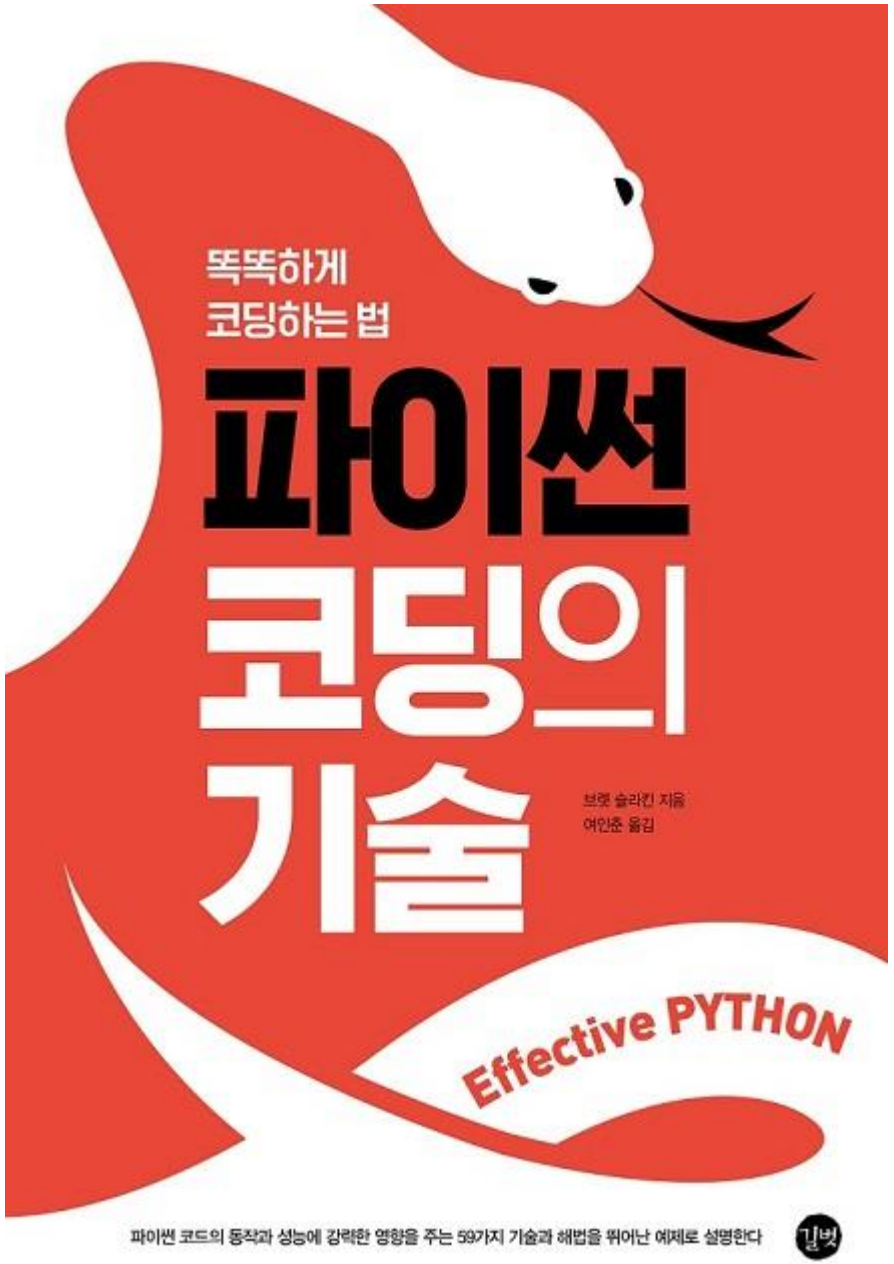
Effective python

강서연

2020. 01. 07



Chapter 1. *pythonic*

- Pythonic : 파이썬 다운 생각
- 단순함 선호, 가독성 극대화하기 위해서 명료함 선호
- Java, c++등 과 같은 스타일로 작성하기 보다는 파이썬답게 작성하는 것이 가장 중요.

BetterWay 1. 사용 중인 파이썬의 버전 알기

- 2.7 버전과 3.8버전
- 현재 파이썬2와 파이썬3 모두 활발히 사용
- Cpython, Jython, IronPython, PyPy 등 다양한 런타임 존재
- 2to3, six 와 같은 도구로 파이썬3으로 바꿀 수 있음

```
import sys
print(sys.version_info)
print(sys.version)
```

Python 2.7.0	July 3, 2010

Python 3.8.1	Dec. 18, 2019

```
item_01
"C:\Program Files\Python37\python.exe" C:/Users/PLAS/Desktop/3_winter/006764/item_
sys.version_info(major=3, minor=7, micro=3, releaselevel='final', serial=0)
3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC v.1916 64 bit (AMD64)]
```

BetterWay2. PEP8 가이드 따르기

1. White space
    1. 탭X, 스페이스 사용, 문법적 의미있는 들여쓰기는 스페이스 4칸
    2. 파일에서 함수와 클래스는 빈 줄 두개로 구분
    3. 클래스에서 메서드는 빈 줄 하나로 구분
    4. 리스트 인덱스, 함수 호출, 키워드 인수 할당에는 스페이스 사용X
    5. 변수 할당 앞뒤에 스페이스 하나만 사용

BetterWay2. PEP8 가이드 따르기

2. Naming

1. 함수, 변수, 속성 : lowercase_underscore
2. Protected : _leading_underscore
3. Private : _ _double_underscore
4. 클래스, 예외 : CapitalizedWord
5. 모듈 수준 상수 : ALL_CAPS
6. 클래스의 인스턴스 메서드에서 첫 파라미터(해당 객체를 참조)는 self로 함.
7. 클래스 메서드에서는 첫 번째 파라미터(해당 클래스를 참조)의 이름을 cls로 함

BetterWay2. PEP8 가이드 따르기

3. 표현식과 문장

1. 긍정 표현식의 부정 (if not a is b) 대신에 인라인 부정 (if a is not b)사용
2. 길이 확인으로 null check 하지X. 대신 if not listname 사용
3. 한 줄로 된 if문, for while loop, except 복합문 사용 X. 여러 줄로 나눠서 명료하게 작성
4. Import는 파일의 맨 위에
5. 모듈 import 시 모듈의 절대이름 사용. 상대경로 사용 X. bar패키지의 foo모듈 사용시 from bar import foo

BetterWay3. bytes, str, unicode의 차이점

- Python 2
  - str : raw 8bit
  - Unicode
  - Str이 7비트 아스키문자만 포함시, 연산자에 str과 Unicode 함께 사용 가능
- Python 3
  - Bytes : raw 8bit
  - Str : Unicode
  - >나 +와 같은 연산자에 bytes와 str 인스턴스 함께 사용 불가

  파이썬에서 유니코드에 연관된 바이너리 인코딩이 없음.
  Encode / decode 메서드 사용해야 함.

BetterWay3. bytes, str, unicode의 차이점

- 출력텍스트 인코딩(UTF-8) 유지하면서 다른 텍스트 인코딩(Latin-1, Shift JIS, Big 5) 수용하기 위해서
  - 파이썬 프로그래밍시, 외부에 제공할 인터페이스에서는 유니코드를 인코드하고, 디코드해야 함
  - 프로그램 핵심부에는 유니코드 문자타입(python3 – str / python2- unicode) 사용

BetterWay3. bytes, str, unicode의 차이점

- UTF-8(또는 다른 인코딩)으로 인코드된 문자인 raw 8bit 값을 처리
- 인코딩이 없는 유니코드 문자 처리 / 이 두 경우 다루기 위한 헬퍼함수

Str 반환

```python
# Example 1
def to_str(bytes_or_str):
    if isinstance(bytes_or_str, bytes):
        value = bytes_or_str.decode('utf-8')
    else:
        value = bytes_or_str
    return value  # Instance of str

print(repr(to_str(b'foo')))
print(repr(to_str('foo')))
```

bytes 반환

```python
# Example 2
def to_bytes(bytes_or_str):
    if isinstance(bytes_or_str, str):
        value = bytes_or_str.encode('utf-8')
    else:
        value = bytes_or_str
    return value  # Instance of bytes

print(repr(to_bytes(b'foo')))
print(repr(to_bytes('foo')))
```

```python
# Example 3
def to_unicode(unicode_or_str):
    if isinstance(unicode_or_str, str):
        value = unicode_or_str.decode('utf-8')
    else:
        value = unicode_or_str
    return value  # Instance of unicode

print(repr(to_unicode(u'foo')))
print(repr(to_unicode('foo')))
```

BetterWay3. bytes, str, unicode의 차이점

- Issue.
  - Python3에서 내장함수 open이 반환하는 파일 핸들을 사용하는 연산은 기본으로 utf-8 인코딩 사용.
  - Python2에선 binary 인코딩 사용.

  - 임의의 바이너리 데이터를 파일에 기록하려 하면 에러 발생

```
# Example 5
try:
    import os
    with open('random.bin', 'w') as f:
        f.write(os.urandom(10))
except:
    logging.exception('Expected')
else:
    assert False
```

```
Traceback (most recent call last):
  File "C:/Users/PLAS/Desktop/3_winter/006764/item_03.py
    f.write(os.urandom(10))
TypeError: write() argument must be str, not bytes
```

BetterWay3. bytes, str, unicode의 차이점

- Python3의 open에 새 encoding parameter가 추가되었기 때문
- 기본 값은 utf-8 / 따라서 파일 핸들을 사용하는 read와 write연산은 바이너리가 아닌 유니코드 인스턴스를 기대함.

- 문자쓰기모드('w')가 아닌 바이너리 쓰기모드('wb')로 하여 해결
- Read 시에도 마찬가지

```
# Example 6
with open('random.bin', 'wb') as f:
    f.write(os.urandom(10))
```

BetterWay4. 복잡한 표현식 대신 헬퍼 함수 작성

- 간결한 문법을 이용하면 많은 로직을 한줄로 쉽게 작성할 수 있음

다음과 같이 쿼리 문자열을 디코드 할 때

```
from urllib.parse import parse_qs
my_values = parse_qs('red=5&blue=0&green=',
                     keep_blank_values=True)
print(repr(my_values))
```

```
{'red': ['5'], 'blue': ['0'], 'green': ['']}
```

결과 딕셔너리에 get 메서드를 사용하면

```
# Example 2
print('Red:     ', my_values.get('red'))
print('Green:   ', my_values.get('green'))
print('Opacity: ', my_values.get('opacity'))
```

```
Red:      ['5']
Green:    ['']
Opacity:  None
```

BetterWay4. 복잡한 표현식 대신 헬퍼 함수 작성

- 파라미터가 없거나 비어있다면 0을 할당
- Boolean 표현식으로 처리한다면
  - 빈문자열, 빈 리스트, 0이 모두 함시적으로 false

```
red = my_values.get('red', [''])[0] or 0
green = my_values.get('green', [''])[0] or 0
opacity = my_values.get('opacity', [''])[0] or 0
```

➔

```
Red:     '5'
Green:   0
Opacity: 0
```

Or을 사용하여 간단하게 표현할 수 있음. But 아직 정수형으로 표현돼 있지 않음.

```
red = int(my_values.get('red', [''])[0] or 0)
green = int(my_values.get('green', [''])[0] or 0)
opacity = int(my_values.get('opacity', [''])[0] or 0)
```

➔

```
Red:     5
Green:   0
Opacity: 0
```

Int로 형변환하도록 수정하였지만, 읽기 매우 힘듦.

BetterWay4. 복잡한 표현식 대신 헬퍼 함수 작성

- If/else 조건식 (삼항문)이용하도록 하여 더 명확하게 표현

```
red = my_values.get('red', [''])
red = int(red[0]) if red[0] else 0
green = my_values.get('green', [''])
green = int(green[0]) if green[0] else 0
opacity = my_values.get('opacity', [''])
opacity = int(opacity[0]) if opacity[0] else 0
```

```
a if test else b
```

➔ Test가 true일 경우 a를,
그렇지 않으면 b를 리턴

- 복잡한 로직을 처리한다면 다음과같이 펼쳐서 작성할 수 있음.

```
# Example 6
green = my_values.get('green', [''])
if green[0]:
    green = int(green[0])
else:
    green = 0
print('Green:   %r' % green)
```

- 한 로직을 반복해서 사용해야 한다면 헬퍼함수를 만들어서 사용
- 무조건 짧은 코드를 만들기 보다는 가독성을 선택하는 편이 나음

```
# Example 7
def get_first_int(values, key, default=0):
    found = values.get(key, [''])
    if found[0]:
        found = int(found[0])
    else:
        found = default
    return found


# Example 8
green = get_first_int(my_values, 'green')
print('Green:   %r' % green)
```

BetterWay5. 시퀀스를 슬라이스하는 방법

- 파이썬은 시퀀스를 슬라이스해서 조각으로 만드는 문법을 제공
- 간단한 슬라이싱 대상, 내장타입 list, str, bytes
    - __getitem__ 과 __setitem__이라는 메서드를 구현하는 클래스에도 적용 가능
- 기본 형태 : somelist[start:end]
    - start 인덱스 포함, end 인덱스 제외

```
a = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
print('First four:', a[:4])
print('Last four: ', a[-4:])
print('Middle two:', a[3:-3])
```

    - 리스트의 처음 또는 끝을 인덱싱 할 경우. 하지만 생략하는 것을 추천

```
# Example 2
assert a[:5] == a[0:5]

# Example 3
assert a[5:] == a[5:len(a)]
```

BetterWay5. 시퀀스를 슬라이스하는 방법

- 리스트의 끝을 기준으로 오프셋을 계산할 때에는 음수로 표현하는게 편함.
  - Start와 end의 인덱스가 리스트의 경계를 벗어나도 적절히 처리됨.
  - 인덱스 음수 사용시 주의, somelist[-0:] 은 원본 리스트의 복사본

```
# Example 5
a[:]      # ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
a[:5]     # ['a', 'b', 'c', 'd', 'e']
a[:-1]    # ['a', 'b', 'c', 'd', 'e', 'f', 'g']
a[4:]     #                     ['e', 'f', 'g', 'h']
a[-3:]    #                           ['f', 'g', 'h']
a[2:5]    #           ['c', 'd', 'e']
a[2:-1]   #           ['c', 'd', 'e', 'f', 'g']
a[-3:-1]  #                           ['f', 'g']
```

  - 슬라이싱의 결과는 완전히 새로운 리스트라서 원본에 영향미치지X

BetterWay5. 시퀀스를 슬라이스하는 방법

- 할당을 사용하면 슬라이스는 원본리스트에서 지정한 범위를 대체
  - 길이가 달라도 동작함.

```
# Example 9
print('Before ', a)
a[2:7] = [99, 22, 14]
print('After  ', a)
```

```
Before  ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
After   ['a', 'b', 99, 22, 14, 'h']
```

- 시작과 끝 인덱스를 모두 생략하면 원본리스트의 복사본 얻음

```
# Example 10
b = a[:]
assert b == a and b is not a
```

- 시작과 끝 인덱스 생략한 채로 할당시.

```
# Example 11
b = a
print('Before', a)
a[:] = [101, 102, 103]
assert a is b          # Still the same list object
print('After ', a)     # Now has different contents
```

```
Before ['a', 'b', 99, 22, 14, 'h']
After  [101, 102, 103]
```

BetterWay6. 한 슬라이스에 start, end, stride를 함께 쓰지 말자

- Somelist[start:end:stride] stride로 간격 설정 가능
  - 시퀀스를 슬라이스 시 매 n번째 아이템을 가져옴

```
a = ['red', 'orange', 'yellow', 'green', 'blue', 'purple']
odds = a[::2]
evens = a[1::2]
print(odds)
print(evens)
```

```
['red', 'yellow', 'blue']
['orange', 'green', 'purple']
```

  - 문자열을 역순으로 만드는 방법에서 버그 생길 수 있음.
  - Stride로 -1을 사용하면 문자열을 역순으로. 바이트문자열이나 아스키문자에는 잘 동작. But utf-8 바이트 문자열로 인코드된 유니코드 문자에는 동작 X

```
# Example 2
x = b'mongoose'
y = x[::-1]
print(y)
```

```
b'esoognom'
```

```
# Example 3
try:
    w = '謝謝'
    x = w.encode('utf-8')
    y = x[::-1]
    z = y.decode('utf-8')
except:
    logging.exception('Expected')
else:
    assert False
```

```
UnicodeDecodeError: 'utf-8' codec can't decode byte 0x9d in position 0: invalid start byte
```

BetterWay6. 한 슬라이스에 start, end, stride를 함께 쓰지 말자

- -1을 제외한 다른 음수를 stride로 한다면

```
# Example 4
a = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
a[::2]   # ['a', 'c', 'e', 'g']
a[::-2]  # ['h', 'f', 'd', 'b']
```

- 다음과 같은 예제5는 대괄호에 숫자가 많아서 어떤 작용을 하는지 분명하지 않음. 따라서 stride를 start, end 인덱스와 함께 사용하지 말아야함.

```
# Example 5
a[2::2]     # ['c', 'e', 'g']
a[-2::-2]   # ['g', 'e', 'c', 'a']
a[-2:2:-2]  # ['g', 'e']
a[2:2:-2]   # []
```

BetterWay6. 한 슬라이스에 start, end, stride를 함께 쓰지 말자

- Stride를 사용해야 한다면 양수 값을 사용하고, start와 end 인덱스를 생략하는게 좋음

- 한 슬라이스에 start, end, stride를 함께 사용하는 상황은 피하자. 파라미터 세개를 사용해야 한다면 할당 두 개(하나는 슬라이스, 다른 하나는 스트라이드)를 사용하거나 내장 모듈 itertools의 islice 사용

```
b = a[::2]   # ['a', 'c', 'e', 'g']
c = b[1:-1]  # ['c', 'e']
```

➔ 슬라이싱부터 하고 스트라이딩을 하면 데이터의 얕은 복사본이 추가로 생김. 첫 번째 연산은 결과로 나오는 슬라이스의 크기를 최대한 줄여야 함.

BetterWay7.map과 filter대신 list comprehension을 사용하자

- 한 리스트에서 다른 리스트를 만들어내는 간결한 문법을 사용한 표현식을 list comprehension이라 함.
- Ex) 리스트의 각 숫자 제곱

```python
# Example 1
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
squares = [x**2 for x in a]
print(squares)
```

```python
# Example 2
squares = map(lambda x: x ** 2, a)
print(list(squares))
```

  - 인수가 하나뿐인 함수를 적용하는 상황이 아니면, 간단한 연산에는 내장함수 map보다 명확함.
  - Map 사용시 lambda함수를 생성해야 해서 깔끔해 보이지 않음

  - Item filtering도 가능

```python
# Example 3
even_squares = [x**2 for x in a if x % 2 == 0]
print(even_squares)
```

  - 내장함수 filter를 map과 연계해서 사용한 경우. 위와 결과는 같지만, 읽기 어려움

```python
# Example 4
alt = map(lambda x: x**2, filter(lambda x: x % 2 == 0, a))
assert even_squares == list(alt)
```

BetterWay7.map과 filter대신 list comprehension을 사용하자

- Dict와 set에도 list comprehension에 해당하는 문법이 있음.

```
# Example 5
chile_ranks = {'ghost': 1, 'habanero': 2, 'cayenne': 3}
rank_dict = {rank: name for name, rank in chile_ranks.items()}
chile_len_set = {len(name) for name in rank_dict.values()}
print(rank_dict)
print(chile_len_set)
```

```
{1: 'ghost', 2: 'habanero', 3: 'cayenne'}
{8, 5, 7}
```

BetterWay8. list comprehension에서 표현식을 두 개 넘게 쓰지 말자

- List comprehension에서는 다중루프를 지원.
  - 표현식은 왼쪽에서 오른쪽 순서로 실행

```
# Example 1
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flat = [x for row in matrix for x in row]
print(flat)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

  - 또 다른 다중루프 표현

```
# Example 2
squared = [[x**2 for x in row] for row in matrix]
print(squared)
```

```
[[1, 4, 9], [16, 25, 36], [49, 64, 81]]
```

BetterWay8. list comprehension에서 표현식을 두 개 넘게 쓰지 말자

```python
# Example 3
my_lists = [
    [[1, 2, 3], [4, 5, 6]],
    [[7, 8, 9], [10, 11, 12]],
]
flat = [x for sublist1 in my_lists
        for sublist2 in sublist1
        for x in sublist2]
print(flat)
```

```python
# Example 4
flat = []
for sublist1 in my_lists:
    for sublist2 in sublist1:
        flat.extend(sublist2)
print(flat)
```

Mylist를 flatten하는 식.

컴프리헨션을 일반 루프문으로 작성하면 들여쓰기를 사용해서 더 이해하기 쉬움

BetterWay8. list comprehension에서 표현식을 두 개 넘게 쓰지 말자

- 리스트 컴프리헨션도 다중 if 조건 지원. 같은 루프레벨에 여러 조건이 있으면 암시적인 and표현식이 됨.

```
# Example 5
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
b = [x for x in a if x > 4 if x % 2 == 0]
c = [x for x in a if x > 4 and x % 2 == 0]
```

```
[6, 8, 10]
[6, 8, 10]
```

- 조건은 루프의 각 레벨에서 for 표현식 뒤에 설정 할 수 있음. 다음과 같이 리스트 컴프리헨션으로 표현하면 간단하지만 이해하기 어려움.
  - Ex) Row의 합이 10이상이고 3으로 나누어 떨어지는 셀 구하는 표현식

```
# Example 6
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
filtered = [[x for x in row if x % 3 == 0]
            for row in matrix if sum(row) >= 10]
print(filtered)
```

```
[[6], [9]]
```

  - 리스트 컴프리헨션이 표현은 짧지만 이해하기 어려우므로 피해야함. 표현식이 두 개를 넘는다면 일반적인 if문과 for문을 사용하고 헬퍼함수를 작성해 사용.

BetterWay9.comprehension이 클 때는 generator표현식을 고려하자

- 리스트 컴프리헨션의 문제점. 입력 시퀀스에 있는 각 값별로 아이템을 하나씩 담은 새 리스트를 통째로 생성함. 입력이 큰 경우 메모리를 많이 소모해서 문제가 됨.

- 파일을 읽고 각 줄에 있는 문자 개수 반환하는 예제.
  - 리스트 컴프리헨션으로 표현시, 각 줄의 길이만큼 메모리 필요. 파일에 오류있거나 끊김이 없는 네트워크 소켓일 경우 문제가 발생.

```
value = [len(x) for x in open('my_file.txt')]
print(value)
```

```
[11, 47, 2, 40, 6, 83, 92, 7, 85, 101]
```

BetterWay9.comprehension이 클 때는 generator표현식을 고려하자

- generator 표현식 (list comprehension과 generator를 일반화) 제공
  - Iterator 로 평가되고 한 번에 한 출력만 만드므로 메모리 문제 해결.
  - Generator 함수로 표현 시 yield 사용.
    - Yield : 일반적인 함수의 경우 사용이 종료되면 메모리 상에서 클리어.
      generator 함수가 실행 중에 yield를 만나면, 해당 함수는 그대로 정지되며 반환 값을 next()를 호출한 위치로 전달. 함수가 종료되는게 아니라 그 상태로 유지됨.

```
def generator(n):
    i = 0
    while i < n:
        yield i
        i += 1

for x in generator(5):
    print x


0
1
2
3
4
```

  - ( ) 사이에 리스트 컴프리헨션과 비슷한 문법을 사용.

```
# Example 2
it = (len(x) for x in open('my_file.txt'))
print(it)

<generator object <genexpr> at 0x0000021611B3B7C8>
```

← 이터레이터로 평가되어서

```
print(next(it))    11
print(next(it))    47
```

BetterWay9.comprehension이 클 때는 generator표현식을 고려하자

- Generator 퍼포먼스
  - 100만개의 정보가 들어가는 리스트 생성 시
  - List 사용시

```
$ python generator.py
시작 전 메모리 사용량: 8.45703125 MB
종료 후 메모리 사용량: 311.4765625 MB
총 소요된 시간: 2.677738 초
```

generator 사용시

```
$ python generator.py
시작 전 메모리 사용량: 8.42578125 MB
종료 후 메모리 사용량: 8.42578125 MB
총 소요된 시간: 0.000003 초
```

- 제너레이터 표현식은 다른 제너레이터 표현식과 함께 사용할 수 있음.
  - 이터레이터를 전진시킬 때마다 내부 이터레이터도 전진시킴.

```
# Example 4
roots = ((x, x**0.5) for x in it)


# Example 5
print(next(roots))
```

```
(2, 1.4142135623730951)
```

BetterWay10. range보다는 enumerator를 사용하자

- Range는 정수집합을 iterate하는 루프를 실행 시 유용.

```
random_bits = 0
for i in range(64):
    if randint(0, 1):
        random_bits |= 1 << i
print(bin(random_bits))
```

- 순회할 자료구조 있으면 직접 루프를 실행할 수 있음

```
# Example 2
flavor_list = ['vanilla', 'chocolate', 'pecan', 'strawberry']
for flavor in flavor_list:
    print('%s is delicious' % flavor)
```

- 리스트의 아이템의 인덱스를 출력한다면, range를 사용하는 경우

```
# Example 3
for i in range(len(flavor_list)):
    flavor = flavor_list[i]
    print('%d: %s' % (i + 1, flavor))
```

```
1: vanilla
2: chocolate
3: pecan
4: strawberry
```

BetterWay10. range보다는 enumerat를 사용하자

- 앞선 경우는 읽기 불편함. Enumerate를 사용하여 개선할 수 있음.
- Enumerate는 지연 제너레이터로 이터레이터를 감싼다. 이 제너레이터는 이터레이터에서 루프인덱스와 다음 값을 한 쌍으로 가져와 넘겨줌.

```
# Example 4
for i, flavor in enumerate(flavor_list):
    print('%d: %s' % (i + 1, flavor))
```

```
1: vanilla
2: chocolate
3: pecan
4: strawberry
```

- 시작 숫자를 지정할 수 있음.

```
# Example 5
for i, flavor in enumerate(flavor_list, 1):
    print('%d: %s' % (i, flavor))
```

BetterWay11. iterator를 병렬로 처리하려면 zip을 사용하자

- 소스리스트와 파생된 파생리스트에 대하여.
- 파생리스트와 소스리스트의 아이템은 서로 인덱스로 연관되어 있음. 따라서 두 리스트를 병렬로 순회하려면 소스리스트의 길이만큼 순회하면 됨.

- Names와 letters를 인덱스로 접근한 코드 (읽기 어려움)

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


# Example 2
longest_name = None
max_letters = 0

for i in range(len(names)):
    count = letters[i]
    if count > max_letters:
        longest_name = names[i]
        max_letters = count
```

BetterWay11. iterator를 병렬로 처리하려면 zip을 사용하자

- <Betterway10. range보다는 enumerate를 사용하자>에서와 같이 enumerate를 사용하여 개선 (여전히 완벽하지는 않음)

```
# Example 3
longest_name = None
max_letters = 0
for i, name in enumerate(names):
    count = letters[i]
    if count > max_letters:
        longest_name = name
        max_letters = count
print(longest_name)
```

- Python3의 zip사용하여 개선
  - 지연 제너레이터로 이터레이터 두 개 이상을 감싼다. Zip 제너레이터는 각 이터레이터로부터 다음 값을 담은 튜플을 얻어옴.

```
# Example 4
longest_name = None
max_letters = 0
for name, count in zip(names, letters):
    if count > max_letters:
        longest_name = name
        max_letters = count
print(longest_name)
```

BetterWay11. iterator를 병렬로 처리하려면 zip을 사용하자

- 문제점 두가지

1. python2에서 제공하는 zip이 제너레이터가 아님.
    1. 제공한 이터레이터를 완전히 순회해서 zip으로 생성한 모든 튜플을 반환 ➔ 메모리 사용많음
    2. 매우 큰 이터레이터를 zip으로 사용시 내장 모듈 itertools에 있는 izip 사용

2. 입력 이터레이터들의 길이가 다르면 잘못 동작
    1. 예를 들어 소스리스트엔 아이템을 추가했지만 파생리스트에는 업데이트 하지 않은 경우
    2. 두 이터레이터의 길이가 다름. 길이가 짧은 이터레이터까지만 순회함
    3. 내장모듈 itertools의 zip_longest 사용하면 길이에 상관없이 병렬순회

BetterWay12.for와 while루프 뒤에는 else 블록을 쓰지 말자

- Python 의 루프에서 반복되는 내부블록 바로 다음에 else블록이 올 수 있다. 루프에서 break문을 사용해야 else블록을 건너 뛸 수 있음.

```
# Example 1
for i in range(3):
    print('Loop %d' % i)
else:
    print('Else block!')
```

```
Loop 0
Loop 1
Loop 2
Else block!
```

```
# Example 2
for i in range(3):
    print('Loop %d' % i)
    if i == 1:
        break
else:
    print('Else block!')
```

```
Loop 0
Loop 1
```

```
# Example 3
for x in []:
    print('Never runs')
else:
    print('For Else block!')
```

```
For Else block!
```

BetterWay12.for와 while루프 뒤에는 else 블록을 쓰지 말자

- 사용한 예제 / 두 숫자가 서로소인지 판별

```
# Example 5
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
```

- 다른 개발자가 봤을 때 이해하기 어렵기 때문에 이런 코드 지양해야 함. 절대 사용하지 말아야함.

BetterWay12.for와 while루프 뒤에는 else 블록을 쓰지 말자

- 헬퍼함수 작성하여 개선

1. 찾으려는 조건 찾았을 때 바로 반환

```python
# Example 6
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
```

2. 루프에서 찾으려는 대상을 찾았는지 알려주는 결과 변수를 사용

```python
# Example 7
def coprime2(a, b):
    is_coprime = True
    for i in range(2, min(a, b) + 1):
        if a % i == 0 and b % i == 0:
            is_coprime = False
            break
    return is_coprime
```

BetterWay13. try/except/else/finally에서 각 블록의 장점을 이용하자

- Finally 블록

```
handle = open('random_data.txt', 'w', encoding='utf-8')
handle.write('success\nand\nnew\nlines')
handle.close()
handle = open('random_data.txt')  # May raise IOError
try:
    data = handle.read()  # May raise UnicodeDecodeError
finally:
    handle.close()        # Always runs after try:
```

  - 예외를 전달하고 싶지만, 예외가 발생해도 정리 코드를 실행하고 싶을 때 try/finally 사용
  - 일반적인 사용 예, 파일 핸들러 종료시

  - Read에서 발생한 예외는 항상 호출코드까지 전달되며, handle의 close 메소드 또한 finally 블록에서 실행되는 것이 보장됨.

BetterWay13. try/except/else/finally에서 각 블록의 장점을 이용하자

- Else 블록
  - 코드에서 어떤 예외를 처리하고 어떤 예외를 전달할지 명확하게 하려면 try/except/else 사용해야 함
  - Try블록이 예외를 일으키지 않으면 else 블록 실행
  - Else 블록 사용시 try 블록의 코드 최소로 줄이고 가독성 높임.
  - Try 코드 정상 실행 후, finally 블록에서 정리코드 실행하기 전에 추가 작업 시 사용

```python
# Example 2
import json

def load_json_key(data, key):
    try:
        result_dict = json.loads(data)  # May raise ValueError
    except ValueError as e:
        raise KeyError from e
    else:
        return result_dict[key]         # May raise KeyError

# JSON decode successful
assert load_json_key('{"foo": "bar"}', 'foo') == 'bar'
try:
    load_json_key('{"foo": "bar"}', 'does not exist')
    assert False
except KeyError:
    pass  # Expected

# JSON decode fails
try:
    load_json_key('{"foo": bad payload', 'foo')
    assert False
except KeyError:
    pass  # Expected
```

BetterWay13. try/except/else/finally에서 각 블록의 장점을 이용하자

- 모두 함께 사용
- Try/except/else/finally
- Ex) 파일에서 수행할 작업 설명을 읽고 처리한 후 즉석에서 파일 업데이트
  - Try 블록 : 파일을 읽고 처리하는 데 사용
  - Except 블록 : try 블록에서 일어난 예외를 처리하는 데 사용
  - Else블록 : 파일을 즉석에서 업데이트, 이와 관련된 예외 전달되게 함
  - Finally블록 : 파일 핸들을 정리(예외가 일어나도 동작함)

```
# Example 3
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
```