

# 파이썬 프로그래밍

## 강의노트 #17

---

# 이터레이터(Iterator) 활용

---

## 이터레이터 사용하기

- 이터레이터(iterator)는 값을 차례대로 꺼낼 수 있는 객체(object)임
- 파이썬에서는 이터레이터만 생성하고 값이 필요한 시점이 되었을 때 값을 만드는 방식을 사용함
- 데이터 생성을 뒤로 미루는 것인데 이런 방식을 지연 평가(lazy evaluation)라고 함
- 이터레이터는 반복자라고 부르기도 함

# 이터레이터(Iterator) 활용

## 반복 가능한 객체 알아보기

- 반복 가능한 객체는 말 그대로 반복할 수 있는 객체인데 우리가 흔히 사용하는 문자열, 리스트, 딕셔너리, 세트가 반복 가능한 객체임
- 요소가 여러 개 들어있고, 한 번에 하나씩 꺼낼 수 있는 객체임
- 객체가 반복 가능한 객체인지 알아보는 방법은 객체에 `_iter_` 메서드가 들어있는지 확인해보면 됨
  - `dir(객체)`

```
>>> dir([1, 2, 3])
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__', '__format__',
 '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__',
 '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__',
 '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse',
 'sort']
```

```
>>> [1, 2, 3].__iter__()
<list_iterator object at 0x03616630>
```

# 이터레이터(Iterator) 활용

## 반복 가능한 객체 알아보기

- 리스트의 이터레이터를 변수에 저장한 뒤 `_next_` 메서드를 호출해보면 요소를 차례대로 꺼낼 수 있음

```
>>> it = [1, 2, 3].__iter__()
>>> it.__next__()
1
>>> it.__next__()
2
>>> it.__next__()
3
>>> it.__next__()
Traceback (most recent call last):
  File "<pyshell#48>", line 1, in <module>
    it.__next__()
StopIteration
```

- 이터레이터는 `_next_`로 요소를 계속 꺼내다가 꺼낼 요소가 없으면 `StopIteration` 예외를 발생시켜서 반복을 끝냄

# 이터레이터(Iterator) 활용

## 반복 가능한 객체 알아보기

- 리스트뿐만 아니라 문자열, 딕셔너리, 세트도 `__iter__`를 호출하면 이터레이터가 나옴
- 이터레이터에서 `__next__`를 호출하면 차례대로 값을 꺼냄

```
>>> 'Hello, world!'.__iter__()
<str_iterator object at 0x03616770>
>>> {'a': 1, 'b': 2}.__iter__()
<dict_keyiterator object at 0x03870B10>
>>> {1, 2, 3}.__iter__()
<set_iterator object at 0x03878418>
```

- 리스트, 문자열, 딕셔너리, 세트는 요소가 눈에 보이는 반복 가능한 객체임

# 이터레이터(Iterator) 활용

## 반복 가능한 객체 알아보기

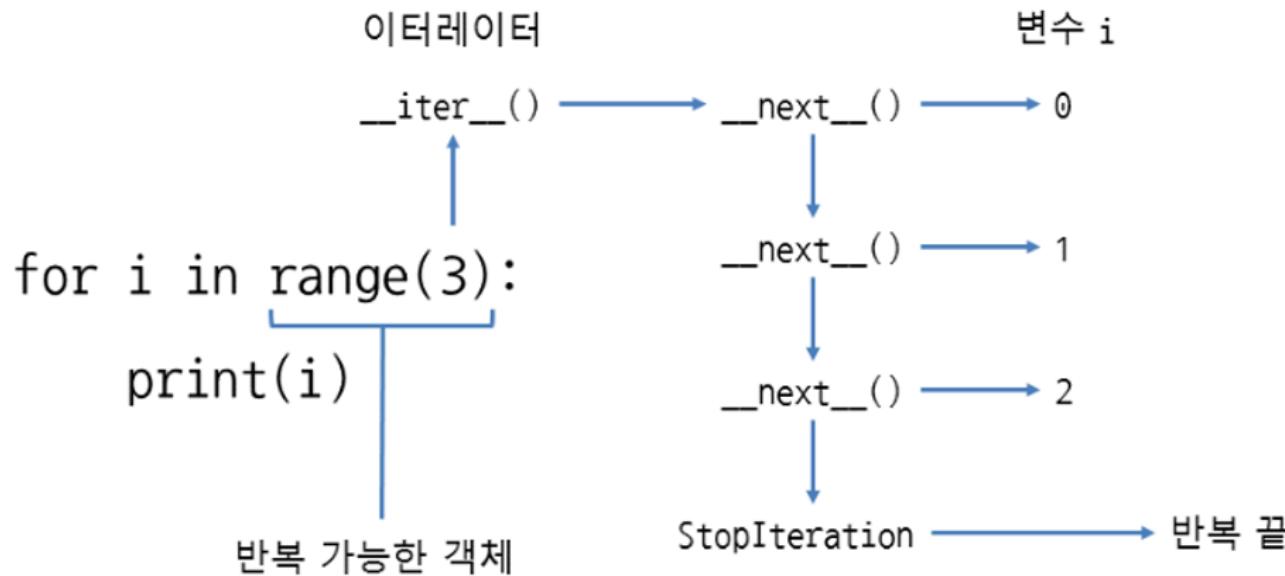
- 다음과 같이 range(3)에서 \_\_iter\_\_로 이터레이터를 얻어낸 뒤 \_\_next\_\_ 메서드를 호출해보자

```
>>> it = range(3).__iter__()
>>> it.__next__()
0
>>> it.__next__()
1
>>> it.__next__()
2
>>> it.__next__()
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    it.__next__()
StopIteration
```

# 이터레이터(Iterator) 활용

for와 반복 가능한 객체

- 다음과 같이 for에 range(3)을 사용했다면 먼저 range에서 \_\_iter\_\_로 이터레이터를 얻음
  - 한 번 반복할 때마다 이터레이터에서 \_\_next\_\_로 숫자를 꺼내서 i에 저장하고, 지정된 숫자 3이 되면 StopIteration을 발생시켜서 반복을 끝냄
- ▼ 그림 39-1 for에서 range의 동작 과정



# 이터레이터(Iterator) 활용

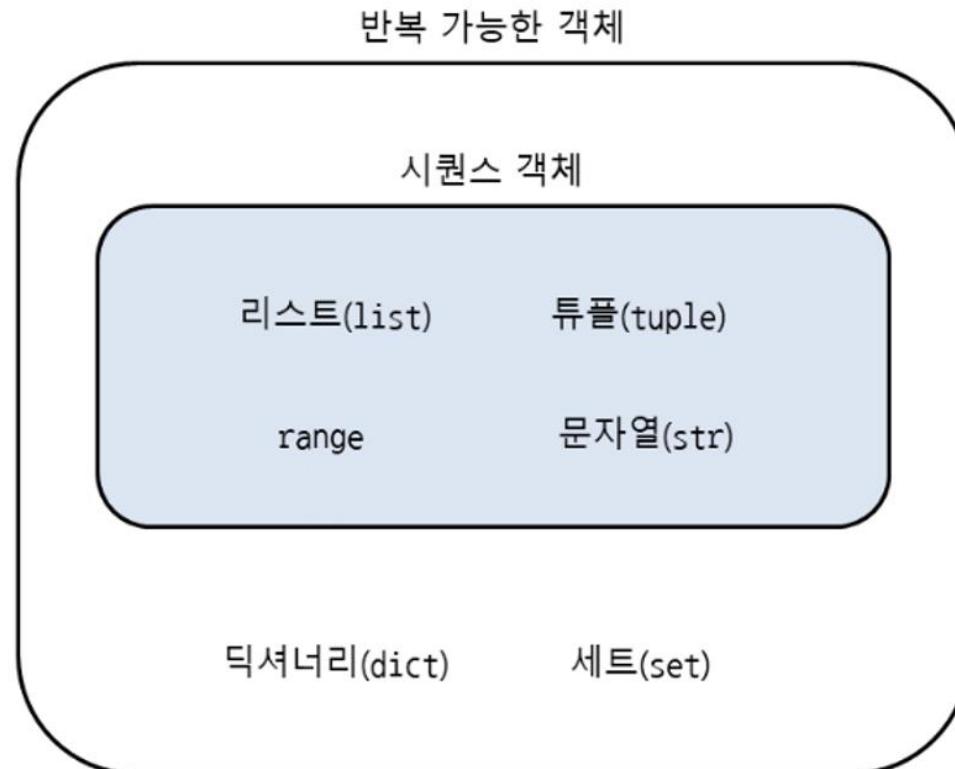
---

## for와 반복 가능한 객체

- 반복 가능한 객체는 요소를 한 번에 하나씩 가져올 수 있는 객체이고, 이터레이터는 `_next_` 메서드를 사용해서 차례대로 값을 꺼낼 수 있는 객체임
- 반복 가능한 객체(`iterable`)와 이터레이터(`iterator`)는 별개의 객체이므로 둘은 구분해야 함
- 반복 가능한 객체에서 `_iter_` 메서드로 이터레이터를 얻음

# 이터레이터(Iterator) 활용

▼ 그림 39-2 반복 가능한 객체는 시퀀스 객체를 포함



# 이터레이터(Iterator) 활용

## 이터레이터 만들기

- 간단하게 range(횟수)처럼 동작하는 이터레이터임

```
class 이터레이터이름:  
    def __iter__(self):  
        코드  
  
    def __next__(self):  
        코드
```

# 이터레이터(Iterator) 활용

## 이터레이터 만들기

iterator.py

```
class Counter:
    def __init__(self, stop):
        self.current = 0      # 현재 숫자 유지, 0부터 지정된 숫자 직전까지 반복
        self.stop = stop      # 반복을 끝낼 숫자

    def __iter__(self):
        return self          # 현재 인스턴스를 반환

    def __next__(self):
        if self.current < self.stop:      # 현재 숫자가 반복을 끝낼 숫자보다 작을 때
            r = self.current           # 반환할 숫자를 변수에 저장
            self.current += 1          # 현재 숫자를 1 증가시킴
            return r                  # 숫자를 반환
        else:                         # 현재 숫자가 반복을 끝낼 숫자보다 크거나 같을 때
            raise StopIteration      # 예외 발생

    for i in Counter(3):
        print(i, end=' ')
```

실행 결과

0 1 2

# 이터레이터(Iterator) 활용

## 이터레이터 만들기

- 클래스로 이터레이터를 작성하려면 `_init_` 메서드를 만듦

```
def __init__(self, stop):
    self.current = 0      # 현재 숫자 유지, 0부터 지정된 숫자 직전까지 반복
    self.stop = stop      # 반복을 끝낼 숫자
```

- 이 객체는 리스트, 문자열, 딕셔너리, 세트, range처럼 `_iter_`를 호출해줄 반복 가능한 객체(iterable)가 없으므로 현재 인스턴스를 반환하면 됨
- 이 객체는 반복 가능한 객체이면서 이터레이터임

```
def __iter__(self):
    return self           # 현재 인스턴스를 반환
```

# 이터레이터(Iterator) 활용

## 이터레이터 만들기

```
def __next__(self):
    if self.current < self.stop:
        r = self.current          # 현재 숫자가 반복을 끝낼 숫자보다 작을 때
        self.current += 1         # 반환할 숫자를 변수에 저장
        return r                 # 현재 숫자를 1 증가시킴
    else:
        raise StopIteration     # 숫자를 반환
                                # 현재 숫자가 반복을 끝낼 숫자보다 크거나 같을 때
                                # 예외 발생
```

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

- 이터레이터를 만들 때는 `_init_` 메서드에서 초기값, `_next_` 메서드에서 조건식과 현재값 부분을 주의해야 함
- 이 부분이 잘못되면 미묘한 버그가 생길 수 있음

# 이터레이터(Iterator) 활용

## 이터레이터 언패킹

- 다음과 같이 Counter()의 결과를 변수 여러 개에 할당할 수 있음
- 이터레이터가 반복하는 횟수와 변수의 개수는 같아야 함

```
>>> a, b, c = Counter(3)
>>> print(a, b, c)
0 1 2
>>> a, b, c, d, e = Counter(5)
>>> print(a, b, c, d, e)
0 1 2 3 4
```

- 사실 우리가 자주 사용하는 map도 이터레이터임
- a, b, c = map(int, input().split())처럼 언패킹으로 변수 여러 개에 값을 할당할 수 있음

# 이터레이터(Iterator) 활용

인덱스로 접근할 수 있는 이터레이터 만들기

```
class 이터레이터이름:  
    def __getitem__(self, 인덱스):  
        코드
```

iterator\_getitem.py

```
class Counter:  
    def __init__(self, stop):  
        self.stop = stop  
  
    def __getitem__(self, index):  
        if index < self.stop:  
            return index  
        else:  
            raise IndexError  
  
print(Counter(3)[0], Counter(3)[1], Counter(3)[2])  
  
for i in Counter(3):  
    print(i, end=' ')
```

실행 결과

```
0 1 2  
0 1 2
```

# 이터레이터(Iterator) 활용

## 인덱스로 접근할 수 있는 이터레이터 만들기

- `_init_` 메서드부터는 Counter(3)처럼 반복을 끝낼 숫자를 받았으므로 `self.stop`에 `stop`을 넣어줌

```
class Counter:  
    def __init__(self, stop):  
        self.stop = stop # 반복을 끝낼 숫자
```

- 클래스에서 `_getitem_` 메서드를 구현하면 인덱스로 접근할 수 있는 이터레이터가 됨
- Counter(3)과 같이 반복을 끝낼 숫자가 3이면 인덱스는 2까지 지정할 수 있음

```
def __getitem__(self, index): # 인덱스를 받음  
    if index < self.stop: # 인덱스가 반복을 끝낼 숫자보다 작을 때  
        return index # 인덱스를 반환  
    else: # 인덱스가 반복을 끝낼 숫자보다 크거나 같을 때  
        raise IndexError # 예외 발생
```

- 이렇게 하면 Counter(3)[0]처럼 이터레이터를 인덱스로 접근할 수 있음
- 반복할 숫자와 인덱스가 같아서 `index`를 그대로 반환했지만, `index`와 식을 조합해서 다른 숫자를 만드는 방식으로 활용할 수 있음

# 이터레이터(Iterator) 활용

## iter, next 함수 활용하기

- iter는 객체의 `__iter__` 메서드를 호출해주고, next는 객체의 `__next__` 메서드를 호출해줌

```
>>> it = iter(range(3))
>>> next(it)
0
>>> next(it)
1
>>> next(it)
2
>>> next(it)
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    next(it)
StopIteration
```

- iter는 반복 가능한 객체에서 이터레이터를 반환하고, next는 이터레이터에서 값을 차례대로 꺼냄

# 이터레이터(Iterator) 활용

## iter

- iter는 반복을 끝낼 값을 지정하면 특정 값이 나올 때 반복을 끝냄
- 이 경우에는 반복 가능한 객체 대신 호출 가능한 객체(callable)를 넣어줌
- 참고로 반복을 끝낼 값은 sentinel이라고 부르는데 감시병이라는 뜻임
- 반복을 감시하다가 특정 값이 나오면 반복을 끝낸다고 해서 sentinel임
- **iter(호출가능한객체, 반복을끝낼값)**
- 호출 가능한 객체를 넣어야 하므로 매개변수가 없는 함수 또는 람다 표현식으로 만들어줌

```
>>> import random
>>> it = iter(lambda : random.randint(0, 5), 2)
>>> next(it)
0
>>> next(it)
3
>>> next(it)
1
>>> next(it)
Traceback (most recent call last):
  File "<pyshell#37>", line 1, in <module>
    next(it)
StopIteration
```

# 이터레이터(Iterator) 활용

## iter

- 다음과 같이 for 반복문에 넣어서 사용할 수도 있음

```
>>> import random
>>> for i in iter(lambda : random.randint(0, 5), 2):
...     print(i, end=' ')
...
3 1 4 0 5 3 3 5 0 4 1
```

- iter 함수를 활용하면 if 조건문으로 매번 숫자가 2인지 검사하지 않아도 되므로 코드가 좀 더 간단해짐

```
import random

while True:
    i = random.randint(0, 5)
    if i == 2:
        break
    print(i, end=' ')
```

# 이터레이터(Iterator) 활용

## next

- next는 기본값을 지정할 수 있음
- 기본값을 지정하면 반복이 끝나더라도 StopIteration이 발생하지 않고 기본값을 출력함
- 반복할 수 있을 때는 해당 값을 출력하고, 반복이 끝났을 때는 기본값을 출력함
- 다음은 range(3)으로 0, 1, 2 세 번 반복하는데 next에 기본값으로 10을 지정함
  - `next(반복가능한객체, 기본값)`

```
>>> it = iter(range(3))
>>> next(it, 10)
0
>>> next(it, 10)
1
>>> next(it, 10)
2
>>> next(it, 10)
10
>>> next(it, 10)
10
```

# 제너레이터(generator) 활용

---

## 제너레이터 사용하기

- 제너레이터는 이터레이터를 생성해주는 함수임
- 이터레이터는 클래스에 `_iter_`, `_next_` 또는 `_getitem_` 메서드를 구현해야 하지만 제너레이터는 함수 안에서 `yield`라는 키워드만 사용하면 끝임
- 제너레이터는 이터레이터보다 훨씬 간단하게 작성할 수 있음
- 제너레이터는 발생자라고 부르기도 함

# 제너레이터(generator) 활용

## 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
```

# 제너레이터(generator) 활용

## 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', 'throw']
```

- number\_generator 함수를 호출하면 제너레이터 객체(generator object)가 반환됨
- 이 객체를 dir 함수로 살펴보면 이터레이터에서 볼 수 있는 \_\_iter\_\_, \_\_next\_\_ 메서드가 들어있음

# 제너레이터(generator) 활용

## 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
```

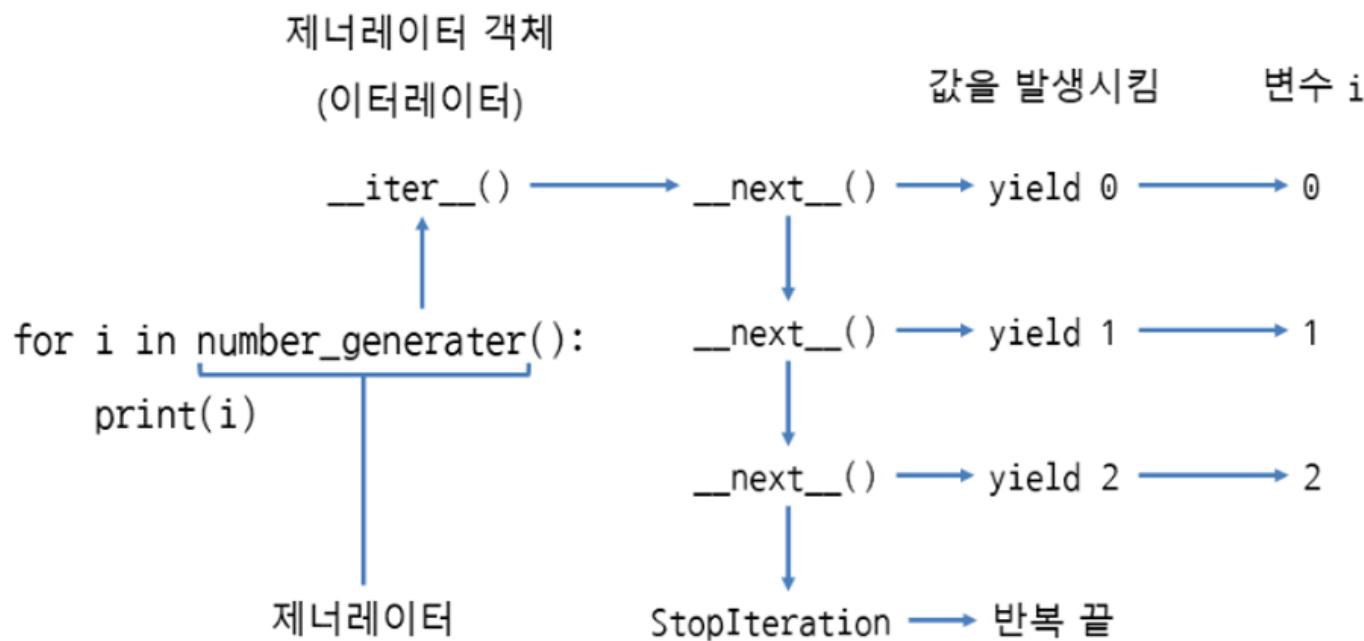
- 함수에 yield만 사용해서 간단하게 이터레이터를 구현할 수 있음
- 이터레이터는 `_next_` 메서드 안에서 직접 `return`으로 값을 반환했지만 제너레이터는 `yield`에 지정한 값이 `_next_` 메서드(`next` 함수)의 반환값으로 나옴
- 이터레이터는 `raise`로 `StopIteration` 예외를 직접 발생시켰지만 제너레이터는 함수의 끝까지 도달하면 `StopIteration` 예외가 자동으로 발생함
- 제너레이터는 제너레이터 객체에서 `_next_` 메서드를 호출할 때마다 함수 안의 `yield`까지 코드를 실행하며 `yield`에서 값을 발생시킴(generate)

# 제너레이터(generator) 활용

## for와 제너레이터

- 다음과 같이 for 반복문은 반복할 때마다 `_next_()`를 호출하므로 `yield`에서 발생시킨 값을 가져옴

### ▼ 그림 40-1 for 반복문과 제너레이터



# 제너레이터(generator) 활용

## for와 제너레이터

- 제너레이터 객체에서 `_iter_`를 호출하면 `self`를 반환하므로 같은 객체가 나옴(제너레이터 함수 호출 > 제너레이터 객체 > `_iter_`는 `self` 반환 > 제너레이터 객체)
- `yield`를 사용하면 값을 함수 바깥으로 전달하면서 코드 실행을 함수 바깥에 양보함
- `yield`는 현재 함수를 잠시 중단하고 함수 바깥의 코드가 실행되도록 만듬

# 제너레이터(generator) 활용

## 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

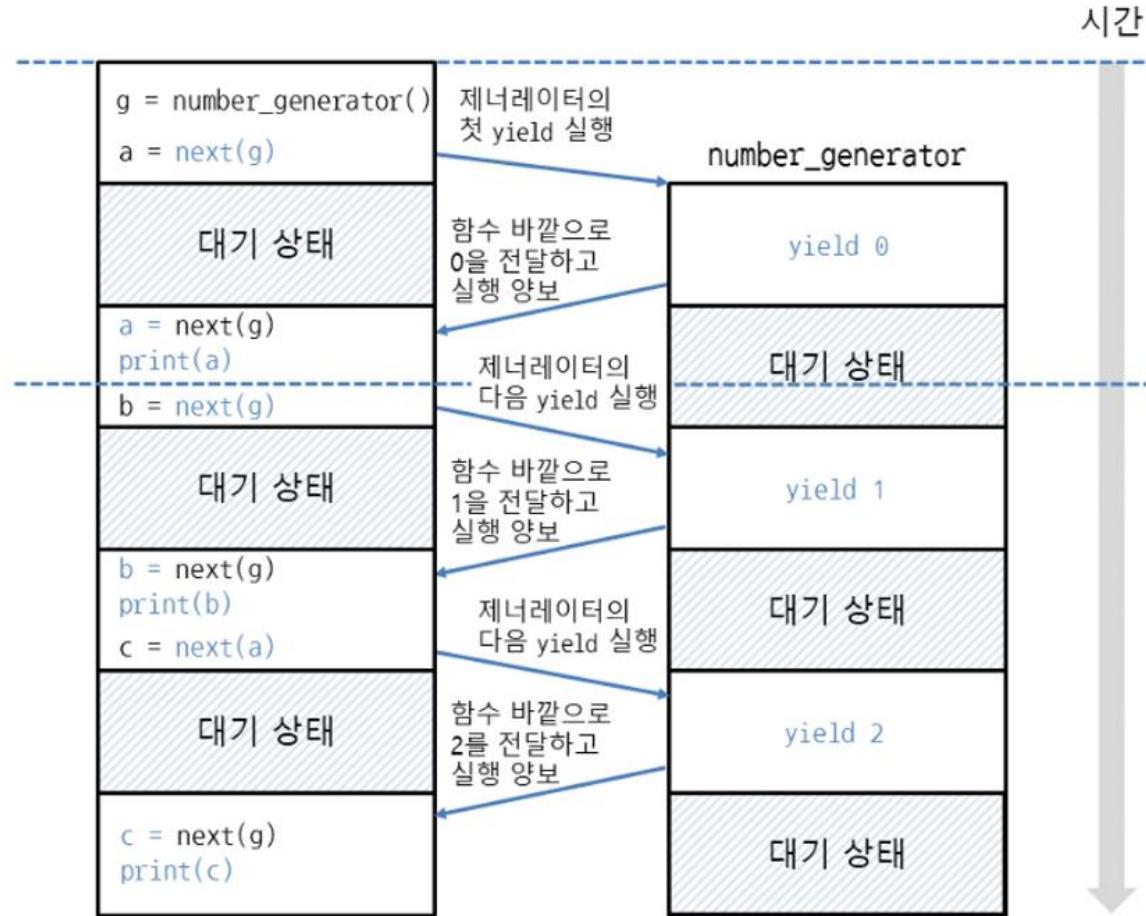
c = next(g)
print(c)        # 2
```

실행 결과

```
0
1
2
```

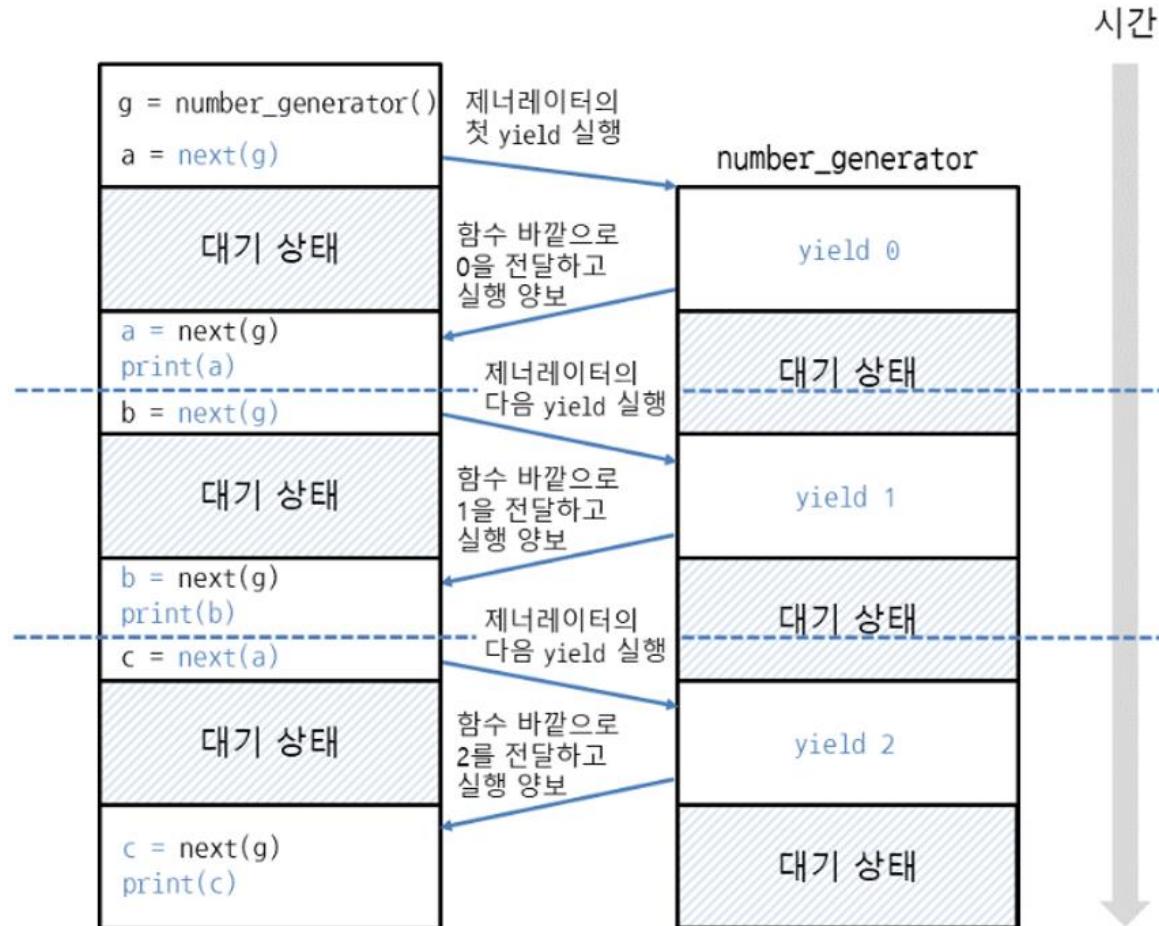
# 제너레이터(generator) 활용

▼ 그림 40-2 yield 0의 실행 양보



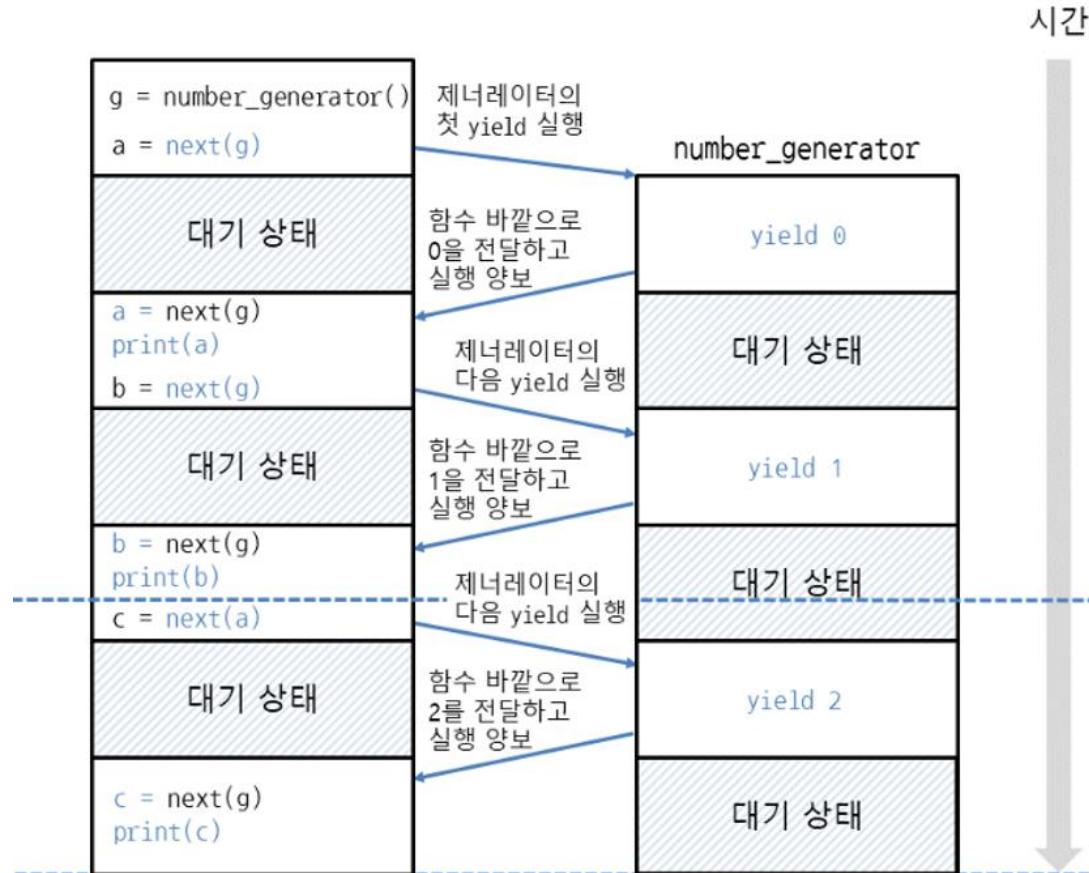
# 제너레이터(generator) 활용

▼ 그림 40-3 yield 1의 실행 양보



# 제너레이터(generator) 활용

▼ 그림 40-4 yield 2의 실행 양보



# 제너레이터(generator) 활용

## 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
```

# 제너레이터(generator) 활용

## 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
```

# 제너레이터(generator) 활용

## 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
```

# 제너레이터(generator) 활용

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
```

# 제너레이터(generator) 활용

yield from으로 값을 여러 번 바깥으로 전달하기

- 이런 경우에는 매번 반복문을 사용하지 않고, yield from을 사용하면 됨
  - `yield from` 반복가능한 객체
  - `yield from` 이터레이터
  - `yield from` 제너레이터 객체
- 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
```

# 제너레이터(generator) 활용

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
```

# 제너레이터(generator) 활용

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)

# 코루틴(coroutin) 활용

## 코루틴 사용하기

```
def add(a, b):
    c = a + b      # add 함수가 끝나면 변수와 계산식은 사라짐
    print(c)
    print('add 함수')

def calc():
    add(1, 2)      # add 함수가 끝나면 다시 calc 함수로 돌아옴
    print('calc 함수')

calc()
```

- calc가 메인 루틴(main routine)이면 add는 calc의 서브 루틴(sub routine)임

# 코루틴(coroutine) 활용

▼ 그림 41-1 메인 루틴과 서브 루틴의 동작 과정



# 코루틴(coroutin) 활용

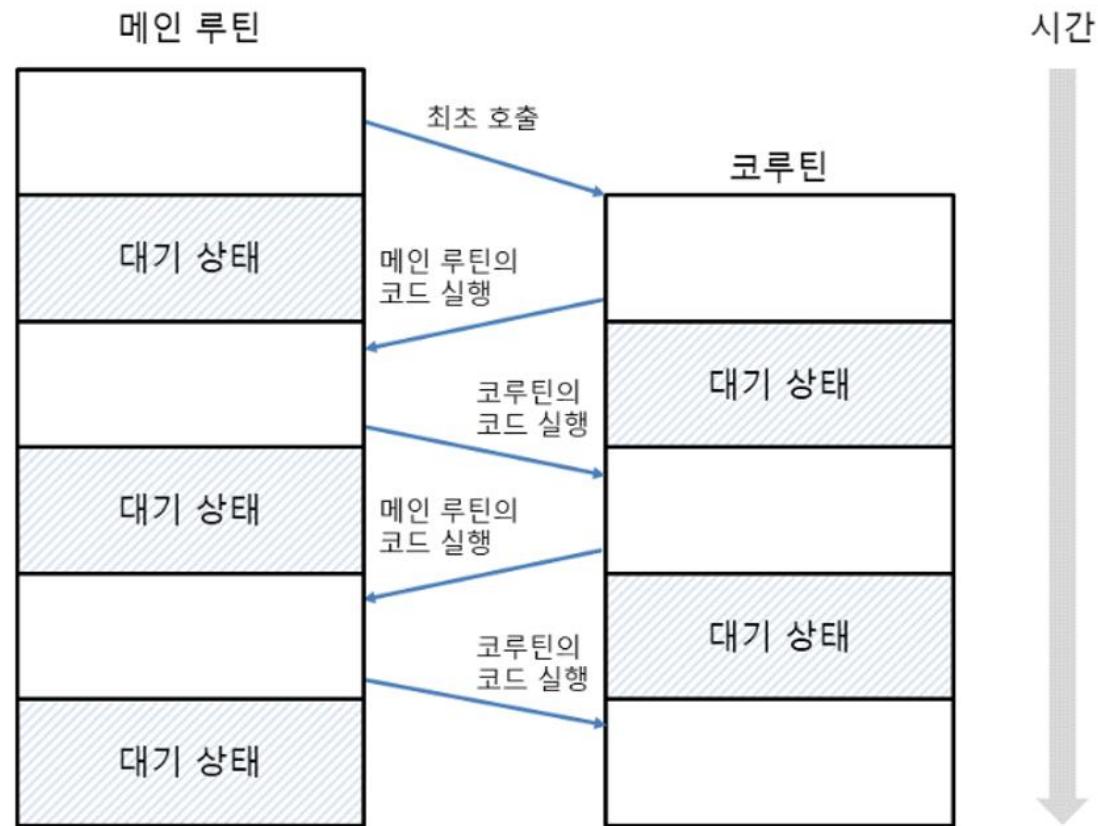
---

## 코루틴 사용하기

- 메인 루틴에서 서브 루틴을 호출하면 서브 루틴의 코드를 실행한 뒤 다시 메인 루틴으로 돌아옴
- 서브 루틴이 끝나면 서브 루틴의 내용은 모두 사라짐
- 서브 루틴은 메인 루틴에 종속된 관계임
- 코루틴은 방식이 조금 다른데 코루틴 coroutine은 cooperative routine를 의미하는데 서로 협력하는 루틴이라는 뜻임
- 메인 루틴과 서브 루틴처럼 종속된 관계가 아니라 서로 대등한 관계이며 특정 시점에 상대방의 코드를 실행함

# 코루틴(coroutine) 활용

▼ 그림 41-2 코루틴의 동작 과정



# 코루틴(coroutine) 활용

---

## 코루틴 사용하기

- 코루틴은 함수가 종료되지 않은 상태에서 메인 루틴의 코드를 실행한 뒤 다시 돌아와서 코루틴의 코드를 실행함
- 코루틴이 종료되지 않았으므로 코루틴의 내용도 계속 유지됨
- 일반 함수를 호출하면 코드를 한 번만 실행할 수 있지만, 코루틴은 코드를 여러 번 실행할 수 있음
- 참고로 함수의 코드를 실행하는 지점을 진입점(entry point)이라고 하는데, 코루틴은 진입점이 여러 개인 함수임

# 코루틴(coroutin) 활용

---

## 코루틴에 값 보내기

- 코루틴은 제너레이터의 특별한 형태임
- 제너레이터는 yield로 값을 발생시켰지만 코루틴은 yield로 값을 받아올 수 있음
- 다음과 같이 코루틴에 값을 보내면서 코드를 실행할 때는 send 메서드를 사용함
- send 메서드가 보낸 값을 받아오려면 (yield) 형식으로 yield를 괄호로 묶어준 뒤 변수에 저장함
  - 코루틴객체.send(값)
  - 변수 = (yield)

# 코루틴(coroutin) 활용

## 코루틴에 값 보내기

coroutine\_consumer.py

```
def number_coroutine():
    while True:          # 코루틴을 계속 유지하기 위해 무한 루프 사용
        x = (yield)      # 코루틴 바깥에서 값을 받아옴, yield를 괄호로 묶어야 함
        print(x)

co = number_coroutine()
next(co)      # 코루틴 안의 yield까지 코드 실행(최초 실행)

co.send(1)    # 코루틴에 숫자 1을 보냄
co.send(2)    # 코루틴에 숫자 2를 보냄
co.send(3)    # 코루틴에 숫자 3을 보냄
```

실행 결과

```
1
2
3
```

# 코루틴(coroutin) 활용

## 코루틴에 값 보내기

```
def number_coroutine():
    while True:      # 코루틴을 계속 유지하기 위해 무한 루프 사용
        x = (yield)  # 코루틴 바깥에서 값을 받아옴, yield를 괄호로 묶어야 함
        print(x)
```

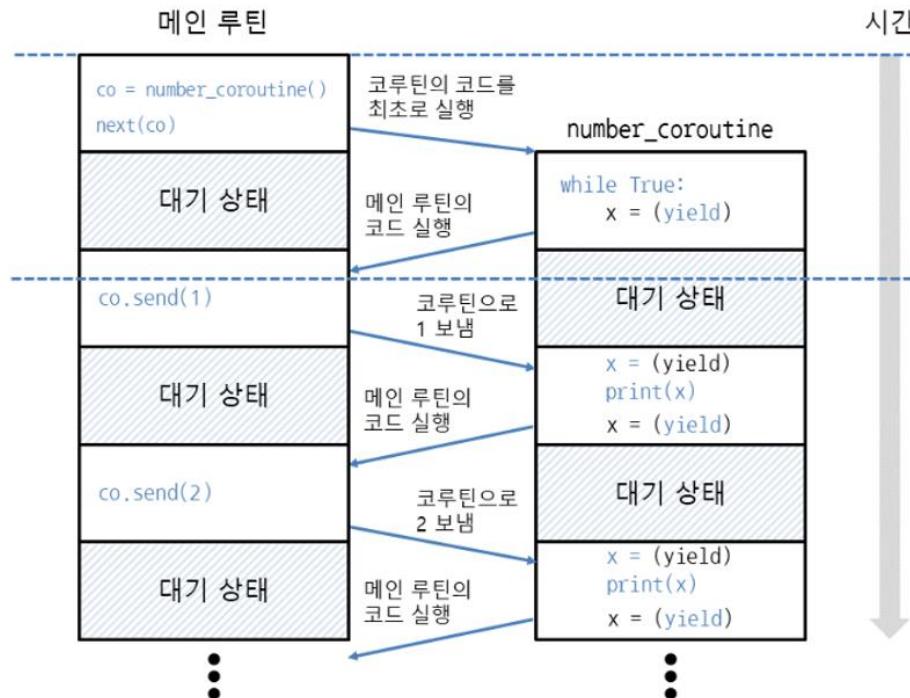
- next(코루틴객체)

```
co = number_coroutine()
next(co)      # 코루틴 안의 yield까지 코드 실행(최초 실행)
```

```
co.send(1)    # 코루틴에 숫자 1을 보냄
co.send(2)    # 코루틴에 숫자 2를 보냄
co.send(3)    # 코루틴에 숫자 3를 보냄
```

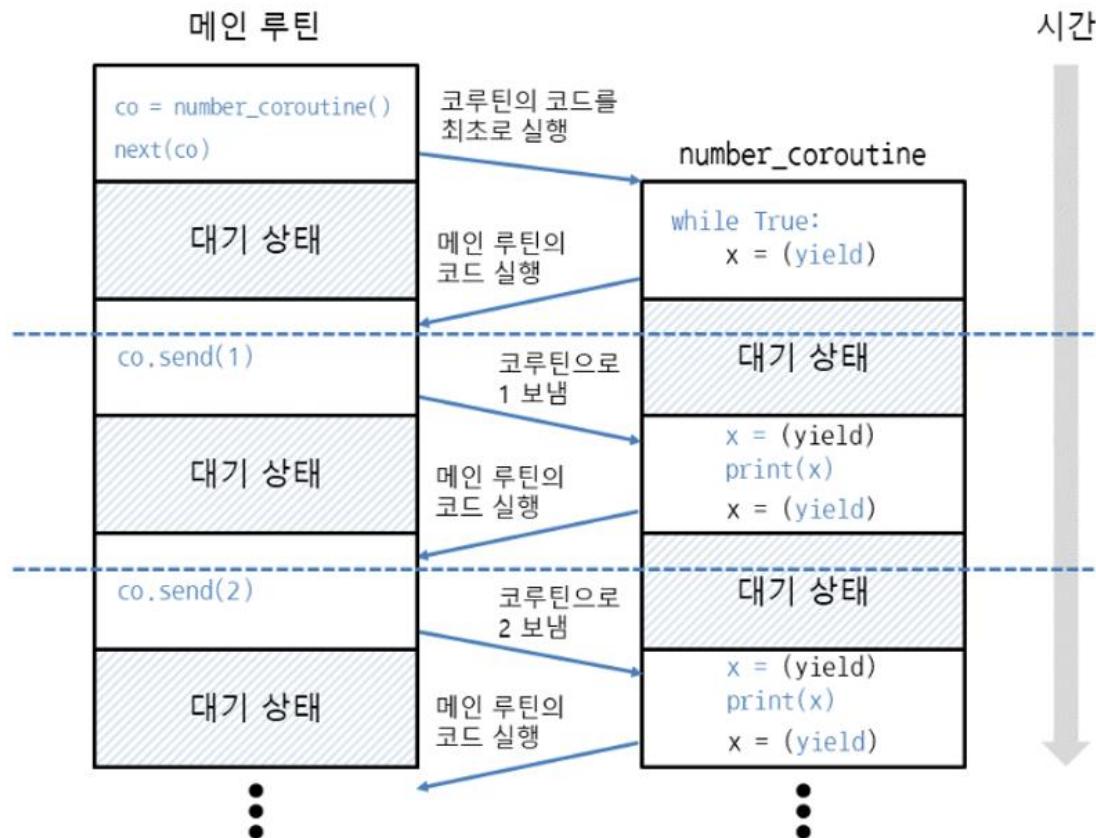
# 코루틴(coroutin) 활용

▼ 그림 41-3 코루틴의 코드를 최초로 실행한 뒤 메인 루틴으로 돌아옴



# 코루틴(coroutine) 활용

▼ 그림 41-4 send로 값을 보내면 코루틴에서 값을 받아서 출력



# 코루틴(coroutine) 활용

## 코루틴 바깥으로 값 전달하기

- 다음과 같이 (yield 변수) 형식으로 yield에 변수를 지정한 뒤 괄호로 묶어주면 값을 받아오면서 바깥으로 값을 전달함
- 변수 = (yield 변수)
- 변수 = next(코루틴객체)
- 변수 = 코루틴객체.send(값)

coroutine\_producer\_consumer.py

```
def sum_coroutine():
    total = 0
    while True:
        x = (yield total)    # 코루틴 바깥에서 값을 받아오면서 바깥으로 값을 전달
        total += x

co = sum_coroutine()
print(next(co))      # 0: 코루틴 안의 yield까지 코드를 실행하고 코루틴에서 나온 값 출력

print(co.send(1))    # 1: 코루틴에 숫자 1을 보내고 코루틴에서 나온 값 출력
print(co.send(2))    # 3: 코루틴에 숫자 2를 보내고 코루틴에서 나온 값 출력
print(co.send(3))    # 6: 코루틴에 숫자 3를 보내고 코루틴에서 나온 값 출력
```

실행 결과

```
0
1
3
6
```

# 코루틴(coroutin) 활용

## 코루틴 바깥으로 값 전달하기

- 그다음에 total += x와 같이 받은 값을 누적해줌

```
def sum_coroutine():
    total = 0
    while True:
        x = (yield total)      # 코루틴 바깥에서 값을 받아오면서 바깥으로 값을 전달
        total += n
```

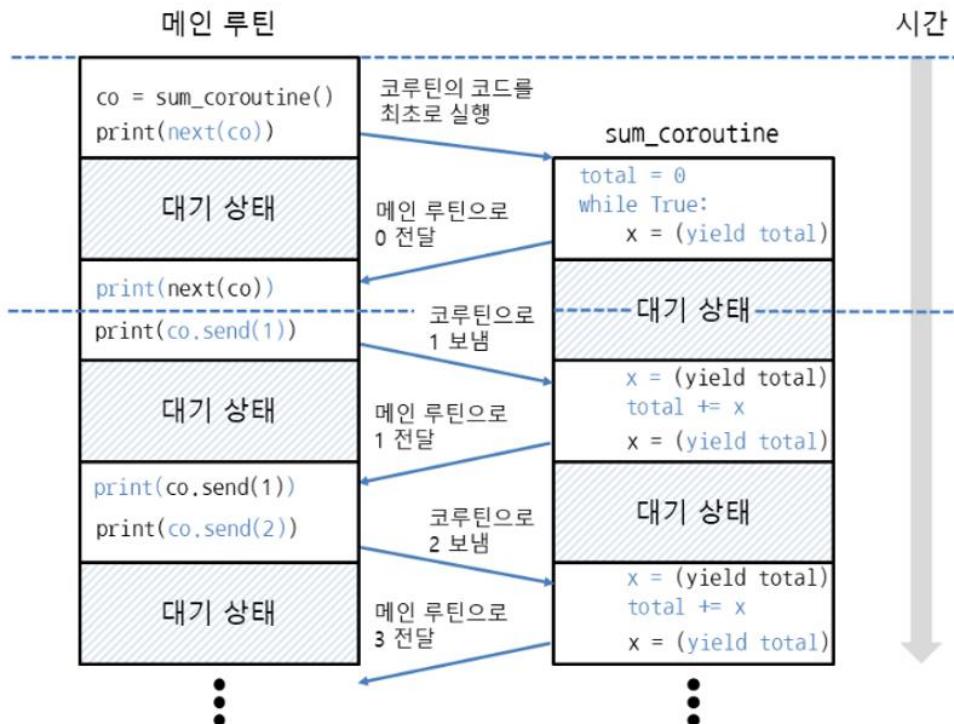
```
co = sum_coroutine()
print(next(co))      # 0: 코루틴 안의 yield까지 코드를 실행하고 코루틴에서 나온 값 출력

print(co.send(1))    # 1: 코루틴에 숫자 1을 보내고 코루틴에서 나온 값 출력
print(co.send(2))    # 3: 코루틴에 숫자 2를 보내고 코루틴에서 나온 값 출력
print(co.send(3))    # 6: 코루틴에 숫자 3을 보내고 코루틴에서 나온 값 출력
```

- 참고로 next와 send의 차이를 살펴보면 next는 코루틴의 코드를 실행하지만 값을 보내지 않을 때 사용하고, send는 값을 보내면서 코루틴의 코드를 실행할 때 사용함

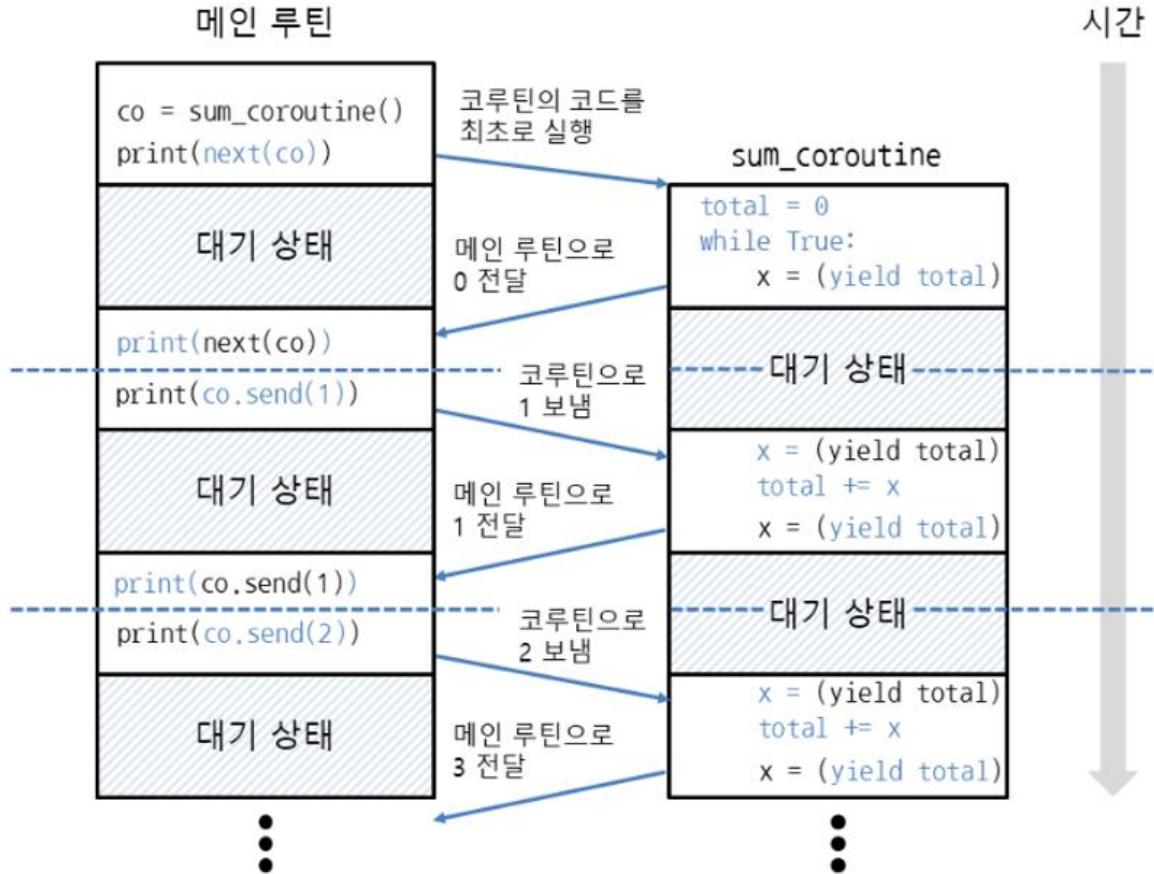
# 코루틴(coroutine) 활용

▼ 그림 41-5 코루틴의 코드를 최초로 실행한 뒤 yield의 값을 받아와서 출력



# 코루틴(coroutine) 활용

▼ 그림 41-6 코루틴에 값을 보낸 뒤 결과를 받아서 출력



# 코루틴(coroutin) 활용

---

## 코루틴 바깥으로 값 전달하기

- 제너레이터와 코루틴의 차이점을 정리해보자
  - 제너레이터는 `next` 함수(`__next__` 메서드)를 반복 호출하여 값을 얻어내는 방식
  - 코루틴은 `next` 함수(`__next__` 메서드)를 한 번만 호출한 뒤 `send`로 값을 주고 받는 방식

# 코루틴(coroutin) 활용

## 코루틴을 종료하고 예외 처리하기

- 코루틴은 실행 상태를 유지하기 위해 while True:를 사용해서 끝나지 않는 무한 루프로 동작함
- 코루틴을 강제로 종료하고 싶다면 close 메서드를 사용함
  - 코루틴객체.close()

coroutine\_close.py

```
def number_coroutine():
    while True:
        x = (yield)
        print(x, end=' ')

co = number_coroutine()
next(co)

for i in range(20):
    co.send(i)

co.close()    # 코루틴 종료
```

실행 결과

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

# 코루틴(coroutin) 활용

## GeneratorExit 예외 처리하기

- 코루틴 객체에서 close 메서드를 호출하면 코루틴이 종료될 때 GeneratorExit 예외가 발생함
- 이 예외를 처리하면 코루틴의 종료 시점을 알 수 있음

coroutine\_generator\_exit.py

```
def number_coroutine():
    try:
        while True:
            x = (yield)
            print(x, end=' ')
    except GeneratorExit:    # 코루틴이 종료 될 때 GeneratorExit 예외 발생
        print()
        print('코루틴 종료')

co = number_coroutine()
next(co)

for i in range(20):
    co.send(i)

co.close()
```

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19  
코루틴 종료

# 코루틴(coroutin) 활용

## GeneratorExit 예외 처리하기

- 코루틴 안에서 try except로 GeneratorExit 예외가 발생하면 '코루틴 종료'가 출력되도록 만들었음
- close 메서드로 코루틴을 종료할 때 원하는 코드를 실행할 수 있음

```
except GeneratorExit:    # 코루틴이 종료 될 때 GeneratorExit 예외 발생
    print()
    print('코루틴 종료')
```

# 코루틴(coroutin) 활용

---

## 코루틴 안에서 예외 발생시키기

- 코루틴 안에 예외를 발생 시킬 때는 throw 메서드를 사용함
- throw는 말그대로 던지다라는 뜻인데 예외를 코루틴 안으로 던짐
- throw 메서드에 지정한 에러 메시지는 except as의 변수에 들어감
- 코루틴객체.throw(예외이름, 에러메시지)

# 코루틴(coroutin) 활용

## 코루틴 안에서 예외 발생시키기

- 다음은 코루틴에 숫자를 보내서 누적하다가 RuntimeError 예외가 발생하면 에러 메시지를 출력하고 누적된 값을 코루틴 바깥으로 전달함

coroutine\_throw.py

```
def sum_coroutine():
    try:
        total = 0
        while True:
            x = (yield)
            total += x
    except RuntimeError as e:
        print(e)
        yield total    # 코루틴 바깥으로 값 전달

co = sum_coroutine()
next(co)

for i in range(20):
    co.send(i)

print(co.throw(RuntimeError, '예외로 코루틴 끝내기')) # 190
                                                # 코루틴의 except에서 yield로 전달받은 값
```

실행 결과

예외로 코루틴 끝내기

190

# 코루틴(coroutin) 활용

## 코루틴 안에서 예외 발생시키기

```
except RuntimeError as e:  
    print(e)  
    yield total    # 코루틴 바깥으로 값 전달
```

# 코루틴(coroutin) 활용

---

## 하위 코루틴의 반환값 가져오기

- yield from에 코루틴을 지정하면 해당 코루틴에서 return으로 반환한 값을 가져옴(yield from은 파이썬 3.3 이상부터 사용 가능)
  - 변수 = `yield from 코루틴()`

# 코루틴(coroutin) 활용

## 하위 코루틴의 반환값 가져오기

- 다음은 코루틴에서 숫자를 누적한 뒤 합계를 yield from으로 가져옴

coroutine\_yield\_from.py

```
def accumulate():
    total = 0
    while True:
        x = (yield)      # 코루틴 바깥에서 값을 받아옴
        if x is None:   # 받아온 값이 None이면
            return total # 함께 total을 반환
        total += x

def sum_coroutine():
    while True:
        total = yield from accumulate()    # accumulate의 반환값을 가져옴
        print(total)

co = sum_coroutine()
next(co)

for i in range(1, 11):    # 1부터 10)까지 반복
    co.send(i)            # 코루틴 accumulate에 숫자를 보냄
co.send(None)             # 코루틴 accumulate에 None을 보내서 숫자 누적을 끝냄

for i in range(1, 101):   # 1부터 100)까지 반복
    co.send(i)            # 코루틴 accumulate에 숫자를 보냄
co.send(None)             # 코루틴 accumulate에 None을 보내서 숫자 누적을 끝냄
```

실행 결과

55

5050

# 코루틴(coroutin) 활용

## 하위 코루틴의 반환값 가져오기

- 먼저 숫자를 받아서 누적할 코루틴을 만듬

```
def accumulate():
    total = 0
    while True:
        x = (yield)          # 코루틴 바깥에서 값을 받아옴
        if x is None:        # 받아온 값이 None이면
            return total    # 합계 total을 반환, 코루틴을 끝냄
        total += x
```

- 이제 합계를 출력할 코루틴을 만듬

```
def sum_coroutine():
    while True:
        total = yield from accumulate()    # accumulate의 반환값을 가져옴
        print(total)
```

# 코루틴(coroutin) 활용

## 하위 코루틴의 반환값 가져오기

- 코루틴에서 `yield from`을 사용하면 코루틴 바깥에서 `send`로 하위 코루틴까지 값을 보낼 수 있음

```
co = sum_coroutine()
next(co)

for i in range(1, 11):    # 1부터 10까지 반복
    co.send(i)            # 코루틴 accumulate에 숫자를 보냄
```

```
co.send(None)             # 코루틴 accumulate에 None을 보내서 숫자 누적을 끝냄
```

```
def sum_coroutine():
    while True:
        total = yield from accumulate()    # accumulate가 끝나면 yield from으로 다시 실행
        print(total)
```

# 코루틴(coroutin) 활용

## StopIteration 예외 발생시키기(파이썬 3.6)

- 코루틴도 제너레이터이므로 return을 사용하면 StopIteration이 발생함
- 코루틴에서 return 값은 raise StopIteration(값)과 동작이 같음
- raise로 예외를 직접 발생시키고 값을 지정하면 yield from으로 값을 가져올 수 있음
- `raise StopIteration(값)`

## 파이썬 3.7 변경 사항

- 제너레이터에서 사용되는 StopIteration을 Runtime Error로 변경했기 때문에 raise StopIteration을 사용할 수 없습니다.
- raise StopIteration 대신 return 사용으로 수정

# 코루틴(coroutin) 활용

## StopIteration 예외 발생시키기

coroutine\_stopiteration.py

```
def accumulate():
    total = 0
    while True:
        x = (yield)
                    # 코루틴 바깥에서 값을 받아옴
        if x is None:
                    # 받아온 값이 None이면
            raise StopIteration(total)      # StopIteration에 반환할 값을 지정
        total += x
```

파이썬 3.7에서는 다음과 같이 수정  
return total

```
def sum_coroutine():
    while True:
        total = yield from accumulate()    # accumulate의 반환값을 가져옴
        print(total)
```

```
co = sum_coroutine()
next(co)

for i in range(1, 11):    # 1부터 10)까지 반복
    co.send(i)           # 코루틴 accumulate에 숫자를 보냄
co.send(None)             # 코루틴 accumulate에 None을 보내서 숫자 누적을 끝냄

for i in range(1, 101):   # 1부터 100)까지 반복
    co.send(i)           # 코루틴 accumulate에 숫자를 보냄
co.send(None)             # 코루틴 accumulate에 None을 보내서 숫자 누적을 끝냄
```

실행 결과

```
55
5050
```