

# 파이썬 코딩의 기술

4 주차

마영전 클라이언트유닛

김진용

## CH14. None을 반환하기보다는 예외를 일으키자

## 14. None을 반환하기보다는 예외를 일으키자

### None에 의미를 부여

```
def divide(a, b):  
    try:  
        return a / b  
    except ZeroDivisionError:  
        return None
```

```
x, y = 3, 0  
result = divide(x, y)  
if result is None:  
    print('Invalid inputs')
```

**None 반환이 자연스럽다.**

## 14. None을 반환하기보다는 예외를 일으키자

분모는 0이 아니라고 할 때 분자가 0이라면??

```
def divide(a, b):  
    try:  
        return a / b  
    except ZeroDivisionError:  
        return None
```

```
x, y = 0, 5  
result = divide(x, y)  
if not result :  
    print('Invalid inputs')
```

결과를 비교하는 구분을 실수로 **if not**으로 비교하면?  
당연히 잘못된 결과가 찍힌다.

## 14. None을 반환하기보다는 예외를 일으키자

### None과 False

```
if False == 0:
    print("equal")
else:
    print("not equal")    equal

if None == 0:
    print("equal")
else:
    print("not equal")    not equal

if None == False:
    print("equal")
else:
    print("not equal")    not equal
```

## 14. None을 반환하기보다는 예외를 일으키자



## 14. None을 반환하기보다는 예외를 일으키자

해결 방법 1 : 반환값을 두 개로 나눠서 튜플에 담기

```
def divide(a, b):  
    try:  
        return True, a / b  
    except ZeroDivisionError:  
        return False, None
```

### 의도대로 사용

```
success, result = divide(3, 6)  
if not success:  
    print('Invalid inputs')
```

### 잘못 사용

```
_, result = divide(3, 6)  
if not result:  
    print('Invalid inputs')
```

사용자가 결과의 첫 값을 무시하고 잘못 사용

## 14. None을 반환하기보다는 예외를 일으키자

### 해결 방법 2: 절대로 None을 반환하지 않기

```
def divide(a, b):  
    try:  
        return a / b  
    except ZeroDivisionError as e:  
        raise ValueError('Invalid inputs') from e
```

```
x, y = 5, 2  
try:  
    result = divide(x, y)  
except ValueError as error:  
    print( error )  
else:  
    print('Result is %.1f' % result)
```

raise : 오류를 강제로 발생시키기 위해 사용한다.

사용자는 반환값을 조건식으로 검사할 필요가 없다.  
예외가 없다면 반환값은 문제가 없다.



## 14. None을 반환하기보다는 예외를 일으키자

### 핵심 정리

- 특별한 의미를 나타내려고 None을 반환하는 함수가 오류를 일으키기 쉬운 이유는 None이나 다른 값(예를 들면 0이나 빈 문자열)이 조건식에서 False로 평가되기 때문이다.
- 특별한 상황을 알릴 때 None을 반환하는 대신에 예외를 일으키자.  
문서화가 되어있다면 호출하는 코드에서 예외를 적절하게 처리할 것이라고 기대할 수 있다.

## **CH15. 클로저가 변수 스코프와 상호 작용하는 방법을 알자**

## 15. 클로저가 변수 스코프와 상호 작용하는 방법을 알자

### 클로저 사용 예

```
def sort_priority(values, group):  
    def helper(x):  
        if x in group:  
            return (0, x)  
        return (1, x)  
    values.sort(key=helper)
```

```
numbers = [8, 3, 1, 2, 5, 4, 7, 6]  
group = {2, 3, 5, 7}  <- Set : 키만 남은 Dictionary  
sort_priority(numbers, group)  
print(numbers)
```

**[ 2, 3, 5, 7, 1, 4, 6, 8 ]**

## 15. 클로저가 변수 스코프와 상호 작용하는 방법을 알자

### 의도대로 동작하는 이유

- 파이썬은 **클로저**를 지원한다.

클로저란 자신이 정의된 스코프에 있는 변수를 참조하는 함수다.

- 파이썬에서 함수는 **일급 객체(first-class object)**다.

함수를 직접 참조하고 변수에 할당하고 다른 함수의 인수로 전달하고 표현식과 if문 등에서 비교할 수 있다.

- 파이썬은 튜플을 비교하는 **규칙**이 있다.

먼저 인덱스 0으로 아이템을 비교하고 다음 인덱스 순으로 진행한다.

## 15. 클로저가 변수 스코프와 상호 작용하는 방법을 알자

### 튜플 비교 예

```
def getkey( data ) :  
    return data[0]  
  
tuple = [(3, 0), (3, 1), (2, 8), (0, 10) ]  
tuple.sort(key=getkey)  
print(tuple)
```

**[(0, 10), (2, 8), (3, 0), (3, 1)]**

## 15. 클로저가 변수 스코프와 상호 작용하는 방법을 알자

### 지역 변수 스코프 버그??

```
def sort_priority(numbers, group):  
    found = False  
    def helper(x):  
        if x in group:  
            found = True  
            return (0, x)  
        return (1, x)  
    numbers.sort(key=helper)  
    return found
```

```
numbers = [8, 3, 1, 2, 5, 4, 7, 6]  
group = {2, 3, 5, 7}  
found = sort_priority(numbers, group)  
print('Found', found)  
print(numbers)
```

**Found False**  
**[2, 3, 5, 7, 1, 4, 6, 8]**

## 15. 클로저가 변수 스코프와 상호 작용하는 방법을 알자

변수 참조를 위한 **스코프 탐색 순서**

1. 현재 함수의 스코프
2. (현재 스코프를 담고 있는 다른 함수 같은) 감싸고 있는 스코프
3. 코드를 포함하고 있는 모듈의 스코프(전역 스코프라고도 함)
4. (len이나 str 같은 함수를 담고 있는) 내장 스코프

## 15. 클로저가 변수 스코프와 상호 작용하는 방법을 알아

### 해결방법

**nonlocal 문법** : 특정 변수 이름에 할당할 때 **스코프 탐색**이 일어나야 함을 나타낸다.  
제약은 모듈 수준까지 탐색할 수 없다.

- ※ 모듈(module) : 함수나 변수 또는 클래스 등을 모아 필요에 따라서 로드 할 수 있도록 만들어진 파일
- ※ global 문법 : 특정 변수 이름에 참조 할 때 **전역 변수 탐색**이 일어나야 함을 나타낸다.



## 15. 클로저가 변수 스코프와 상호 작용하는 방법을 알자

### nonlocal과 global

```
x = 3
def func():
    print("func %d" % x)

func()
print("global %d" % x)
```

func 3  
global 3

```
x = 3
def func():
    x = 5
    print("func %d" % x)

func()
print("global %d" % x)
```

func 5  
global 3

```
x = 3
def func():
    global x
    x = 5
    print("func %d" % x)

func()
print("global %d" % x)
```

func 5  
global 5

```
x = 3
def func():
    nonlocal x
    x = 5
    print("func %d" % x)

func()
print("global %d" % x)
```

SyntaxError: no binding for nonlocal 'x' found

```
def outerfunc():
    x = 3
    def innerfunc():
        nonlocal x
        x = 5
        print("innerfunc %d" % x)
    innerfunc()
    print("outerfunc %d" % x)

outerfunc()
```

innerfunc 5  
outerfunc 5

```
def outerfunc():
    x = 3
    def innerfunc():
        global x
        x = 5
        print("innerfunc %d" % x)
    innerfunc()
    print("outerfunc %d" % x)

outerfunc()
```

innerfunc 5  
outerfunc 3

## 15. 클로저가 변수 스코프와 상호 작용하는 방법을 알자

### 해결방법 : nonlocal 사용하기

```
def sort_priority(numbers, group):  
    found = False  
    def helper(x):  
        nonlocal found  
        if x in group:  
            found = True  
            return (0, x)  
        return (1, x)  
    numbers.sort(key=helper)  
    return found
```

하지만 nonlocal은 간단한 함수 이외에는 사용하지 말아야 한다.  
코드가 길어지면 해당 변수를 찾거나 이해하기 힘들어진다.

```
numbers = [8, 3, 1, 2, 5, 4, 7, 6]  
group = {2, 3, 5, 7}  
found = sort_priority(numbers, group)  
print('Found', found)  
print(numbers)
```

**Found True**  
**[2, 3, 5, 7, 1, 4, 6, 8]**

## 15. 클로저가 변수 스코프와 상호 작용하는 방법을 알자

해결방법 : nonlocal 대안(**class**)

```
class Sorter(object):
    def __init__(self, group):
        self.group = group
        self.found = False

    def __call__(self, x):
        if x in self.group:
            self.found = True
            return(0, x)
        return (1, x)
```

```
numbers = [8, 3, 1, 2, 5, 4 , 7, 6]
group = {2, 3, 5, 7}
sorter = Sorter(group)
numbers.sort(key=sorter)
assert sorter.found is True
```

sort()는 key인자로 받은게 클래스일 경우에는 내부적으로 \_\_call\_\_()을 부른다.

## 15. 클로저가 변수 스코프와 상호 작용하는 방법을 알자

### 파이썬2에서 문제 우회하기

```
def sort_priority(numbers, group):  
    found = [False] <- 값 수정이 가능한 리스트를 사용하여 문제를 우회하자  
    def helper(x):  
        if x in group:  
            found[0] = True  
            return (0, x)  
        return (1, x)  
    numbers.sort(key=helper)  
    return found[0]
```

어렵게 생각하지 말고 C를 아는 사람들이니  
간단하게 포인터 복사라고 생각하자

```
numbers = [8, 3, 1, 2, 5, 4, 7, 6]  
group = {2, 3, 5, 7}  
found = sort_priority(numbers, group)  
print('Found', found)  
print(numbers)
```

**Found True**  
**[2, 3, 5, 7, 1, 4, 6, 8]**

## 15. 클로저가 변수 스코프와 상호 작용하는 방법을 알자

### 핵심 정리

- 클로저 함수는 자신이 정의된 스코프 중 어디에 있는 변수도 참조 할 수 있다.
- 기본적으로 클로저에서 변수를 할당하면 바깥쪽 스코프에는 영향을 미치지 않는다.
- 파이썬3에서는 `nonlocal`을 사용하여 클로저를 삼싸고 있는 스코프의 변수를 수정 할 수 있음을 알린다.
- 파이썬2에서는 (아이템이 한 개만 있는 리스트 같은) 수정 가능한 값으로 `nonlocal`이 없는 문제를 우회한다.
- 간단한 함수 이외에는 `nonlocal`을 사용하지 말자

## **CH16. 리스트를 반환하는 대신 제너레이터를 고려하자**

## 16. 리스트를 반환하는 대신 제너레이터를 고려하자

### 제너레이터란?

- **yield** 표현식을 사용하는 함수다.
- 제너레이터 함수는 호출되면 실제로 실행하지 않고 바로 **이터레이터(iterator)**를 반환한다.
- 내장 함수 `next`를 호출할 때마다 다음 `yield`로 진행한다.

```
def func():  
    return "func()"  
  
def gen():  
    yield "gen()"  
  
print( type(func) )    <class 'function'>  
print( type(gen) )    <class 'function'>  
print( type(func()) ) <class 'str'>  
print( type(gen()) )  <class 'generator'>
```

```
def func():  
    yield 'a'  
    yield 'b'  
  
ret = func()  
print(ret)          <generator object func at 0x037DEB10>  
print(next(ret))    a  
print(next(ret))    b  
print(next(ret))    StopIteration
```

## 16. 리스트를 반환하는 대신 제너레이터를 고려하자

### 제너레이터를 고려 할만한 상황

```
def index_words(text):  
    result = []  
    if text:  
        result.append(0)  
    for index, letter in enumerate(text):  
        if letter == ' ':  
            result.append(index + 1)  
    return result
```

```
address = 'Four score and seven years ago...'  
result = index_words(address)  
print(result[:3])
```

[0, 5, 11]

1. 코드가 복잡하고 깔끔하지 않다.
2. 반환하기 전에 모든 결과를 리스트에 저장해야 한다.



## 16. 리스트를 반환하는 대신 제너레이터를 고려하자

### 제너레이터로 바꿔보자

```
def index_words(text):  
    result = []  
    if text:  
        result.append(0)  
    for index, letter in enumerate(text):  
        if letter == ' ':  
            result.append(index + 1)  
    return result
```



```
def index_words_iter(text):  
    if text:  
        yield 0  
    for index, letter in enumerate(text):  
        if letter == ' ':  
            yield index + 1
```

```
address = 'Four score and seven years ago...'  
result = index_words(address)  
print(result[:3])
```

```
address = 'Four score and seven years ago...'  
result = list(index_words_iter(address))  
print(result[:3])
```

## 16. 리스트를 반환하는 대신 제너레이터를 고려하자

### 핵심 정리

- 제너레이터를 사용하는 방법이 누적된 결과의 리스트를 반환하는 방법보다 이해하기에 명확하다.
- 제너레이터에서 반환한 이터레이너는 제너레이터 함수의 본문에 있는 `yield` 표현식에 전달된 값들의 집합이다.
- 제너레이터는 모든 입력과 출력을 메모리에 저장하지 않으므로 입력값의 양을 알기 어려울 때도 연속된 출력을 만들 수 있다.

## CH17. 인수를 순회할 때는 방어적으로 하자

## 17. 인수를 순회할 때는 방어적으로 하자

### 이해를 위한 예제 코드

```
def normalize(numbers):  
    total = sum(numbers)  
    result = []  
    for value in numbers:  
        percent = 100 * value / total  
        result.append(percent)  
    return result
```

```
visit = [15, 35, 80]  
percentages = normalize(visit)  
print(percentages)
```

**[11.538461538461538, 26.923076923076923, 61.53846153846154]**

## 17. 인수를 순회할 때는 방어적으로 하자

### 제너레이터 사용으로 문제가 되는 예제 코드

```
def read_visits(data_path):  
    with open(data_path, 'r') as f:  
        for line in f:  
            yield int(line)
```

```
def normalize(numbers):  
    total = sum(numbers)  
    result = []  
    for value in numbers:  
        percent = 100 * value / total  
        result.append(percent)  
    return result
```

```
it = read_visits("numbers.txt")  
percentages = normalize(it)  
print(percentages)
```

numbers - 메모장

파일(F)	편집(E)	서식
115		
35		
80		



[]

## 17. 인수를 순회할 때는 방어적으로 하자

### 문제가 일단은 수정된 예제 코드

```
def read_visits(data_path):  
    with open(data_path, 'r') as f:  
        for line in f:  
            yield int(line)
```

```
def normalize(numbers):  
    numbers = list(numbers)    <- 리스트로 복사해서 갖고 있도록 하자  
    total = sum(numbers)  
    result = []  
    for value in numbers:  
        percent = 100 * value / total  
        result.append(percent)  
    return result
```

```
it = read_visits("numbers.txt")  
percentages = normalize(it)  
print(percentages)
```

하지만 결국 메모리를 몽땅 잡게된다.

## 17. 인수를 순회할 때는 방어적으로 하자

### 이터레이터 반환을 사용하여 수정된 예제 코드

```
def read_visits(data_path):  
    with open(data_path, 'r') as f:  
        for line in f:  
            yield int(line)
```

```
def normalize_func(get_iter):  
    total = sum(get_iter())  
    result = []  
    for value in get_iter():  
        percent = 100 * value / total  
        result.append(percent)  
    return result
```

```
percentages = normalize_func(lambda: read_visits("numbers.txt"))  
print(percentages)
```

하지만 저자는 **lambda**가 세련되지 못하다고 말한다.

## 17. 인수를 순회할 때는 방어적으로 하자

### 클래스로 수정된 예제 코드

```
class ReadVisits(object):
    def __init__(self, data_path):
        self.data_path = data_path

    def __iter__(self):
        with open(self.data_path) as f:
            for line in f:
                yield int(line)
```

sum, for 등 루프나 내장 함수에 Iterable Object를 사용하면,  
해당 Iterable의 `__iter__()` 메서드를 호출하여 iterator를 가져온  
후 그 iterator의 `next()` 메서드를 호출하여 순회한다.

```
def normalize(numbers):
    total = sum(numbers)
    result = []
    for value in numbers:
        percent = 100 * value / total
        result.append(percent)
    return result
```

```
visits = ReadVisits("numbers.txt")
percentages = normalize(visits)
print(percentages)
```



```
def normalize(numbers):
    total = sum(numbers)
    for value in numbers:
        yield 100 * value / total
```

```
visits = ReadVisits("numbers.txt")
print( list( normalize(visits) ) )
```



## 17. 인수를 순회할 때는 방어적으로 하자

### 이터레이터를 지원하는 컨테이너를 보장하도록 수정

```
def normalize_defensive(numbers):  
    if iter(numbers) is iter(numbers):  
        raise TypeError('Must supply a container')  
    total = sum(numbers)  
    result = []  
    for value in numbers:  
        percent = 100 * value / total  
        result.append(percent)  
    return result
```

```
class ReadVisits(object):  
    def __init__(self, data_path):  
        self.data_path = data_path  
  
    def __iter__(self):  
        with open(self.data_path) as f:  
            for line in f:  
                yield int(line)
```

#### 잘 사용되는 예

```
visits = ReadVisits("numbers.txt")  
normalize_defensive(visits)  
print(percentages)
```

```
visits = [15, 35, 80]  
normalize_defensive(visits)  
print(percentages)
```

#### 잘못 사용되는 예

```
it = iter(visits)  
normalize_defensive(it)
```

**TypeError: Must supply a container**

## 17. 인수를 순회할 때는 방어적으로 하자

### 이터레이터를 지원하는 컨테이너를 보장하도록 수정

내장 함수 iter에 이터레이터를 넘기면 이터레이터 자체가 반환된다.  
반면에 iter에 컨테이너 타입을 넘기면 매번 새 이터레이터 객체가 반환된다.

```
visits = [15, 35, 80]
it = iter(visits)
print('-----1-----')
print(it)
print(visits)
print('-----2-----')
print(type(iter(it)))
print(type(iter(visits)))
print('-----3-----')
print(iter(it))
print(next(iter(it)))
print(iter(it))
print('-----4-----')
print(iter(visits))
print(next(iter(visits)))
print(iter(visits))
print('-----5-----')
print(iter(visits))
print(iter(visits))
```

-----1-----  
<list\_iterator object at 0x039332D0>  
[15, 35, 80]  
-----2-----  
<class 'list\_iterator'>  
<class 'list\_iterator'>  
-----3-----  
<list\_iterator object at 0x039332D0>  
15  
<list\_iterator object at 0x039332D0>  
-----4-----  
<list\_iterator object at 0x03933330>  
15  
<list\_iterator object at 0x039332F0>  
-----5-----  
<list\_iterator object at 0x039332B0>  
<list\_iterator object at 0x039332B0>

## 17. 인수를 순회할 때는 방어적으로 하자

### 핵심 정리

- 입력 인수를 여러 번 순회하는 함수를 작성할 때 주의하자.  
입력 인수가 이터레이터라면 이상하게 동작해서 값을 잃어버릴 수 있다.
- 파이썬의 이터레이터 프로토콜은 컨테이너와 이터레이터가  
내장 함수 `iter`, `next`와 `for` 루프 관련 표현식과 상호 작용하는 방법을 정의한다.
- `__iter__` 메서드를 제너레이터로 구현하면 자신만의 이터러블 컨테이너 타입을  
쉽게 정의할 수 있다.
- 어떤 값에 `iter`를 두 번 호출했을 때 같은 결과가 나오고  
내장 함수 `next`로 전진시킬 수 있다면 그 값은 컨테이너가 아닌 이터레이터다.