[오픈소스SW 활용] HW-1

오픈소스SW 활용 3분반, 32170578, 김산

In [1]:

```
import this
```

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

[Item 1] Know which version of Python you're using

In [2]:

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

sys.version_info(major=3, minor=8, micro=5, releaselevel='final', serial=0)
3.8.5 (default, Jan 27 2021, 15:41:15)
[GCC 9.3.0]

print(sys.version)을 통해 현재 사용중인 파이썬 버전을 확인 할 수 있습니다.

이러한 기능은 python버전에 따른 호환성을 고려해 개발해야 할때 유용할 것 같습니다.

[Item 2] Follow the PEP8 style guide as possible as you can

Whitespace

파이썬에서는 인덴트로 탭보다는 4개의 스페이스를 권고합니다.

현재 제가 사용중인 VScode라는 IDE에서는 각 언어마다 인덴트방식을 맞춰주는 기능을 제공하고 있어서 파이썬을 사용할 때에도 탭을 누르면 4개의 스페이스를 만들어주기때문에 크게 신경쓰지 않았지만 한번더 염두에 둘 수 있었습니다.

type annotation

return 값에 대해서는 type annotation을 해주는 편이 좋고, 이외 타입을 꼭 지정해주어야 하는 값들에 대해서 type annotation을 하는것이 좋을것 같다는 생각이 들었습니다. 매번 type annotation을 하는것은 빠른 속도로 구현할 수 있다는 파이썬의 강점을 살리지 못할것 같다는 생각에서 입니다.

2017년쯤에 Leetcode라는 알고리즘 문제를 푸는 사이트에서 파이썬으로 문제를 푼 기억이 있는데 당시에는 python문제에 type annotation을 지정해 주지않았지만, 18년부터는 문제를 제시할때 type annotation을 해놓아서 typeannotation에 대해 조금 찾아본 기억이 있습니다.

Leetcode(알고리즘 문제사이트)의 문제 예시입니다.

```
class Solution:
    def addTwoNumbers(self, l1: ListNode, l2: ListNode) -> ListNode:
```

In [3]:

```
pi : float = 3.142
def circumference(radius : float) -> float :
    return 2 * pi * radius
print(circumference(3))
```

18.852

Naming

- 함수, 변수, 인자는 소문자로작성
- 파이썬의 경우 자바, C++등과 달리 접근제어자가 언더스코어(_)를 통해 암묵적으로 존재합니다.
  - Public 메소드는 언더스코어 없이 시작합니다
  - Protected 메소드는 한개의 언더스코어로 시작합니다
  - Private 메소드는 두개의 언더스코어로 시작합니다

[Item 3] Know the differences between

String vs Bytes

In [8]:

```
a = b'h₩x65llo'
print(list(a))
print(a)
type(a)
```

```
[104, 101, 108, 108, 111]
b'hello'
```

Out[8]:

bytes

In [9]:

```
a = 'a₩u0300 propos'
print(list(a))
print(a)
type(a)
```

```
['a', '̀', ' ', 'p', 'r', 'o', 'p', 'o', 's']
à propos
```

Out[9]:

str

- byte 문자열의 경우 앞첨자에 b ,
- string 문자열의 경우 앞첨자가 없는것을 확인 할 수 있었습니다.

In [7]:

```
def to_string(bytes_or_str):
    if isinstance(bytes_or_str, bytes):
        value = bytes_or_str.decode('utf-8')
    else:
        value = bytes_or_str
    return value
print(repr(to_string(b'foo')))
print(repr(to_string('bar')))
```

```
'foo'
'bar'
```

In [15]:

```
string_byte = 'foo'.encode('utf-16')
string_utf = string_byte.decode('utf-16')

print(string_byte,type(string_byte))
print(string_utf,type(string_utf))
```

```
b'₩xff₩xfef₩x00o₩x00o₩x00' <class 'bytes'>
foo <class 'str'>
```

encode 와 decode() 함수를 통해 문자열type을 utf에서 바이트로, 바이트에서 utf로 바꿔줄수 있습니다

In [8]:

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

```
---------------------------------------------------------------------
-----
NameError                                 Traceback (most recent call last)
<ipython-input-8-68c2b62fd34c> in <module>
      5         value = bytes_or_str
      6     return value # instance of bytes
----> 7 print(repr(to_str(b'foo')))
      8 print(repr(to_str('bar')))

NameError: name 'to_str' is not defined
```

In [9]:

```
print('one' + b'two')
print('one' + 'two')
```

```
---------------------------------------------------------------------
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-9-ed3075a28305> in <module>
----> 1 print('one' + b'two')
      2 print('one' + 'two')

TypeError: can only concatenate str (not "bytes") to str
```

In [10]:

```
print(b'one' + two')
assert b'red' > 'blue'
```

```
  File "<ipython-input-10-7297fa546f35>", line 1
    print(b'one' + two')
                       ^
SyntaxError: EOL while scanning string literal
```

- Byte 자료형과 String자료형은 서로 호환되지 않기 때문에 유의해서 사용해야 합니다
  - TypeError: '>' not supported between instances of 'bytes' and 'str'

In [12]:

```
with open('data.bin','w') as f: # should be 'wb'
  f.write(b'₩xf1₩xf2₩xf3₩xf4₩xf5')
```

```
---------------------------------------------------------------------
-----
TypeError                 Traceback (most recent call last)
<ipython-input-12-5a914683f216> in <module>
    1 with open('data.bin','w') as f: # should be 'wb'
----> 2   f.write(b'₩xf1₩xf2₩xf3₩xf4₩xf5')

TypeError: write() argument must be str, not bytes
```

- byte형태의 문자를 작성하려고 할때 open() 시 r기본적으로 str 을 사용하기 때문에 에러가 발생합니다
- 따라서 open 옵션에 'b'가 아니라 'wb'를 사용해야 오류가 나지 않습니다

In [13]:

```
with open('data.bin','r') as f: # should be 'rb'
  data = f.read()
```

- 파일 입출력에서도 str 과 byte 를 구분하여 open() 함수의 두번째 인자를 설정해 주어야 합니다.

원래 String과 Byte의 구분에 대해서 크게 고려하지 않았었는데 이부분을 배우면서 한번더 염두에 둘 수 있어서 좋았습니다.

[Item 4] Prefer Interpolated F-Strings Over C-Style Format Strings and str.format

In [14]:

```
a = 0b10111011
b = 0xc5f
print('Binary is %d, hex is %d' % (a,b)) # Does this work?
```

Binary is 187, hex is 3167

이러한 % 연산자를 이용한 C-style의 formatting에는 다음과 같은 단점이 있습니다.

1. 타입 호환성을 고려해야한다
2. 읽기 어렵다
3. 같은 값에 대해서 여러번 formatter를 입력해야 한다
4. 딕셔너리 사용시 장황해진다

1.타입 호환성을 고려해야한다

In [15]:

```
# Should keep type compatibility
key = 'my_var'
value = 1.234
formatted = '%-10s = %.2f' % (key, value)
print(formatted)
```

```
my_var     = 1.23
```

2. 읽기 어렵다

In [19]:

```
pantry = [ ('avocados', 1.25), ('bananas', 2.5), ('cherries', 15)]
for i, (item, count) in enumerate(pantry):
   print('#%d: %-10s = %.2f' % (i, item, count))
# Difficult to read
for i, (item, count) in enumerate(pantry):
   print('#%d: %-10s = %.2f' % (i+1,item.title(),round(count)))
```

```
#0: avocados   = 1.25
#1: bananas    = 2.50
#2: cherries   = 15.00
#1: Avocados   = 1.00
#2: Bananas    = 2.00
#3: Cherries   = 15.00
```

3.같은 값에 대해서 여러번 formatter를 입력해야 한다

In [20]:

```
template = '%s loves food. See %s cook.'
name = 'Max'
formatted = template % (name, name) # duplicate name
print(formatted)
```

```
Max loves food. See Max cook.
```

4. 딕셔너리 사용시 코드가 장황해진다

In [21]:

```
template = '%(name)s loves food. See %(name)s cook.'
name = 'Max'
formatted = template % {'name': name} # using a dictionary
print(formatted)
```

```
Max loves food. See Max cook.
```

- 파이썬에서 C-style의 formatting이 가능한것을 처음 알았는데, 가독성도 떨어지고 코드의 길이가 너무 길어지는것 같아 자주 쓸 일은 없을것 같습니다.

In [23]:

```
a = 1234.5678
formatted = format(a, ',.2f')
print(formatted)

b = 'my string'
formatted = format(b, '^20s')
print(formatted)

key = 'my_var'
value = 1.234
formatted = '{} = {}'.format(key, value)
print(formatted)

formatted = '{:<10} = {:.2}'.format(key, value)
print(formatted)

print('%.2f%%' % a) # display %

formatted = '{1} = {0}'.format(key, value) # using positional index
print(formatted)
```

```
1,234.57
    my string
my_var = 1.234
my_var     = 1.2
1234.57%
1.234 = my_var
```

파이썬으로는 알고리즘 문제를 주로 풀었습니다. 그래서인지 문자열을 출력할 일이 잘 없어서 str.format 도 처음 접하게 되었는데 앞선 C-style에 비해 딕셔너리와 같은 이터레이터를 출력할 때 좀더 유리할 것 같습니다

In [17]:

```
# 왼쪽 정렬
s9 = 'align {0:<10} | done {1:<5} |'.format('left', 'a')
print(s9)


# 오른쪽 정렬
s10 = 'align {0:>10} | done {1:>5} |'.format('right', 'b')
print(s10)


# 가운데 정렬
s11 = 'align {0:^10} | done {1:^5} |'.format('center', 'c')
print(s11)
```

```
align left      | done a    |
align     right | done    b |
align   center  | done  c   |
```

또 출력을 깔끔하게 정리해야 할때 위의 예시처럼 사용하면 보기 좋은 출력이 가능할 것 같습니다

[Item 5] Write Helper Functions Instead of Complex Expressions

In [36]:

```
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': ['']}

- parse라이브러리 함수는 웹 크롤링이나 자연어 처리에 자주 사용했던 함수였습니다.
  - 기존에 사용하던 C++에서는 이러한 부분을 직접 구현해 주었는데 python에서는 이러한 부분이 사전에 라이브러리 함수에 구현이 되어있어서 데이터 처리에 매우 유용한것 같습니다.

parse_qs의 주역할은 토큰화입니다.

- keep_blank_values=False -> 공백 제외
  - {'red': ['5'], 'blue': ['0']}
- keep_blank_values=True -> 공백 포함
  - {'red': ['5'], 'blue': ['0'], 'green': ['']}

In [37]:

```
# For query string 'red=5&blue=0&green='
red = my_values.get('red', ['']) [0] or 0
blue = my_values.get('blue', ['']) [0] or 0
green = my_values.get('green', ['']) [0] or 0
opacity = my_values.get('opacity', ['']) [0] or 0

print(f'Red: {red!r}')
print(f'blue: {blue!r}')
print(f'Green: {green!r}')
print(f'Opacity: {opacity!r}')
```

Red: '5'
blue: '0'
Green: 0
Opacity: 0

In [42]:

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

red = get_first_int(my_values, 'red')
blue = get_first_int(my_values, 'blue')
green = get_first_int(my_values, 'green')
opacity = get_first_int(my_values, 'opacity')

print(f'Red: {red!r}')
print(f'blue: {blue!r}')
print(f'Green: {green!r}')
print(f'Opacity: {opacity!r}')
```

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

- 이부분은 모듈화의 좋은 예시였던것 같습니다. 자주 사용하고, 반복되는 코드는 함수로 정의해서 코드의 가독성을 올릴 수 있습니다.

In [33]:

```
from urllib.parse import urlparse, parse_qs

parts = urlparse('https://www.google.com/search?q=%EC%95%88%EB%85%95&oq=%EC%95%88%EB%85%95&aqs=chrome..69i57j0i433j0i131i433j0l4j69i60.2043j0j7&sourceid=chrome&ie=UTF-8')

print(parse_qs(parts.query))
```

```
{'q': ['안녕'], 'oq': ['안녕'], 'aqs': ['chrome..69i57j0i433j0i131i433j0l4j69i60.2043j0j7'], 's
ourceid': ['chrome'], 'ie': ['UTF-8']}
```

구글에 '안녕'을 검색했을때 url입니다.

- 이렇게 복잡한 문자열 형태로 이루어진 query string을 parse_qs 를 사용하여 key에 대해 value들을 list로 묶어서 dictionary로 반환할 수 있습니다.

[Item 6] Prefer Multiple Assignment Unpacking Over Indexing

In [43]:

```
snack_calories = {
   'chips' : 140,
   'popcorn' : 80,
   'nuts' : 190,
} ##imutable, ordered

items = tuple(snack_calories.items()) ## tuple() -> dictionary 에서 tuple값으로 가져옴
print(items)
```

(('chips', 140), ('popcorn', 80), ('nuts', 190))

In [44]:

```
keys = tuple(snack_calories.keys())
print(keys)

values = tuple(snack_calories.values())
print(values)
```

('chips', 'popcorn', 'nuts')
(140, 80, 190)

In [45]:

```
item = ('Peanut butter', 'Jelly')
first, second = item #Unpacking - single assignmet
print(first, 'and', second)
```

Peanut butter and Jelly

In [36]:

```
def bubble_sort(a):
   for _ in range(len(a)):
      for i in range(1,len(a)):
         if a[i] < a[i-1]:
            a[i-1], a[i] = a[i], a[i-1] #Unpacking syntax
names = ['pretzels','crraots','arugula','bacon']
bubble_sort(names)
print(names)
```

['arugula', 'bacon', 'crraots', 'pretzels']

In [37]:

```
test_set = ['ㄱ','ㄷ','ㄹ','Anecdot','Brown','lemon','airplane']
bubble_sort(test_set)
print(test_set)
```

['Anecdot', 'Brown', 'airplane', 'lemon', 'ㄱ', 'ㄷ', 'ㄹ']

- 문자열 비교연산은 아스키 코드값을 기준으로 이루어 지기 때문에 대문자, 소문자, 알파벳순으로 정렬이 되는것을 확인 할 수 있습니다.

In [38]:

```
def bubble_sort_low(a):
    for _ in range(len(a)):
        for i in range(1,len(a)):
            if a[i].lower() < a[i-1].lower():
                a[i-1], a[i] = a[i], a[i-1]
```

In [40]:

```
test_set = ['ㄱ','ㄷ','ㄹ','Anecdot','Brown','lemon','airplane']
bubble_sort_low(test_set)
print(test_set)
```

['airplane', 'Anecdot', 'Brown', 'lemon', 'ㄱ', 'ㄷ', 'ㄹ']

- lower() 함수를 통해 소문자화 한 후 비교한다면 대소문자에 관계없이 알파벳 순으로 정렬할 수 있습니다

In [47]:

```
snacks = [('bacon',350),('donut',240),('muffin',190)]
for i in range(len(snacks)):
    item = snacks[i]
    name = item[0]
    calories = item[1]
    print(f'#{i+1}: {name} has {calories} calories')
```

#1: bacon has 350 calories
#2: donut has 240 calories
#3: muffin has 190 calories

In [48]:

```
for rank, (name,calories) in enumerate(snacks,1): # 1 is a starting index
    print(f'#{rank}: {name} has {calories} calories')
```

#1: bacon has 350 calories
#2: donut has 240 calories
#3: muffin has 190 calories

- index보단 unpacking을 사용하는것이 더 간략하고 편리한 것 같습니다
- enumerate함수를 통해 for문을 훨씬 간결하게 만들수 있었습니다.
    - 저는 평소 C언어에 익숙해서인지 파이썬으로도 전자의 코드로 많이 사용했지만, 후자의 경우처럼 pythonic한 방식으로 코드를 작성하면 훨씬더 직관적으로 이해할 수 있는 간결한 코드로 작성할 수 있는것 같습니다.

[Item 7] Prefer enumerate Over range

In [49]:

```
from random import randint
random_bits = 0
for i in range(32):
    if randint(0, 1):
        random_bits |= 1 << i
print(bin(random_bits))
```

0b10010101001001010010101111100100

보수는 음수를 표현하기 위해서 자주 사용됩니다. 보수 표현방식에는 두가지 방식이 있습니다.

- one's complement
- two's complement

곱셈과 나눗셈

- 비트연산자로 수행

In [51]:

```
flavor_list = ['vanilla', 'chocolate', 'pecan', 'strawberry']
for flavor in flavor_list:
    print(f'{flavor} is delicious')
```

vanilla is delicious
chocolate is delicious
pecan is delicious
strawberry is delicious

In [52]:

```
for i in range(len(flavor_list)):
    flavor = flavor_list[i]
    print(f'{i+1}: {flavor}')
```

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

- 두가지 방식 모두 list를 iterate하여 결과를 출력하지만, 가독성이 떨어집니다. 책에서는 clumsy하다 라고 표현하였습니다

In [54]:

```
it = enumerate(flavor_list)
print(next(it))
print(next(it))
```

(0, 'vanilla')
(1, 'chocolate')

In [57]:

```
# Way 1
for i, flavor in enumerate(flavor_list):
    print(f'{i+1}: {flavor}')
```

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

In [58]:

```
# Way 2
for i, flavor in enumerate(flavor_list, 1):
    print(f'{i}: {flavor}')
```

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

enumerate라는 wrapper function을 이용한 출력예시입니다.

- Dummy 가 없어서 코드를 이해하기 좋고
- pythonic way에서는 loop보다 enumerate를 선호한다고 합니다

[Item 8] : Use zip to Process Iterators in Parallel

In [59]:

```
names = ['Cecilia', 'Lise','Marie']
counts = [len(n) for n in names] # A derived list, List comprehensions

max_count = 0
longest_name = None
for i in range(len(names)):
    count = counts[i]
    if count > max_count:
        longest_name = names[i]
        max_count = count
print(longest_name)
```

Cecilia

In [61]:

```
max_count = 0
longest_name = None
for i, name in enumerate(names):
    count = counts[i]
    if count > max_count:
        longest_name = name
        max_count = count
print(longest_name)
```

Cecilia

- zip은 2개이상의 iterator을 하나의 element pair로 만들어 튜플형태로 반환합니다
- lazy generator라 부르기도 합니다
  - 앞선 enumerate를 이용한 코드에는 인덱스를 위한 추가적인 대입연산이 필요했는데 zip을 통해 하나의 요소를 만들수 있어서 좀더 코드를 간결하게 만들 수 있었던것 같습니다.
  - 이와 비슷하게 C언어에서는 구조체를 따로 선언해서 구조체 배열로만든 뒤에야 이런식의 구현이 가능했는데 파이썬에서는 zip연산을 통해 그때그때 필요할때 사용할 수 있어서 편리한 것 같습니다.

In [64]:

```
for name, count in zip(names, counts):
    if count > max_count:
        longest_name = name
        max_count = count
print(longest_name)
```

Cecilia

In [65]:

```
names.append('Rosalind')
for name, count in zip(names, counts):
    print(name)
```

Cecilia
Lise
Marie

- zip은 동일한 개수의 자료형을 묶어주는 역할을 합니다.
- 어느 한쪽 이터레이터가 더 많을 경우 적은쪽의 이터레이터 기준으로 묶게 됩니다

In [78]:

```
std_num = [1,2,3]
for num, name, count in zip(std_num, names, counts):
    formatted = '{} : {:<7} , {}'.format(num,name,count)
    print(formatted)
```

1 : Cecilia , 7
2 : Lise   , 4
3 : Marie   , 5

- iterable한 자료형 여러개(3개이상)도 가능합니다

In [79]:

```
import itertools
for name, count in itertools.zip_longest(names, counts):
    print(f'{name}: {count}')
```

Cecilia: 7
Lise: 4
Marie: 5
Rosalind: None
Rosalind: None

- zip_longest() 는 기존의 zip과 달리 긴 이터레이터를 기준으로 묶어줍니다.
- 앞서 추가한 Rosalind는 count 값이 지정되어 있지 않기 때문에 None 으로 처리합니다.

[Item 9] : Avoid else Block after for and while Loops

for나 while loop가 끝났을때 else구문을 실행할 수 있습니다.

In [80]:

```
for i in range(3):
  print('Loop', i)
else:
  print('Else block!')
```

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

- try, except와 같은 예외처리와 사용법이 비슷한것 같습니다.

In [81]:

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

```
Loop 0
Loop 1
```

In [82]:

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

```
For Else block!
```

In [83]:

```
while False:
  print("never runs")
else:
  print("while else block!")
```

```
while else block!
```

for이나 while문 뒤에 else가 올수있다는것에 조금 헷갈렸지만 조건문을 포함 하는 문장에 대해 else가 올수있다 라는 생각을 하니 좀더 이해가 편했습니다

In [84]:

```
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('Comprise')
```

```
Testing 2
Testing 3
Testing 4
Comprise
```

In [85]:

```
# Way 1 returns the default outcome if I fall through the loop.
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
assert coprime(4, 9)
assert not coprime(3, 6)
```

In [87]:

```
# Way 2 breaks out of the loop as soon as I find something.
def coprime_alternate(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
assert coprime(4, 9)
assert not coprime(3, 6)
```

Item 10 Prevent Repetition with Assignment Expressions

Issue 1 : multiple reference

- a:=b "a walrus b"
- code duplication을 줄이기 위해 사용합니다
- := 모양이 코끼리(walrus)닮았다 해서 붙인 이름입니다

In [88]:

```
fresh_fruit = { 'apple': 10, 'banana': 8, 'lemon': 5}
def make_lemonade(count):
    print(count)
def out_of_stock():
    print('out_of_stock')
def make_cider(count):
    print(count)
count = fresh_fruit.get('lemon', 0)
if count: # Multiple references
    make_lemonade(count)
else:
    out_of_stock()
```

```
5
```

In [89]:

```
if count := fresh_fruit.get('lemon', 0): # using the walrus operator
    make_lemonade(count)
else:
    out_of_stock()

if (count := fresh_fruit.get('apple', 0)) >= 4: # using the walrus operator
    make_cider(count)
else:
    out_of_stock()
if (count := fresh_fruit.get('strawberry', 0)) >= 4: # using the walrus operator
    make_cider(count)
else:
    out_of_stock()
```

```
5
10
out_of_stock
```

개인적으로 walrus operation이 c언어에서 조건문안에 대입연산을 하는 것 if(pid = fork()) 과 비슷하다고 느꼈습니다.

Issue 2 Repetitive pattern

In [95]:

```
def slice_bananas(count):
    ...
class OutOfBananas(Exception):
    pass
def make_smoothies(count):
    ...
def slice_smoothies(count):
    ...
pieces = 0
count = fresh_fruit.get('banana', 0)
if count >= 2:
    pieces = slice_bananas(count)

try:
    smoothies = make_smoothies(pieces)
except OutOfBananas:
    out_of_stock()
```

In [96]:

```
if (count := fresh_fruit.get('banana', 0)) >= 2:
    pieces = slice_smoothies(count)
else:
    pieces = 0

try:
    smoothies = make_smoothies(pieces)
except OutOfBananas:
    out_of_stock()

print(smoothies)
```

None

- count와 관련한 연산이 조건문 내부로 들어가니 더 가독성이 좋은 코드가 된것 같습니다
- 앞서 언급한 C언어의 사례처럼 저도 C언어에서 자주 사용하는 방식이기도 합니다

Issue 3 : Simulate switch-case statement

In [ ]:

```
count = fresh_fruit.get('banana', 0)
if count >= 2:
    pieces = slice_bananas(count)
    to_enjoy = make_smoothies(pieces)
else:
    count = fresh_fruit.get('apple', 0)
    if count >= 4:
        to_enjoy = make_cider(count)
    else:
        count = fresh_fruit.get('lemon', 0)
        if count:
            to_enjoy = make_lemonade(count)
        else:
            to_enjoy = 'Nothing'
```

In [ ]:

```
if (count := fresh_fruit.get('banana', 0)) >= 2:
    pieces = slice_bananas(count)
    to_enjoy = make_smoothies(pieces)
elif (count := fresh_fruit.get('apple', 0)) >= 4:
    to_enjoy = make_cider(count)
elif count := fresh_fruit.get('lemon', 0):
    to_enjoy = make_lemonade(count)
else:
    to_enjoy = 'Nothing'
```

- 조금 예시가 극단적이긴 하지만, walrus operation을 가장 유용한 사례인것 같습니다.
- 조건문이 여러단계로 늘어져 있으면 가독성이 많이 떨어집니다.
- 자신의 코드를 통해 생각을 전달해야하는 오픈 소스에서는 특히나 이러한 부분들이 중요한 것 같습니다

Issue 4. Simulate do-while statments

In [98]:

```
def pick_fruit():
    ...
def make_juice(fruit, count):
    ...
bottles = []
fresh_fruit = pick_fruit()
while fresh_fruit:
    for fruit, count in fresh_fruit.items():
        batch = make_juice(fruit, count)
        bottles.extend(batch)
        fresh_fruit = pick_fruit()
```

In [99]:

```
bottles = []
while True: # Loop
  fresh_fruit = pick_fruit()
  if not fresh_fruit: # And a half
    break
  for fruit, count in fresh_fruit.items():
    batch = make_juice(fruit, count)
    bottles.extend(batch)
```

In [101]:

```
bottles = []
while fresh_fruit := pick_fruit():
  for fruit, count in fresh_fruit.items():
    batch = make_juice(fruit, count)
    bottles.extend(batch)
```

이러한 walrus operation은 아래처럼 대화형 앱이나 소켓 관련 앱을 만들때에도 유용하게 사용할 수 있을 것 같습니다

In [ ]:

```
while (command := input("> ")) != "quit":
  print(f"You entered: {command}")


while data := sock.recv(8192):
  print(f"Received data: {data}")
```

개발 모드와 운영 모드를 구분하기 위해서 애플리케이션이 돌아가는 환경의 환경변수를 사용해야 할 때가있는데, 이러한 환경변수는 코드 상에 저장하기가 난해합니다.

이때 OS라이브러리의 os.environ.get() 함수를 통해 환경변수를 가져올 수 있습니다.

In [102]:

```
import os
# Way 1
env_path = os.environ.get("PYTHONPATH", None)
if env_path:
  print(env_path)
# Way 2
if env_path := os.environ.get("PYTHONPATH", None):
  print(env_path)
```

```
/home/san/.vscode-server/extensions/ms-toolsai.jupyter-2021.5.702919634/pythonFil
es:/home/san/.vscode-server/extensions/ms-toolsai.jupyter-2021.5.702919634/pytho
nFiles/lib/python
/home/san/.vscode-server/extensions/ms-toolsai.jupyter-2021.5.702919634/pythonFil
es:/home/san/.vscode-server/extensions/ms-toolsai.jupyter-2021.5.702919634/pytho
nFiles/lib/python
```