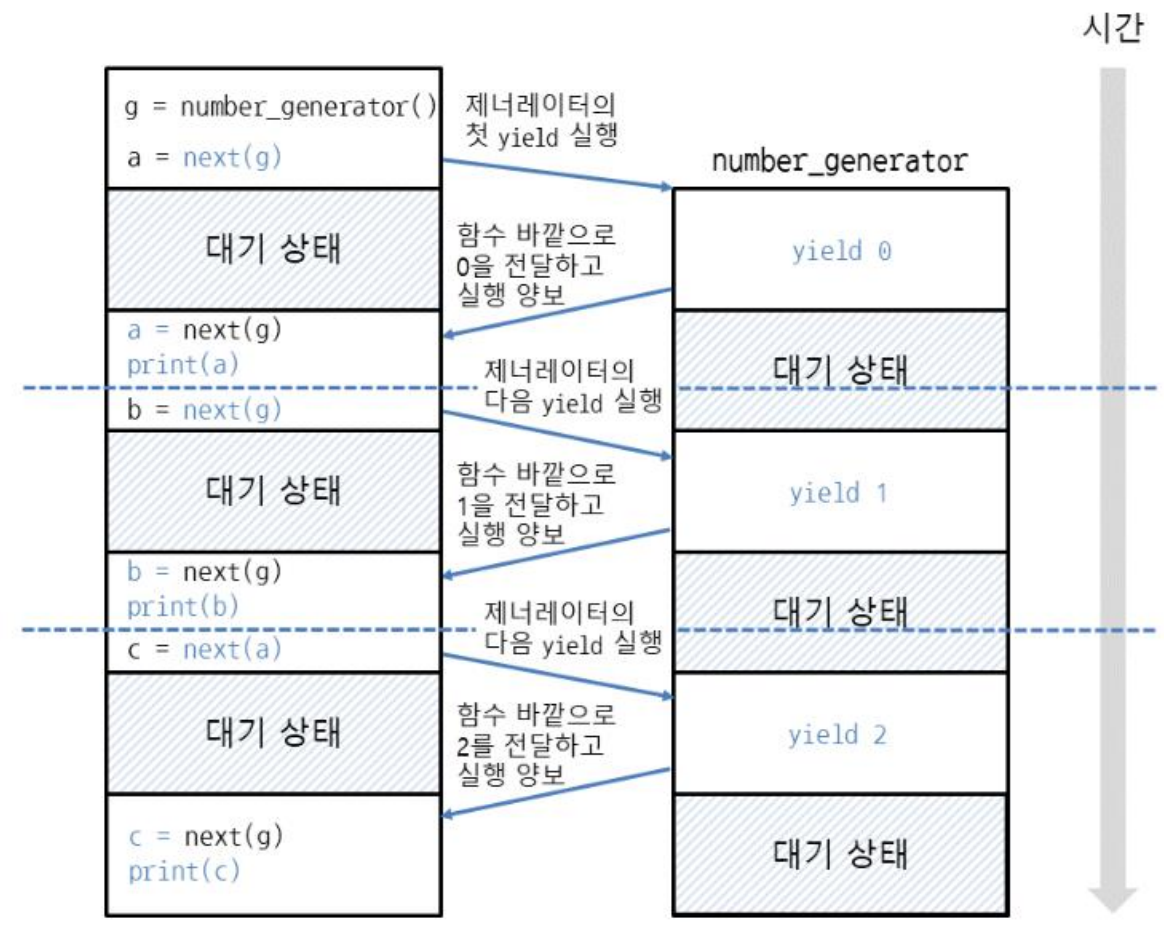
40.1 제너레이터와 yield 알아보기

▼ 그림 yield 0의 실행 양보



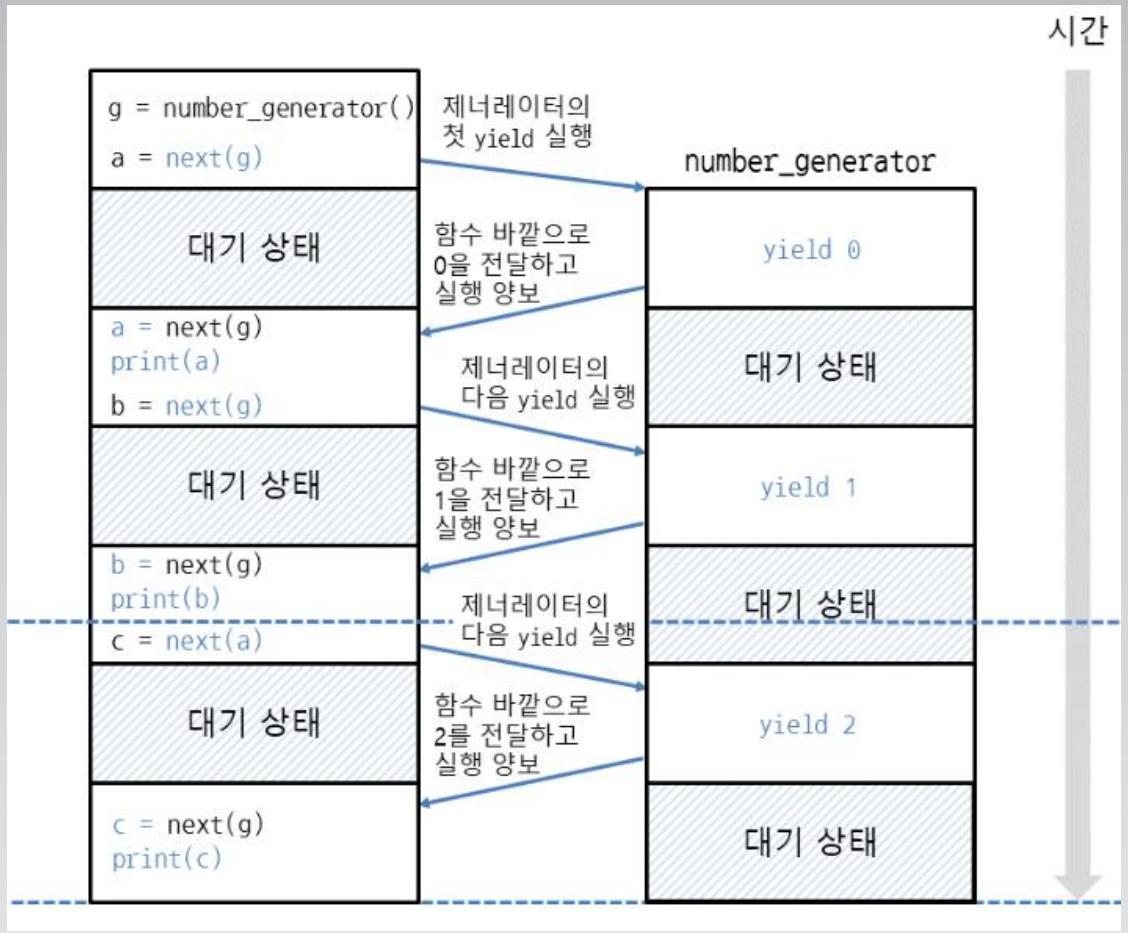
40.1 제너레이터와 yield 알아보기

▼ 그림 yield 1의 실행 양보



40.1 제너레이터와 yield 알아보기

▼ 그림 yield 2의 실행 양보



40.2 제너레이터 만들기

제너레이터 만들기

- 이번에는 range(횟수)처럼 동작을 하는 제너레이터를 만들어보자

generator.py

```
def number_generator(stop):
    n = 0               # 숫자는 0부터 시작
    while n < stop:     # 현재 숫자가 반복을 끝낼 숫자보다 작을 때 반복
        yield n         # 현재 숫자를 바깥으로 전달
        n += 1          # 현재 숫자를 증가시킴

for i in number_generator(3):
    print(i)
```

실행 결과

```
0
1
2
```

40.2 제너레이터 만들기

제너레이터 만들기

- next 함수(__next__ 메서드)도 3번 사용할 수 있음

```
>>> g = number_generator(3)
>>> next(g)
0
>>> next(g)
1
>>> next(g)
2
>>> next(g)
Traceback (most recent call last):
  File "<pyshell#100>", line 1, in <module>
    next(g)
StopIteration
```

40.2 제너레이터 만들기

yield에서 함수 호출하기

- 다음은 리스트에 들어있는 문자열을 대문자로 변환하여 함수 바깥으로 전달함

generator_yield_function.py

```
def upper_generator(x):
    for i in x:
        yield i.upper()    # 함수의 반환값을 바깥으로 전달

fruits = ['apple', 'pear', 'grape', 'pineapple', 'orange']
for i in upper_generator(fruits):
    print(i)
```

실행 결과

```
APPLE
PEAR
GRAPE
PINEAPPLE
ORANGE
```

40.3 yield from으로 값을 여러 번 바깥으로 전달하기

yield from으로 값을 여러 번 바깥으로 전달하기

- 값을 여러 번 바깥으로 전달할 때는 for 또는 while 반복문으로 반복하면서 yield를 사용함

generate_for_yield.py

```
def number_generator():
    x = [1, 2, 3]
    for i in x:
        yield i

for i in number_generator():
    print(i)
```

실행 결과

```
1
2
3
```

40.3 yield from으로 값을 여러 번 바깥으로 전달하기

yield from으로 값을 여러 번 바깥으로 전달하기

- 이런 경우에는 매번 반복문을 사용하지 않고, yield from을 사용하면 됨

```
• yield from 반복가능한객체
• yield from 이터레이터
• yield from 제너레이터객체
```

- 그럼 yield from에 리스트를 지정해서 숫자 1, 2, 3을 바깥으로 전달해보자

generator_yield_from_iterable.py

```
def number_generator():
    x = [1, 2, 3]
    yield from x    # 리스트에 들어있는 요소를 한 개씩 바깥으로 전달

for i in number_generator():
    print(i)
```

실행 결과

```
1
2
3
```

40.3 yield from으로 값을 여러 번 바깥으로 전달하기

yield from으로 값을 여러 번 바깥으로 전달하기

- yield from을 한 번 사용하여 값을 세 번 바깥으로 전달함
- next 함수(__next__ 메서드)를 세 번 호출할 수 있음

```
>>> g = number_generator()
>>> next(g)
1
>>> next(g)
2
>>> next(g)
3
>>> next(g)
Traceback (most recent call last):
  File "<pyshell#105>", line 1, in <module>
    next(g)
StopIteration
```

40.3 yield from으로 값을 여러 번 바깥으로 전달하기

yield from으로 값을 여러 번 바깥으로 전달하기

generator_yield_from_generator.py

```
def number_generator(stop):
    n = 0
    while n < stop:
        yield n
        n += 1

def three_generator():
    yield from number_generator(3)    # 숫자를 세 번 바깥으로 전달

for i in three_generator():
    print(i)
```

실행 결과

```
0
1
2
```

- for 반복문에 three_generator()를 사용하면 숫자를 세 번 출력함(next 함수 또는 __next__ 메서드도 세 번 호출 가능)