PEP 8 – Style Guide for Python Code

📌

PEP 8 – Style Guide for Python Code | peps.python.org

Python Enhancement Proposals (PEPs)

https://peps.python.org/pep-0008/

Summary

문서의 목적 : 코드의 가독성을 향상시키고 일관성을 부여하기 위함

*하지만 항상 지켜야하는 것은 아님, 가이드라인을 따름으로써 가독성이나 일관성이 깨지거나, 수정으로 인해 이전 버전의 코드와 충돌하는 것과 같은 경우엔 가이드라인을 무시해도 괜찮음

Indentation (들여쓰기)

사용 : 스페이스 4개

should : 1. 첫번째줄에 인자가 오면 안 됨, 들여쓰기를 계속 사용할 경우 줄이 연결된다는 것을 알 수 있게 해야함

```
# Correct:
# Aligned with opening delimiter. (괄호에 맞춰 정렬)
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

# Add 4 spaces (an extra level of indentation) to distingui
sh arguments from the rest. (인자를 구분하기 위해 들여쓰기를 추가
로 사용)
def long_function_name(
```

```
        var_one, var_two, var_three,
        var_four):
    print(var_one)

# Hanging indents should add a level. (꼭 4 스페이스일 필요는
없음)
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

if문

```
if (this_is_one_thing and
    that_is_another_thing):
        # 주석도 들여쓰기 하기
    do_something()

if (this_is_one_thing
        and that_is_another_thing):
    do_something()
```

닫는 괄호

```
my_list = [
    1, 2, 3,
    4, 5, 6,
    ]
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
    )

my_list = [
    1, 2, 3,
    4, 5, 6,
]
```

```
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
)
```

❗ tab 말고 space를 사용하자

최대 줄 길이 (Maximum Line Length)

79자 제한 (docstring이나 주석의 경우에는 72자 제한)

(**우선 사용**) 긴 문장의 경우에는 괄호를 적절히 사용하여 여러 개의 문장으로 쪼개기

문장이 이어지는 것을 `\(backslashes)` 를 사용하여 나타낼 수 있음 (특히 with, assert 문)

줄바꿈 (Line Break Before or After a Binary Operator)

```
# Correct:
# easy to match operators with operands
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

연산자 앞뒤에서 줄바꿈하는 것 둘다 가능하지만, 이제부터는 위의 방식을 따르자

Blank

top-level function 이나 class 정의에는 2줄 띄기

class 안의 method 정의는 1줄 띄기

관련 있는 함수들의 그룹을 구분할 때 줄 띄기를 사용할 수 있음

*control-L은 공백

인코딩 (Encoding)

UTF-8 사용

모든 파이썬 식별자는 ASCII만 사용

SHOULD 영어 사용

임포트 (imports)

```
# 한 줄 씩
import os
import sys

from subprocess import Popen, PIPE
```

코드 파일의 맨 위에 import 선언 (모듈 주석과 doctstrings 다음에 모듈 설명 전에 …)

should import 순서 (줄띄기 금지)

1. Standard library imports.
2. Related third party imports.
3. Local application/library specific imports.

absolute import를 권장

```
import mypkg.sibling
from mypkgimport sibling
from mypkg.siblingimport example

from . import sibling
from .sibling import example

# importing a class from a class-containing module
from myclass import MyClass
from foo.bar.yourclass import YourClass

# If this spelling causes local name clashes
import myclass
```

```
import foo.bar.yourclass
# and use myclass.MyClass and foo.bar.yourclass.YourClass
```

should not Wildcard imports (`from <module> import *`)

Dunder Names

docstring 이후에, import문 전에 선언 `from __future__` 문은 예외

```
"""This is the example module.

This module does stuff.
"""

from __future__ import barry_as_FLUFL

__all__ = ['a', 'b', 'c']
__version__ = '0.1'
__author__ = 'Cardinal Biggles'

import os
import sys
```

문자열 인용 (String Quotes)

`''` `""` 둘다 사용 가능

`"""` `"""` 는 쌍따옴표 사용

공백 (Whitespace in Expressions and Statements)

```
spam(ham[1], {eggs: 2})

foo = (0,)

if x == 4: print(x, y); x, y = y, x
```

```
ham[1:9], ham[1:9:3], ham[:9:3], ham[1::3], ham[1:9:]
ham[lower:upper], ham[lower:upper:], ham[lower::step]
ham[lower+offset : upper+offset]
ham[: upper_fn(x) : step_fn(x)], ham[:: step_fn(x)]
ham[lower + offset : upper + offset]

spam(1)

dct['key'] = lst[index]

x = 1
y = 2
long_variable = 3

#
i = i + 1
submitted += 1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)
```

❌ : 튜플 안 쉼표 뒤 공백, `: 및 ,` 앞 공백, 괄호 앞뒤 공백, 필요없는 공백

⭕ : 이진 연산자(Binaray operator, `=`, `+=`, `-=`, `==`, `<`, `>`, `!=`, `<>`, `<=`, `>=`, `in`, `not in`, `is`, `is not`, `and`, `or`, `not`) 앞뒤에는 공백을 넣어준다!

```
# 함수 선언 앞뒤에 공백
def munge(input: AnyStr): ...
def munge() -> PosInt: ...

# 인자 지정 앞뒤에는 공백 X
def complex(real, imag=0.0):
    return magic(r=real, i=imag)

# default value를 지정할 때는 앞뒤에 공백 추가
def munge(sep: AnyStr = None): ...
```

```
def munge(input: AnyStr, sep: AnyStr = None, limit=1000):
...

# 수행문은 한 줄에 한 문장씩
if foo == 'blah':
    do_blah_thing()
do_one()
do_two()
do_three()
```

짧은 조건문은 한 줄에 써도 되지만, 여러 조건문에는 절대 ❌

Trailing Commas (가장 마지막 요소 다음에 오는 콤마...)

```
FILES = ('setup.cfg',)

FILES = [
    'setup.cfg',
    'tox.ini',
    ]
initialize(FILES,
           error=True,
           )
```

주석 (Comments)

완전한 문장으로 구성할 것, 영어로 작성할 것, 여러 문장 사이에는 공백 사용 가능

`#`한개로 시작

Inline Comments : 꼭 필요할 때만 사용하기

```
x = x + 1                 # Increment x ❌
x = x + 1                 # Compensate for border ⭕
```

문서화 (Documentation Strings)

모듈, 함수, 클래스, 메서드 모두에 docsring 작성하기

```
"""Return a foobang

Optional plotz says to frobnicate the bizbaz first.
"""

"""Return an ex-parrot."""
```

Naming Conventions

naming styles

- `lowercase`(소문자)
- `lower_case_with_underscores`(소문자 with 언더바)
- `UPPERCASE`(대문자)
- `UPPER_CASE_WITH_UNDERSCORES`(대문자 with 언더바)
- `CapitalizedWords`(CapWords)
- mixedCase

*접두사를 활용하여 관련 있는 그룹들을 묶는 데 사용하기도 함

- `_single_leading_underscore` : weak “internal use” indicator.

  * `from M import *` 는 _시작 object는 import 안 람
- `single_trailing_underscore_` : python 키워드와의 충돌을 피하기 위해

  ex. `tkinter.Toplevel(master, class_='ClassName')`
- `__double_leading_underscore` : 클래스 속성 이름 지을 때, 클래스의 속성의 이름을 바꿔버림
- `__double_leading_and_trailing_underscore__` : 클래스 속성 live in user-controlled namespaces

  * `__init__`, `__import__` or `__file__` 얘넨 사용 금지

naming coventions

- 소문자 L, 대문자 o, 대문자 i는 변수명으로 사용하지 않기

- ASCII compatible
- **Modules** : 짧은, all `lowercase` names (가독성을 높인다면 언더바 사용 가능)
- **Packages** : 짧은, all `lowercase` names (언더바 사용 자제)
- **Class** : `CapitalizedWords` , builtin names은 대부분 한 단어임, `CapitalizedWords` 규칙은 exception names과 builtin constants
- **Type Variable** : 짧은 `CapitalizedWords`

```
from typingimport TypeVar

VT_co = TypeVar('VT_co', covariant=True) # _co 접미사
KT_contra = TypeVar('KT_contra', contravariant=True) # _
contra 접미사
```

- **Exception** : class 규칙 적용, 접미사 Error 이용 가능
- **Global Variale** : funtion에서와 같음, `__all__` mechanism을 이용해 전역 변수의 exporting 방지
- **Function** : `lowercase` (가독성을 높인다면 언더바 사용 가능)
- **Variable** : `lowercase` (가독성을 높인다면 언더바 사용 가능)
- Always use `self` for the first argument to instance methods
- Always use `cls` for the first argument to class methods
- **Method** : `lowercase` (가독성을 높인다면 언더바 사용 가능)
- **Instance** : 1 leading 언더바 가능
- **Constant** : `CapitalizedWords` + 단어를 구분할 때 언더바 사용
- **상속을 위한 설계** : 클래스 메서드와 인스턴스가 non-public → public이 편함

Public and Internal Intrefaces

public과 internal interfaces를 구분해줘야함 (분명하게 선언해주지 않으면 public으로 간주함)

public API에서 `__all__` 속성을 사용하여 분명하게 명시 (`__all__` 이 empty로 설정되면 그 모듈은 no public API인 것임)

코딩 팁 (Programming Recommendations)

- PyPy, Jython, IronPython, Cython, Psyco 등에도 적용 가능하게 작성한다
- `if x` 대신 `if x is not None` 을 사용하기
- `not ... is` 대신 `is not` 를 사용하기
- 식별자에 직접적으로 `lambda`를 사용하기 보다는 `def`을 사용하기

```
def f(x): return 2*x # ⭕
f = lambda x: 2*x # ❌
```

- `BaseException` 대신 `Exception`을 사용하기
  - exception은 무엇이 문제인지 알 수 있을 정도로 설계하기 (exception chaining)
  - Error 접두사를 붙여 error exception 설계하기

```
try:
import platform_specific_module
except ImportError:
    platform_specific_module =None
```

- bare `except`는 피한다 (중지시키거나 에러 추적 힘듦) → `except BaseException:` 말고 `except Exception:` 이용하기
- `try`절은 최소 단위에서 시작해야함

```
try:
    value = collection[key]
except KeyError:
        return key_not_found(key)
else:
        return handle_value(value)
```

- 함수의 모든 return 절은 expression이나 none을 꼭 반환해야함, no value가 return되면 `return None` 명시

```
def foo(x):
if x >= 0:
return math.sqrt(x)
```

```
else:
returnNonedef bar(x):
if x < 0:
returnNonereturn math.sqrt(x)
```

- 특정 접두사나 접미사를 확인하려는 문자열 슬라이싱의 경우에는 `''.startswith()` 이나 `''.endswith()`를 사용하기

```
if foo.startswith('bar'): # ⭕
if foo[:3] == 'bar': # ❌
```

- 객체 type을 비교할 때는 `isinstance()` 사용하기

```
if isinstance(obj, int): # ⭕
if type(obj) is type(1): # ❌
```

- 문자열, 리스트, 튜플 등이 비었다면 `false`라고 판단하고 싶을 때

```
# ⭕
if not seq:
if seq:

# ❌
if len(seq):
if not len(seq):
```

- boolean 값이 True인지 False인지 비교할 때는 `==` 사용하지 않기

```
if greeting: # ⭕
if greeting == True: # ❌
```

Function Annotations

PEP 484 – Type Hints | peps.python.org

Python Enhancement Proposals (PEPs)

https://peps.python.org/pep-0484/

Variable Annotations

```
# colon 다음에 스페이스, = 앞뒤 스페이스

code: int

class Point:
    coords: Tuple[int, int]
    label: str = '<unknown>'
```

소감문

이 문서를 통해 크게 세 가지를 배울 수 있었다.

우선, 이 공식 가이드라인을 통해 내가 해왔던 잘못된 코드 작성에 대해 깨우치고 올바른 코드 작성법을 알 수 있었다. 코드 작성 방법이나 레이아웃에 대해서는 한번도 배운 적이 없기 때문에 지금까지 체계없이 코드를 짜거나, 나만의 규칙에 따라 그저 '깔끔해보이는 대로' 코드를 짰던 경험이 많았다. 특히 스페이스의 사용이나 닫는 괄호의 배치 등은 지금까지 완전히 잘못된 방식으로 사용하고 있었던 것을 알 수 있었다. 친구들과 괄호나 연산자의 앞뒤에서 공백 사용에 대해 이야기할 때, 모두가 올바른 작성법을 알지 못 하기 때문에 결국 정답을 모르고 넘어갔던 기억이 있다. 이 공식 문서를 꼼꼼히 읽고 정리한 경험을 바탕으로 앞으로 올바르고 가독성이 좋은 코드를 작성할 수 있을 것 같다.

다음으로는 '규칙'이라는 게 존재하는 지 몰랐던 코드 작성 분야에 대한 가이드라인을 배울 수 있었다는 점이다. 특히 naming 분야는 항상 그저 그 클래스, 메서드의 기능이나 핵심을 담으면 된다고 생각했지 특정한 가이드라인이 존재한다고는 생각해본 적 없었다. 하지만 이 공식 문서를 통해 내가 실제로 모델을 만들거나 구현하는 과정에서 depth가 깊어질수록 확실한 naming의 규칙이 필요하다는 생각이 들었다.

마지막으로는 체계적이고 구체적인 코드를 짜야겠다는 생각을 하게 되었다. 코딩을 하면서는 사실 class나 module을 불러와서 사용하는 경우가 대부분이기 때문에 attributes나 instance에 대해서 제대로 고민해본 적이 없다. 하지만 이 문서를 통해 naming을 시작으로 python의 식별자들이 철저하게 구분되는 것을 알 수 있었고, python의 구성 요소들에 대한 완벽한 이해를 기반으로 철저한 코드를 짜야겠다는 생각을 했다. 또한 항상 코드는 간결하게 짜야한다고 생각했는데, exception 코드의 작성 가이드라인을 통해 원활한 문제 추적을 위해서는 내 코드의 단계별로 구체적인 명시가 필요함을 깨닫게 되었다.