

# Google Python Style Guide

Python Language Rules

학부생 연구원 박원영

<http://google.github.io/styleguide/pyguide.html>

# pylint

버그와 스타일 문제를 찾기 위한 도구 ex) 오타, 할당 전 변수 사용

```
C:\#>pylint --list-msgs
:blacklisted-name (C0102): *Black listed name "%s"*
    Used when the name is listed in the black list (unauthorized names).
:invalid-name (C0103): *%s name "%s" doesn't conform to %s*
    Used when the name doesn't conform to naming rules associated to its type
    (constant, variable, class, ...)
```

Pylint의 모든 경고 목록 불러오기

```
C:\#>pylint --help-msg=W1662
:comprehension-escape (W1662): *Using a variable that was bound inside a comprehension*
    Emitted when using a variable, that was bound in a comprehension handler,
    outside of the comprehension itself. On Python 3 these variables will be
    deleted outside of the comprehension. This message belongs to the python3
    checker.
```

특정 메시지(경고) 불러오기

# pylint

```
test.py x
1  def viking_cafe_order(spam, beans, eggs=None):
2      del beans, eggs
3      return spam + spam + spam
```

```
C:\WDev>pylint test.py
***** Module test
test.py:3:0: C0304: Final newline missing (missing-final-newline)
test.py:1:0: C0114: Missing module docstring (missing-module-docstring)
test.py:1:0: C0116: Missing function or method docstring (missing-function-docstring)

-----
Your code has been rated at 0.00/10
```

test.py에 대한 pylint 검사

# import

한 모듈에서 다른 모듈로 코드를 공유 (개별 클래스/함수 X)

객체 y, 모듈 x에 정의 (x.y)

import x : 모듈 전체를 갖고 올

from x import y : x모듈에서 y패키지를 갖고 올

from x import y as z : 두 모듈에서 y 갖고 오기 or y가 긴이름

sound.effects.echo 모듈을 갖고 올 경우

```
from sound.effects import echo
...
echo.EchoFilter(input, output, delay=0.7, atten=4)
```

# package

모듈의 전체 이름을 사용해서 갖고 올

장점 : 모듈 이름의 충돌 또는 잘못된 import 방지 가능

단점 : 패키지 계층 구조를 복제 필요

```
from absl import flags  
from doctor.who import jodie
```

# Exception(예외)

코드블록의 정상적인 제어 흐름을 벗어남 for 오류/예외 조건 처리

장점 : 제어 흐름의 복잡성 ↓

특정 조건 발생시 여러 프레임 통과 가능

단점 : 제어 흐름 혼동

# Exception(예외)

## 사용 조건

- 함수 형식으로 사용 ex) raise MyError ('Error message')
- built-in exception classs 사용
- try/except: 문 사용 but, 최소화
- try문의 예외 확인 → finally 사용

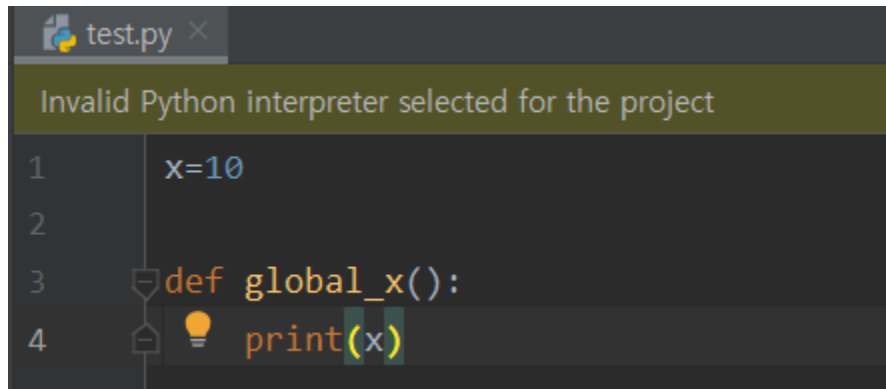
```
try:  
    raise Error()  
except Error as error:  
    pass
```

# Global Variables(전역변수)

모듈/클래스 속성에서 선언된 변수

모듈 처음에 수행 -> 모듈 동작 변경 O

전역변수는 변수지만 모듈레벨의 상수가 허용



```
test.py x
Invalid Python interpreter selected for the project
1 x=10
2
3 def global_x():
4     print(x)
```



# Nested/local/inner 클래스와 함수

중첩된 지역함수/클래스는 지역변수 종료시 사용

클래스 : 메서드, 함수, 클래스 내에서 정의

함수 : 메서드, 함수 내에서 정의

단점: 중첩/지역 클래스의 인스턴스는 선택X, 외부 함수 ↑

local값 초과하는 경우, 중첩 피하기

이름: \_모듈

# Comprehension & Generator expression

컨테이너 타입과 이터레이터 사용 -> 간결, 효율

장점 : 명확, 단순, 효율

단점 : 복잡하면 읽기 힘들

한 줄에 모든 식 작성

```
result = [mapping_expr for value in iterable if filter_expr]
```

```
squares_generator = (x**2 for x in range(10))
```

# Default iterator/operator

컨테이너 타입은 default iterator와 테스트 operator를 정의

장점 : 추가 메서드 호출X->단순/효율적, 모든 타입 지원->포괄적

단점 : 메서드 이름만으로 객체 타입 판별X

```
for key in adict: ...  
if key not in adict: ...  
if obj in alist: ...  
...
```

```
for key in adict.keys(): ...  
if not adict.has_key(key): ...  
for line in afile.readlines(): ...
```

# Generator

generator함수 실행 → iterator 반환(yield)

장점 : 코드의 단순화, 적은 메모리

함수에서의 사용 Yields: > returns:

# Lambda 함수

함수 내에서 익명함수를 정의

장점 : 편리

단점 : 지역함수보다 읽기&디버깅 ↓, 스택 추적의 이해 ↓, 표현력 ↓

lambda함수: 한 줄 사용 → 중첩함수: 코드 ↑ 사용

연산 : lambda함수 < 연산자 모듈의 함수

# Conditional Expression

조건식(3항 연산자) : if문보다 짧은 구문 제공

ex)  $x = 1 \text{ if cond else } 2$

간단한 식에 사용, if) 식이 길어지면 if문 전체 사용

-> 모든건 한줄에 有 : true-expression, if-expression, else-expression

# Conditional Expression

```
one_line = 'yes' if predicate(value) else 'no'
slightly_split = ('yes' if predicate(value)
                  else 'no, nein, nyet')
the_longest_ternary_style_that_can_be_done = (
    'yes, true, affirmative, confirmed, correct'
    if predicate(value)
    else 'no, false, negative, nay')
```

```
bad_line_breaking = ('yes' if predicate(value) else
                     'no')
portion_too_long = ('yes'
                    if some_long_module.some_long_predicate_function(
                        really_long_variable_name)
                    else 'no, false, negative, nay')
```

# Default Argument values

함수 파라미터에 변수값 지정 ex) `def foo(a, b=0):`

장점 : 드문 예외에 함수를 정의하지 않아도 수행 가능(오버로드)

단점 : 모듈 불러올 때 한번만 측정->변경 가능한 객체(list,dic) 문제

```
def foo(a, b=None):  
    if b is None:  
        b = []  
def foo(a, b: Optional[Sequence] = None):  
    if b is None:  
        b = []
```

```
def foo(a, b=[]):  
    ...  
def foo(a, b=time.time()):  
    ...
```



# Property

데이터(단순한 접근자/setter 메서드)에 접근&설정 사용

장점 : get/set 메서드 제거 → 가독성 향상  
클래스의 인터페이스 유지 (python)

단점 : object를 파이썬2에서 상속  
부작용(연산자 오버로딩) 숨김  
서브클래스 혼돈 야기

접근자/세터메서드의 데이터에 접근/지정 → @property 사용

# Property

```
import math

class Square(object):

    def __init__(self, side):
        self.side = side

    @property
    def area(self):
        return self._get_area()

    @area.setter
    def area(self, area):
        return self._set_area(area)
```

# True/False Evaluation

빈 값(0, None, [], {}, '') = false

오류 ↓, 속도 ↑

```
Yes: if not users:
    print('no users')

if foo == 0:
    self.handle_zero()

if i % 10 == 0:
    self.handle_multiple_of_ten()

def f(x=None):
    if x is None:
        x = []
```

```
No: if len(users) == 0:
    print('no users')

if foo is not None and not foo:
    self.handle_zero()

if not i % 10:
    self.handle_multiple_of_ten()

def f(x=None):
    x = x or []
```

문자열 '0' == True