

Ask Django

코루틴 및 제너레이터

Python2 에서의 range & xrange

- Python3 에서는 range 가 제거되고, xrange 가 range 로 변경됨
- 차이
 - range : 가용값들을 미리 생성하고 시작하느냐 (+ 메모리 공간)
 - xrange : 값의 범위만 정해두고, 값을 그때 그때 생성해내느냐 (+ CPU 연산)

Python2 에서의 range

한 리스트에 **300,000,000**개를 구성하고 나서, 첫 값을 출력하고, **break**

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals
import time
begin_t = time.time()

for i in range(300000000):
    print(i)
    break

print('{:.1f}초 소요'.format(time.time() - begin_t))
```

실행결과

0
17.8초 소요

Python2 에서의 xrange

첫 값을 생산하자마자, 출력하고, **break**

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals
import time
begin_t = time.time()

for i in xrange(300000000):
    print(i)
    break

print('{:.1f}초 소요'.format(time.time() - begin_t))
```

실행결과

0
0.1초 소요

range, xrange 개념 구현

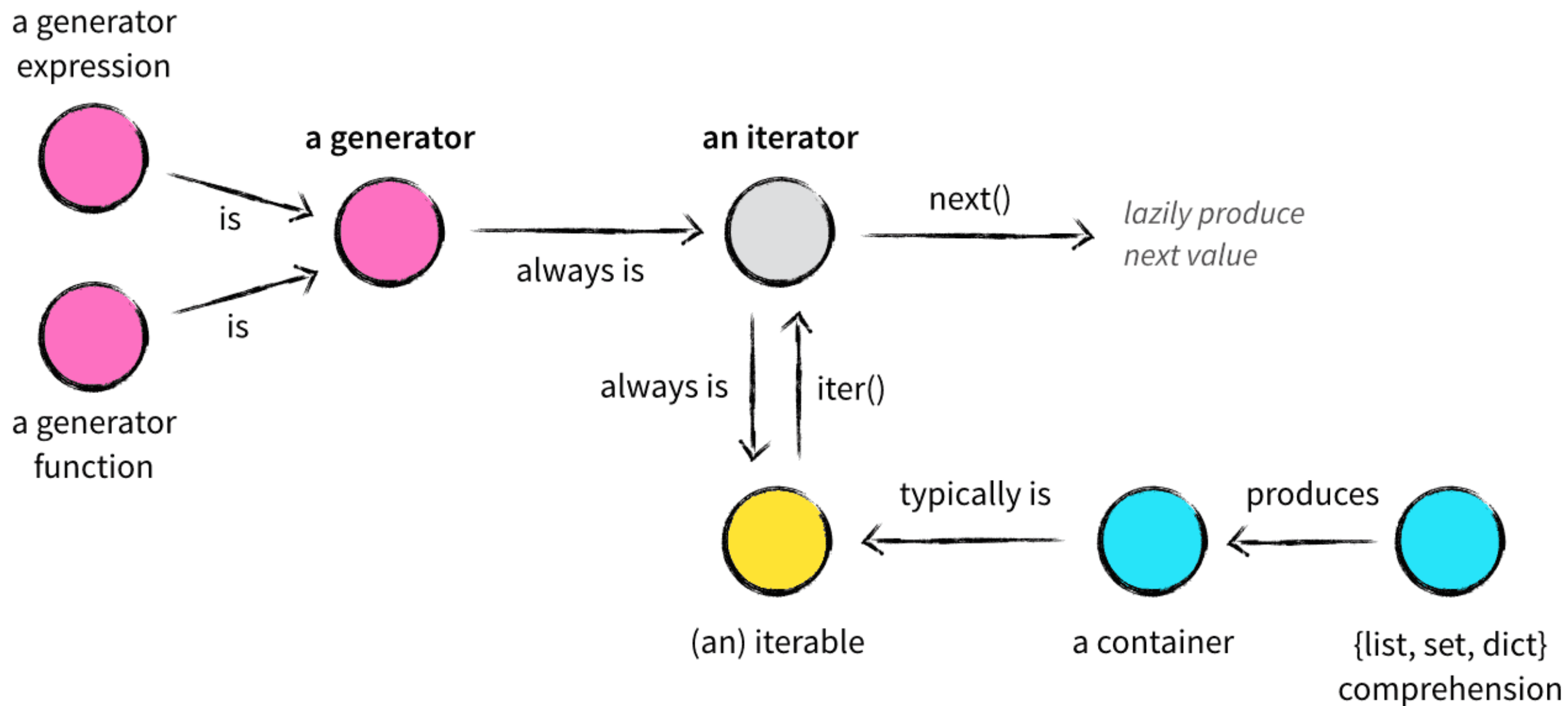
```
def myrange(start, end, step):  
    mylist = []  
    while start < end:  
        mylist.append(start)  
        start += step  
    return mylist  
  
def myxrange(start, end, step):      # 코루틴 생성  
    while start < end:  
        yield start      # generator 문법  
        start += step
```

구현/동작의 차이도 있지만, 동일하게 사용 가능

```
if __name__ == '__main__':  
    for i in myrange(0, 10, 2):  
        print(i)  
  
    for i in myxrange(0, 10, 2):  
        print(i)
```

Iterables vs. Iterators vs. Generators #doc #번역

Iterable, Callable EP에서 소개했던 내용



Co-Routine (코루틴)

- Sub Routine : 일반적인 함수
 - 진입점이 하나이며, 부모/자식의 종속적인 관계가 성립
 - 매 호출시마다, Routine 내 context 가 초기화
- Co Routine : 코루틴
 - 진입점이 여럿이며, 병렬(Concurrency, not Parallelism) 수행
 - 호출부와 대등한 관계
 - 여러번 호출이 되어도, Routine 내 Context가 유지

Generator 문법으로 간편히 코루틴 구현

```
def sub_routine():  
    return 10
```

```
def co_routine():  
    yield 10  
    yield 20  
    yield 30
```

```
>>> sub_routine()  
10
```

```
>>> generator1 = co_routine()  
>>> generator1  
<generator object co_routine at 0x1062a3fc0>
```

```
>>> next(generator1) # 이제서야 루틴이 실행 => 첫 yield까지 수행  
10
```

```
>>> next(generator1) # 이어서, 다음 yield까지 수행  
20
```

```
>>> next(generator1) # 이어서, 다음 yield까지 수행  
30
```

```
>>> next(generator1) # 더 이상 생산(yield)할 수 없으므로, StopIteration 예외 자동 발생  
StopIteration:
```

코드 예시

```
def mysum(x, y):
    x += 1
    y += 1
    return x + y

def to_3(base):
    i = 0
    yield base
    i += 1
    yield base + 1
    i += 1
    yield base + 2
    i += 1
    yield base + 3
    yield i

def main():
    a = 1
    b = 2

    generator_obj1 = to_3(10)
    generator_obj2 = to_3(20)
    generator_obj3 = to_3(30)

    c = a + b

    print(mysum(a, b))

    print(next(generator_obj1)) # 10
    print(next(generator_obj1)) # 11
    print(next(generator_obj2)) # 20
    print(next(generator_obj2)) # 21
    print(next(generator_obj3)) # 30
    print(c) # 3

    print(mysum(10, 20))

    main()
```

main 함수

```
a = 1
b = 2

generator_obj1 = to_3(10)
generator_obj2 = to_3(20)
generator_obj3 = to_3(30)

c = a + b

print(next(generator_obj1)) # 10
print(next(generator_obj1)) # 11

print(next(generator_obj2)) # 20
print(next(generator_obj2)) # 21

print(next(generator_obj3)) # 30

print(c) # 3
```

generator_obj1

```
i = 0
yield base
i += 1
yield base + 1
i += 1
yield base + 2
i += 1
yield base + 3
yield i
```

generator_obj2

```
i = 0
yield base
i += 1
yield base + 1
i += 1
yield base + 2
i += 1
yield base + 3
yield i
```

generator_obj3

```
i = 0
yield base
i += 1
yield base + 1
i += 1
yield base + 2
i += 1
yield base + 3
yield i
```

Generator #doc

generator는 항상 **iterator**입니다.

- 연속된 (Sequence) 값들을 생산해내는 함수
- 함수에 `yield` 키워드가 쓰여지면, Generator
- `yield` 한 값들이 순차적으로 생산됩니다.
- Generator에서 `return`문을 만나더라도 종료만 될 뿐, 리턴값이 사용되지는 않습니다.

- 추가 yield 가 없으면, StopIteration 예외 자동 발생 (#ref)
 - for 루프는 StopIteration 예외를 자동으로 처리
 - 직접 StopIteration 예외를 발생시켜도 됩니다.

```
def myxrange(start, end):  
    while start < end:  
        yield start          # 함수 generator 문법  
        start += 1
```

중첩된 Generator는 Pipeline

- 매 Generator가 완료된 후에, 다음 Generator가 수행되는 것이 아니라,
- 한 Generator에서 값이 생산될 때마다 다음 Generator로 값이 전달

```
>>> gen1 = (i**2 for i in range(10)) # gen1에서 0이 생산되자마자
>>> gen2 = (j+10 for j in gen1)      # gen2로 전달 ... (쭈욱~)
>>> gen3 = (k*10 for k in gen2)      # gen3로 전달 ...

>>> for i in gen3: # 첫 번째 값을 받아올 때, 그제서야 gen1에서 값 생산 시작
    print(i, end=' ')
```

100 110 140 190 260 350 460 590 740 910

- 데이터가 아무리 많아도, 첫 스타트업 시작이 짧습니다.
- 그렇기에, 가급적이면 Generator를 그대로 써주세요. list/tuple로 변환하지마세요.

```
>>> t1 = tuple(i**2 for i in range(10)) # 모든 값을 tuple로 생성
>>> t2 = tuple(j+10 for j in t1)        # (이하 동문)
>>> t3 = tuple(k*10 for k in t2)        # (이하 동문)

>>> for i in t3: # 이미 생성된 tuple값에 대해서 순회
    print(i, end=' ')
```

100 110 140 190 260 350 460 590 740 910

iterator로 tuple/list 생성하기

Generator is iterator.

```
def to_3():  
    yield 1  
    yield 2  
    yield 3
```

```
numbers_list = list(to_3())  
numbers_tuple = tuple(to_3())
```

tuple/list/iterator를 dict으로 변환

dict는 key/value 쌍으로 구성

```
mylist = [['a', 1], ['b', 2]]  
dict(mylist)  # {'a': 1, 'b': 2}
```

key가 중복이 되면, 마지막 key/value가 남습니다.

```
mylist = [['a', 1], ['b', 2], ['a', 3]]  
dict(mylist)  # {'a': 3, 'b': 2}
```

key/value 쌍이 맞지 않을 경우, ValueError 발생

```
mylist = [['a', 1], ['b', 2], ['a']]  
dict(mylist)  # ValueError: dictionary update sequence element #2 has length 1; 2 is required
```


Generator로 피보나치 수열 생산하기

제한된 크기만큼만 생산

```
def fib(max_count):  
    x, y, count = 1, 1, 0  
    while True:  
        if count >= max_count:  
            break  
        yield x  
        x, y = y, x + y  
        count += 1
```

```
>>> for x in fib(10):  
    print(x, end=' ')
```

1 1 2 3 5 8 13 21 34 55

소비하는 만큼만 생산 (좀 더 범용적으로 활용이 가능)

```
def fib():
    x, y = 1, 1
    while True:
        yield x
        x, y = y, x + y
```

```
>>> count = 0
      for x in fib():
          print(x, end= ' ')
          count += 1
          if count >= 10:
              break
```

1 1 2 3 5 8 13 21 34 55

```
>>> from itertools import islice
```

```
>>> islice(fib(), 10)
<itertools.islice at 0x106864958>
```

```
>>> tuple(islice(fib(), 10))
(1, 1, 2, 3, 5, 8, 13, 21, 34, 55)
```

list/set/dict Comprehension

- 순회가능한 객체를 조작하여, 필터링/새로운 리스트/사전/집합을 만들 수 있는 아주 간편한 방법
- tuple comprehension은 없어요. 필요하다면 tuple(순회가능한객체)

```
# list comprehension
[ 표현식 for 변수 in 순회가능한객체 ]
[ 표현식 for 변수 in 순회가능한객체 if 필터링조건 ]

[i**2 for i in range(5)]           # [0, 1, 4, 9, 16]
[i**2 for i in range(5) if i%2 == 0] # [0, 4, 16]

# dict comprehension
{ Key:표현식 for 변수 in 순회가능한객체 }
{ Key:표현식 for 변수 in 순회가능한객체 if 필터링조건 }

{i:i**2 for i in range(5)}         # {0:0, 1:1, 2:4, 3:9, 4:16}
{i:i**2 for i in range(5) if i%2 == 0} # {0:0, 2:4, 4:16}

# set comprehension
{ 표현식 for 변수 in 순회가능한객체 }
{ 표현식 for 변수 in 순회가능한객체 if 필터링조건 }

{i%5 for i in range(20)}           # {0, 1, 2, 3, 4}
{i%5 for i in range(20) if i%2==0} # {0, 1, 2, 3, 4}
```

Generator Expression (제너레이터 표현식)

- 문법비교) list comprehension : 한 번에 list를 생성

```
>>> [i**2 for i in range(100)]  
[0, 1, 4, 9, 16, 25, 36, 49, 64, ...]
```


- generator expression : 값을 그때그때 생성하여 **yield**

```
>>> (i**2 for i in range(100))  
<generator object <genexpr> at 0x10654ec50>
```

- 제너레이터 표현식으로 list/tuple 만들기

```
>>> list(i**2 for i in range(100))  
[0, 1, 4, 9, 16, 25, 36, 49, 64, ...]
```

```
>>> tuple(i**2 for i in range(100))  
(0, 1, 4, 9, 16, 25, 36, 49, 64, ...)
```

A person is seen from the side, sitting and working on a laptop. The background is a scenic view of a lake and snow-capped mountains under a clear blue sky. The text "Life is short, use Python3/Django." is overlaid in a large, elegant script font.

*Life is short,
use Python3/Django.*