파이썬 스터디

3주차

Ch10. range보다는 enumerate를 사용하자

루프의 사용

- range는 순회하는 루프를 실행할 때 자주 사용됨

```
for i in range(64):
    """sth"""
```

- 직접 루프를 실행할 수 있음

```
colors = ['red','green','blue']
for color in colors:
    """sth"""
```

range에 대해 알아보자

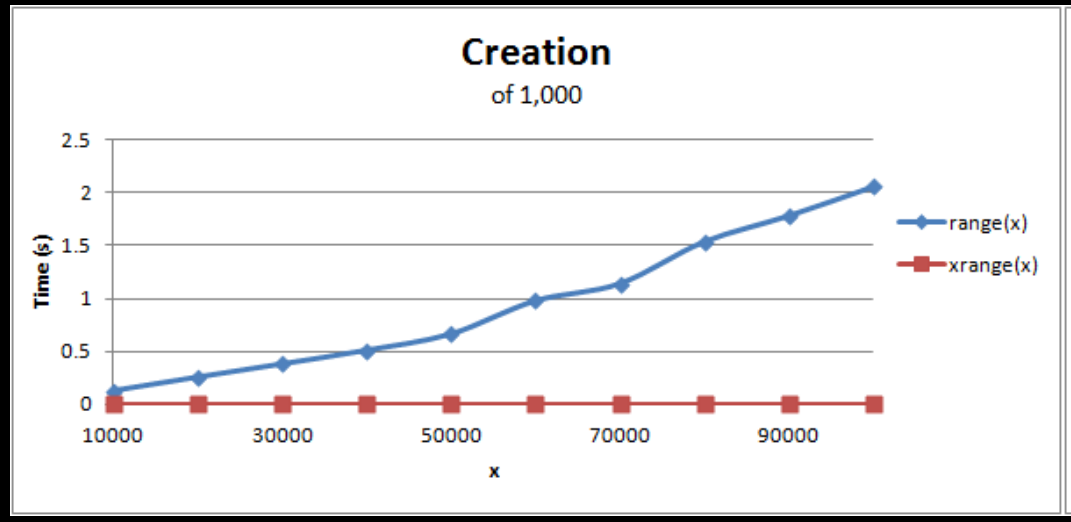
range([start], stop[, step])

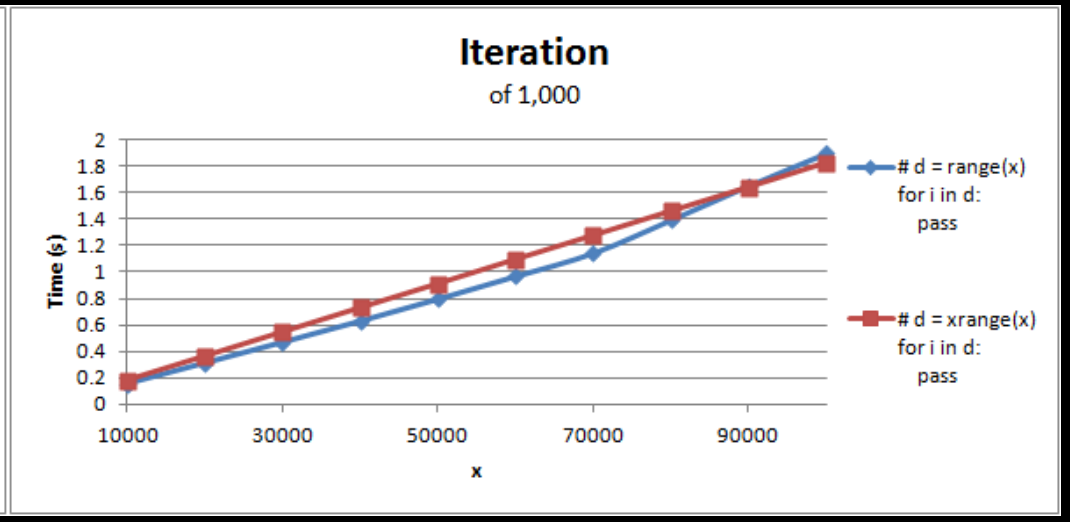
- start: Starting number of the sequence.
- stop: Generate numbers up to, but not including this number.
- step: Difference between each number in the sequence.

Python 2.x의 3.x의 range는 다르다

- 2.x range의 type은 list -> list 객체 생성하게 됨
- 3.x range의 type은 range -> 제너레이터

- 순회할 목적의 list 객체 리턴은 메모리의 낭비, 속도 저하를 초래





Python 2.x의 range

- Python 2.x 에서 range와 xrange를 나누어 놓았음
  - range -> list 객체 리턴
  - xrange -> 제너레이터

- 사용하는 목적이 다름
  - 리스트 연산이 필요할 때 range

```
r = range(5)[2:-1]
r = range(5) + [5,6,7]
```

Python 2.x의 range

- 3.x 버전에서 2.x range는 사라짐

- 2.x xrange -> 3.x range 가 됨

- List가 필요한 경우 list로 변환하면 됨

```
intList = list(range(5))
```

다시 본론으로 돌아와서

- List를 순회하는데, 인덱스를 알고 싶으면

```
colors = ['red','green','blue']

for i in range(len(colors)):
    color = colors[i]
    print('%d : %s' % (i, color))
>>>
0 : red
1 : green
2 : blue
```

enumerate를 사용하자

enumerate(sequence, start=0)

```
colors = ['red','green','blue']

for idx, color in enumerate(colors):
    print('%d : %s' % (idx, color))
>>>
0 : red
1 : green
2 : blue
```

enumerate를 사용하자

enumerate(sequence, start=0)

```
colors = ['red','green','blue']

for idx, color in enumerate(colors, 1):
    print('%d : %s' % (idx, color))
>>>
1 : red
2 : green
3 : blue
```

Ch11. 이터레이터를 병렬로 처리하려면 zip을 사용하자

zip

- Python 내장함수 zip(*iterables)
- 입력받은 각 이터레이터로부터 다음 값을 담은 tuple을 얻어옴

```
x = [1, 2, 3]
y = ['a', 'b', 'c']
for zipped in zip(x, y):
    print(zipped)
>>>
(1, 'a')
(2, 'b')
(3, 'c')
```

zip은 언제 사용할까??

- 소스 리스트와 파생 리스트를 묶을 때

```
names = ['Cecilia', 'Lise', 'Marie']
# list comprehension
letters = [len(n) for n in names]

longest_name = None
max_letters = 0

for name, count in zip(names, letters):
    if count > max_letters:
        longest_name, max_letters = name, count

print(longest_name, max_letters)
>>>
Cecilia 7
```

zip 사용 시 주의사항

- 입력받은 이터레이터들 중 하나라도 끝일 때 까지 tuple을 넘겨줌

```
x = [1, 2, 3]
y = [4, 5, 6, 7]

zipped = zip(x, y)
print(list(zipped))
>>>
[(1, 4), (2, 5), (3, 6)]
```

zip 사용 시 주의사항

- 길이가 다르다면 itertools의 zip_longest를 사용

```
import itertools

x = [1, 2, 3]
y = [4, 5, 6, 7]

zipLongest = itertools.zip_longest(x, y)
print(list(zipLongest))
>>>
[(1, 4), (2, 5), (3, 6), (None, 7)]
```

zip 사용 시 주의사항

- Python 2.x는 zip과 izip이 구분되어 있음

- zip은 tuple list를 리턴
  - range와 같이 순회 목적이라면 메모리, 속도의 낭비

- Izip은 제너레이터

- 길이가 다른 경우 izip_longest 사용

Ch12. for와 while 루프 뒤에는 else 블록을 쓰지 말자

루프 뒤의 else

```
for x in []:
    print('Never runs')
else:
    print('For Else block!')
>>>
For Else block!
```

루프 뒤의 else

```
for i in range(3):
    print('Loop %d' % i)
else:
    print('Else block!')
>>>
Loop 0
Loop 1
Loop 2
Else block!
```

루프 뒤의 else

```
for i in range(3):
    print('Loop %d' % i)
    if i == 1:
        break
else:
    print('Else block!')
>>>
Loop 0
Loop 1
```

루프 뒤의 else

```
while False:
    print('Never runs')
else:
    print('While Else block!')
>>>
While Else block!
```

루프 뒤의 else

- Python은 루프 뒤에 else블록이 올 수 있다.

- Else 의 의미가 조금 다름
  - 루프 조건에 맞지 않으면 실행된다는 의미가 아님
  - 루프에서 break가 걸려야 else문을 뛰어넘음

루프 뒤의 else

```
a = 4
b = 9

for i in range(2, min(a, b) + 1):
    print('Testing', i)
    if a % i == 0 and b % i == 0:
        print('Not coprime')
        break
else:
    print('Coprime')
>>>
Testing 2
Testing 3
Testing 4
Coprime
```

루프 뒤의 else

- 루프 뒤 else의 편리함 보다 코드 해석에 대해 마이너스 요소가 크므로 사용하지 말자

- 루프 뒤 else 사용 대신 헬퍼 함수를 사용하자

-루프 뒤 else 사용 대신 헬퍼 함수를 사용

```
def coprime(a, b):
    for i in range(2, min(a, b) + 1):
        if (a % i == 0) and (b % i == 0):
            return False
    return True

print(coprime(4, 9))
print(coprime(3, 6))
>>>
True
False
```

Ch13. try/except/else/finally에서 각 블록의 장점을 이용하자

try/except/else/finally

- try
  - 예외를 처리하고 싶은 코드 부분
- except
  - try블록 실행 중 오류 발생하면 수행됨
- else
  - try블록 실행 중 오류가 발생하지 않으면 수행됨
- finally
  - 무조건 수행됨

```python
import json
UNDEFINED = object()

def divide_json(path):
    handle = open(path, 'r+')   # May raise IOError
    try:
        data = handle.read()    # May raise UnicodeDecodeError
        op = json.loads(data)   # May raise ValueError
        value = (
            op['numerator'] /
            op['denominator'])  # May raise ZeroDivisionError
    except ZeroDivisionError as e:
        return UNDEFINED
    else:
        op['result'] = value
        result = json.dumps(op)
        handle.seek(0)
        handle.write(result)    # May raise IOError
        return value
    finally:
        handle.close()          # Always runs
```

예외 처리 시 else문

- try 블록이 예외를 일으키지 않으면 else블록이 실행됨

- try 블록의 코드를 줄일 수 있음

- 실제 수행하고 싶은 코드는 else 부분으로 구분되어 가독성을 높임