

# 교수의 파이썬

04\_3 제너레이터와 yield

창원대학교 정보통신공학과 교수 박동규

# 넌넌한 교수의 파이썬

04\_3 제너레이터와 yield

창원대학교 정보통신공학과 교수 박동규

# 넌넌한 교수의 고급 파이썬

04\_3 제너레이터와 yield

창원대학교 정보통신공학과 교수 박동규

# 넌넌한 교수의 고급 파이썬

04\_3 제너레이터와 yield

창원대학교 정보통신공학과 교수 박동규

# 제너레이터generator

# 제너레이터generator

- 제너레이터 객체는 모든 값을 메모리에 올려두고 이용하는 것이 아니라 필요할 때마다 생성해서 반환하는 일을 한다.
- 이 때문에 메모리를 효율적으로 사용할 수 있다는 장점이 있다.

제너레이터와 이를 이용한 for 문

# 제너레이터와 이를 이용한 for 문

```
>>> my_generator = (x for x in range(1, 4))
>>> for n in my_generator :
...     print(n)
...
1
2
3
>>> type(my_generator)
<class 'generator'>
```



# 제너레이터와 이를 이용한 for 문

```
>>> my_generator = (x for x in range(1, 4))
>>> for n in my_generator :
...     print(n)
...
1
2
3
>>> type(my_generator)
<class 'generator'>
```

반복자와 동일한 일을 하는 것처럼 보이지만  
여기에서 생성된 1, 2, 3을 미리 메모리에 만들어 두는 것이 아니라  
for 문에서 필요로 할 때마다 my\_generator로 부터 받아오며  
**메모리에서 보관하지 않는다**는 점이다.(이를 lazy evaluation이라 함)

yield

# yield

일반적인 함수의 return 문과 유사하다.

하지만 yield문은 제너레이터를 반환한다는 점에서 return문과 차이가 있다.

# yield

일반적인 함수의 return 문과 유사하다.

하지만 yield문은 제너레이터를 반환한다는 점에서 return문과 차이가 있다.

```
1 def create_gen():
2     alist = range(1, 4)
3     for x in alist:
4         yield x
5
6 my_generator = create_gen()
7 print(my_generator)
8
9 for n in my_generator:
10     print(n)
```

# yield

일반적인 함수의 return 문과 유사하다.

하지만 yield문은 제너레이터를 반환한다는 점에서 return문과 차이가 있다.


```
1 def create_gen():  
2     alist = range(1, 4)  
3     for x in alist:  
4         yield x  
5  
6 my_generator = create_gen()  
7 print(my_generator)  
8  
9 for n in my_generator:  
10     print(n)
```

# yield

일반적인 함수의 return 문과 유사하다.

하지만 yield문은 제너레이터를 반환한다는 점에서 return문과 차이가 있다.

```
1 def create_gen():  
2     alist = range(1, 4)  
3     for x in alist:  
4         yield x  
5  
6 my_generator = create_gen()  
7 print(my_generator)  
8  
9 for n in my_generator:  
10    print(n)
```



# yield

일반적인 함수의 return 문과 유사하다.

하지만 yield문은 제너레이터를 반환한다는 점에서 return문과 차이가 있다.

```
1 def create_gen():  
2     alist = range(1, 4)  
3     for x in alist:  
4         yield x  
5  
6 my_generator = create_gen()  
7 print(my_generator)  
8  
9 for n in my_generator:  
10    print(n)
```

return 문 대신  
yield 문을 가지고 있는  
제너레이터

# yield

일반적인 함수의 return 문과 유사하다.

하지만 yield문은 제너레이터를 반환한다는 점에서 return문과 차이가 있다.

```
1 def create_gen():  
2     alist = range(1, 4)  
3     for x in alist:  
4         yield x  
5  
6 my_generator = create_gen()  
7 print(my_generator)  
8  
9 for n in my_generator:  
10    print(n)
```

return 문 대신  
yield 문을 가지고 있는  
제너레이터



# yield

일반적인 함수의 return 문과 유사하다.

하지만 yield문은 제너레이터를 반환한다는 점에서 return문과 차이가 있다.

```
1 def create_gen():  
2     alist = range(1, 4)  
3     for x in alist:  
4         yield x  
5  
6 my_generator = create_gen()  
7 print(my_generator)  
8  
9 for n in my_generator:  
10     print(n)
```

return 문 대신  
yield 문을 가지고 있는  
제너레이터

실행 결과

```
<generator object create_gen at 0x7f85955d77d8>  
1  
2  
3
```

# return vs yield

## Return

The return statement is where all the local variables are destroyed and the resulting value is given back (returned) to the caller. Should the same function be called some time later, the function will get a fresh new set of variables.

## Yield

But what if the local variables aren't thrown away when we exit a function? This implies that we can `resume the function` where we left off. This is where the concept of `generators` are introduced and the `yield` statement resumes where the `function` left off.

# 제너레이터를 사용할 때 주의사항

# 제너레이터를 사용할 때 주의사항

제너레이터는 실행될 때 함수의 몸체를 실행하는 것이 아니라,  
제너레이터 함수가 가진 객체를 반환하는 일을 한다.

# 제너레이터를 사용할 때 주의사항

제너레이터는 실행될 때 함수의 몸체를 실행하는 것이 아니라,  
제너레이터 함수가 가진 객체를 반환하는 일을 한다.

제너레이터는 한번 생성해서 반환한 객체를 보관하지 않기 때문에  
이전의 코드를 실행한 후 추가한 코드를 실행하면 아무런 객체도 출력되지 않는다.

# 제너레이터를 사용할 때 주의사항

제너레이터는 실행될 때 함수의 몸체를 실행하는 것이 아니라,  
제너레이터 함수가 가진 객체를 반환하는 일을 한다.

제너레이터는 한번 생성해서 반환한 객체를 보관하지 않기 때문에  
이전의 코드를 실행한 후 추가한 코드를 실행하면 아무런 객체도 출력되지 않는다.

이전의 코드

```
1 def create_gen():
2     alist = range(1, 4)
3     for x in alist:
4         yield x
5
6 my_generator = create_gen()
7 print(my_generator)
8 for n in my_generator:
9     print(n)
```

# 제너레이터를 사용할 때 주의사항

제너레이터는 실행될 때 함수의 몸체를 실행하는 것이 아니라,  
제너레이터 함수가 가진 객체를 반환하는 일을 한다.

제너레이터는 한번 생성해서 반환한 객체를 보관하지 않기 때문에  
이전의 코드를 실행한 후 추가한 코드를 실행하면 아무런 객체도 출력되지 않는다.

이전의 코드

```
1 def create_gen():  
2     alist = range(1, 4)  
3     for x in alist:  
4         yield x  
5  
6 my_generator = create_gen()  
7 print(my_generator)  
8 for n in my_generator:  
9     print(n)
```

추가한 코드

```
1 for n in my_generator:  
2     print(n)
```

# 제너레이터를 사용할 때 주의사항

제너레이터는 실행될 때 함수의 몸체를 실행하는 것이 아니라,  
제너레이터 함수가 가진 객체를 반환하는 일을 한다.

제너레이터는 한번 생성해서 반환한 객체를 보관하지 않기 때문에  
이전의 코드를 실행한 후 추가한 코드를 실행하면 아무런 객체도 출력되지 않는다.

이전의 코드

```
1 def create_gen():
2     alist = range(1, 4)
3     for x in alist:
4         yield x
5
6 my_generator = create_gen()
7 print(my_generator)
8 for n in my_generator:
9     print(n)
```

추가한 코드

```
1 for n in my_generator:
2     print(n)
```

실행 결과 :

제너레이터는 한번 실행하면  
아무것도 반환하지 않음



# 왜 제너레이터를 사용하나

출 처 : <https://stackoverflow.com/questions/102535/what-can-you-use-python-generator-functions-for>

# 왜 제너레이터를 사용하나

- 제너레이터는 메모리를 절약해 준다.
- 제너레이터는 수행시간도 절약해 준다.
- 100만개의 피보나치 수열을 출력하고 싶은 경우
  1. 100만개의 피보나치 수열을 생성해서 리스트에 넣고 for 문을 통해 출력한다(일반 함수를 사용해 보자)
  2. 100만개의 수열을 계산할 때마다 for 문에 넘겨준다(제너레이터를 사용해 보자)

출 처 : <https://stackoverflow.com/questions/102535/what-can-you-use-python-generator-functions-for>

일반 함수를 사용한 루프

제너레이터를 사용한 루프

## 일반 함수를 사용한 루프

```
import time

# function version
def fibon(n):
    a = b = 1
    result = []
    for i in range(n):
        result.append(a)
        a, b = b, a + b
    return result

start_t = time.time()
for x in fibon(1000000):
    pass

end_t = time.time()
print('total time = ', end_t - start_t)
```

## 제너레이터를 사용한 루프

## 일반 함수를 사용한 루프

```
import time

# function version
def fibon(n):
    a = b = 1
    result = []
    for i in range(n):
        result.append(a)
        a, b = b, a + b
    return result

start_t = time.time()
for x in fibon(1000000):
    pass

end_t = time.time()
print('total time = ', end_t - start_t)
```

```
RESTART: /Users/dongupak/Documen
fibon_func.py
total time = 148.18809485435486
>>>
```

## 제너레이터를 사용한 루프

## 일반 함수를 사용한 루프

```
import time

# function version
def fibon(n):
    a = b = 1
    result = []
    for i in range(n):
        result.append(a)
        a, b = b, a + b
    return result

start_t = time.time()
for x in fibon(1000000):
    pass

end_t = time.time()
print('total time = ', end_t - start_t)
```

```
RESTART: /Users/dongupak/Documen
fibon_func.py
total time = 148.18809485435486
>>>
```

## 제너레이터를 사용한 루프

```
import time

# generator version
def fibon(n):
    a = b = 1
    for i in range(n):
        yield a
        a, b = b, a + b

start_t = time.time()
for x in fibon(1000000):
    pass

end_t = time.time()
print('total time = ', end_t - start_t)
```

## 일반 함수를 사용한 루프

```
import time

# function version
def fibon(n):
    a = b = 1
    result = []
    for i in range(n):
        result.append(a)
        a, b = b, a + b
    return result

start_t = time.time()
for x in fibon(1000000):
    pass

end_t = time.time()
print('total time = ', end_t - start_t)
```

```
RESTART: /Users/dongupak/Documen
fibon_func.py
total time = 148.18809485435486
>>>
```

## 제너레이터를 사용한 루프

```
import time

# generator version
def fibon(n):
    a = b = 1
    for i in range(n):
        yield a
        a, b = b, a + b

start_t = time.time()
for x in fibon(1000000):
    pass

end_t = time.time()
print('total time = ', end_t - start_t)
```

```
RESTART: /Users/dongupak/Docume
fibon_generator.py
total time = 9.437483072280884
>>>
```





감사합니다.