PEP 8 – Style Guide for Python Code

https://peps.python.org/pep-0008/#a-foolish-consistency-is-the-hobgoblin-of-little-minds

A Foolish Consistency is the Hobgoblin of Little Minds

“Readability counts”

Code Lay-out

Indentation

```
# Correct:

# Aligned with opening delimiter.
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

# Add 4 spaces (an extra level of indentation) to distinguish arguments from the rest.
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)

# Hanging indents should add a level.
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)


# No extra indentation.
if (this_is_one_thing and
    that_is_another_thing):
    do_something()

# Add a comment, which will provide some distinction in editors
```

```
# supporting syntax highlighting.
if (this_is_one_thing and
    that_is_another_thing):
    # Since both conditions are true, we can frobnicate.
    do_something()

# Add some extra indentation on the conditional continuation line.
if (this_is_one_thing
        and that_is_another_thing):
    do_something()


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',
)
```

Tabs or Spaces?

Spaces are the preferred indentation method.

Maximum Line Length

Limit all lines to a maximum of 79 characters.

Should a Line Break Before or After a Binary Operator?

Computers and Typesetting series: “Although formulas within a paragraph always break after binary operations and relations, displayed formulas always break before binary operations”

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

Blank Lines

Surround top-level function and class definitions with two blank lines.

Method definitions inside a class are surrounded by a single blank line.

이 외에 logical section을 나타내도록 blank line 사용

Source File Encoding

핵심 Python 배포판의 코드는 항상 UTF-8을 사용 (인코딩 선언 X)

noisy 유니코드 사용 금지

Imports

```
# Correct:
import os
import sys

# Wrong:
import sys, os

# Correct:
from subprocess import Popen, PIPE
```

Absolute imports are recommended

```
import mypkg.sibling
from mypkg import sibling
from mypkg.sibling import example
```

relative imports

```
from . import sibling
from .sibling import example
```

When 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`.

Module Level Dunder Names

모듈 레벨의 dunder는 Docstring 뒤에 그리고 import문 앞에 와야한다.

단, from **future** import 는 docstring 바로 뒤에 그 어떤 코드보다 먼저 위치해야한다. 그 뒤에 dunder가 위치한다.

```
"""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
```

String Quotes

single-quoted strings and double-quoted strings are the same.

따옴표가 string 내부에 있으면 다른 종류의 따옴표를 이용해서 string을 감싸라. (\이용 방지)

Whitespace in Expressions and Statements

Pet Peeves

Avoid extraneous whitespace

However, in a slice the colon acts like a binary operator, and should have equal amounts on either side

```
# Correct:
ham[lower+offset : upper+offset]

# Wrong:
ham[lower + offset:upper + offset]
```

Other Recommendations

Avoid trailing whitespace anywhere.

Function annotations

화살표가 `->`있는 경우 항상 화살표 주위에 공백이 있어야 함

```
# Correct:
def munge(input: AnyStr): ...
def munge() -> PosInt: ...
# Wrong:
def munge(input:AnyStr): ...
def munge()->PosInt: ...
```

Don’t use spaces around the `=` sign when used to indicate a keyword argument, or when used to indicate a default value for an *unannotated* function parameter

```
# Correct:
def complex(real, imag=0.0):
    return magic(r=real, i=imag)
# Wrong:
def complex(real, imag = 0.0):
    return magic(r = real, i = imag)
```

그러나 인수 주석을 기본값과 결합할 때는 `=`기호 주위에 공백을 사용하십시오.

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

When to Use Trailing Commas

Trailing commas are mandatory when making a tuple of one element.

튜플 외에 trailing commas가 유용한 경우: a list of values, arguments or imported items is expected to be extended over time.

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

Comments

Comments that contradict the code are **worse** than no comments.

Always make a priority of keeping the comments **up-to-date** when the code changes!

Should be complete English sentences, clear and understandable

Block comments

starts with a # and a single space

Inline comments

Should be separated by at least two spaces from the statement

Sometimes distracting if they state the obvious. Don’t do this:

```
x = x + 1                 # Increment x

# But sometimes, this is useful:

x = x + 1                 # Compensate for border
```

Documentation Strings

Write for all **public** modules, functions, classes, and methods

“”” that ends a multiline doctoring should be on a line by itself

```
"""Return a foobang

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

```
# if one liner docstrings
"""Return an ex-parrot."""
```

Naming Conventions

where an existing library has a different style, internal consistency is preferred.

Overriding Principle

Names that are visible to the user as public parts of the API should follow conventions that reflect usage rather than implementation.

Descriptive: Naming Styles

Naming styles that can be distinguished:

- `b` (single lowercase letter)
- `B` (single uppercase letter)
- `lowercase`
- `lower_case_with_underscores`
- `UPPERCASE`
- `UPPER_CASE_WITH_UNDERSCORES`
- `CapitalizedWords` (or CapWords, or CamelCase – so named because of the bumpy look of its letters [4]). This is also sometimes known as StudlyCaps.

  Note: When using acronyms in CapWords, capitalize all the letters of the acronym. Thus HTTPServerError is better than HttpServerError.
- `mixedCase` (differs from CapitalizedWords by initial lowercase character!)
- `Capitalized_Words_With_Underscores` (ugly!)
- Special Namings (`__double_leading_and_trailing_underscore__`: Never invent such names!..)

Prescriptive: Naming Conventions

Names to Avoid

Do not use I, l, O as single character variable (hard to distinguish)

ASCII Compatibility

Identifiers used in the standard library must be ASCII compatible.

Package and Module Names

Modules should have short, all-lowercase names (Underscores can be used)
Packages also (But use of underscore is discouraged)

Class names

Should normally use the CapWords convention

Type Variable Names

Should normally use CapWords preferring short names

Exception Names

Because exceptions should be classes, the class naming convention applies here.
(But should use the suffix “Error” if exception is error)

Global Variable Names

Should use the `__all__` mechanism to prevent exporting globals

Function and Variable Names

Should be lowercase, with words separated by underscores as necessary to improve readability.

Function and Method Arguments

Always use `self` for the first argument to instance methods.

Always use `cls` for the first argument to class methods.

To avoid arguments' name clashes, class_ is can be used.

Method Names and Instance Variables

Use the function naming rules.

Constants

Usually written in all capital letters with underscores separating words.

Designing for Inheritance

Always decide whether a class's methods and instance variables should be **public** or **non-public**.

Pythonic guidelines

- Public attributes should have no leading underscores.
- append a single trailing underscore if collides.
- For simple public data attributes, it is best to expose just the attribute name.
- Subclass have to considered naming them with double leading underscores and no trailing underscores.

Public and Internal Interfaces

It is important that users be able to clearly distinguish between public and internal interfaces.

Programming Recommendations

- Should be written in a way at other implementations of Python
  (PyPy, Python, Cython…)
- Comparisons to singletons (None..) should be done with 'is' or 'is not'

- Use ‘is not’ rather than ‘not … is’

```
# Correct:
if foo is not None:

# Wrong:
if not foo is None:
```

- It is best to implement all six operators (`__eq__`, `__ne__`, `__lt__`, `__le__`, `__gt__`, `__ge__`)
- Use a def statement instead of lambda
- Derive exceptions from `Exception` rather than `BaseException`.
- When catching exceptions, mention specific exceptions whenever possible instead of using a bare `except:`

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

- limit the `try` clause to the absolute minimum amount of code necessary.

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

# Wrong:
try:
    # Too broad!
    return handle_value(collection[key])
except KeyError:
```

```
    # Will also catch KeyError raised by handle_value()
    return key_not_found(key)
```

- When a resource is local to a particular section of code, use a `with` statement. And should be invoked through separate functions or methods.

```
# Correct:
with conn.begin_transaction():
    do_stuff_in_transaction(conn)

# Wrong:
with conn:
    do_stuff_in_transaction(conn)
```

- Be consistent in return statements. ‘return None’ should be at the end of the function

```
# Correct:

def foo(x):
    if x >= 0:
        return math.sqrt(x)
    else:
        return None

def bar(x):
    if x < 0:
        return None
    return math.sqrt(x)

# Wrong:

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

def bar(x):
```

```
    if x < 0:
        return
    return math.sqrt(x)
```

- Use `''.startswith()` and `''.endswith()` instead of string slicing to check for prefixes or suffixes.
- Object type comparisons should always use isinstance() instead of comparing types directly.
- Don’t compare boolean values to True or False using `==`.

Variable Annotations

- Should have a single space after the colon, Should be no space before the colon
- If an assignment has a right hand side, then the equality sign should have exactly one space on both sides:

```
# Correct:

code: int

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

# Wrong:

code:int  # No space after colon
code : int  # Space before colon

class Test:
    result: int=0  # No spaces around equality sign
```

소감

PEP 8을 읽은 후, 코드 스타일의 일관성이 코드의 가독성과 유지보수성에 어떻게 긍정적인 영향을 미칠 수 있는지에 대한 인식이 강화되었습니다. 스타일 가이드는 단순히 코드의 형식을 규정하는 것 이상으로, 팀이나 프로젝트 간에 일관성 있는 코딩 스타일을 유지함으로써 협업과 이해가 훨씬 원활해질 수 있다는 것을 이해했습니다. 특히 들여쓰기와 라인 길이의 제한은 코드를 읽는 데 있어 시선을 효과적으로 유도하고, 오랜 기간에 걸친 프로젝트에서의 유지보수를 훨씬 간편하게 만들어줄 것으로 느껴집니다.

또한, PEP 8이 파이썬 커뮤니티의 협력적이고 일관된 코드 작성 문화를 형성하는 데 어떻게 기여했는지에 대한 인상도 깊어졌습니다. 이러한 스타일 가이드가 개발자 간의 의사소통을 원활하게 하고, 새로운 프로젝트에 참여하는데 있어 진입 장벽을 낮춰준다는 점에서 프로페셔널한 소프트웨어 개발 환경을 조성하는데 큰 역할을 하는 것 같습니다.

각 class를 디자인할 때 고려한 부분:

- `BaseTokenizer` **클래스:**
  - **generalization:** 기본 토크나이저 클래스로서, 다양한 토크나이저를 구현할 수 있는 범용성을 고려하였습니다.
  - **상속성:** 파생 클래스에서 필요한 메서드를 구현하도록 하여, 이를 기반으로 다양한 토크나이저를 만들 수 있도록 설계되었습니다.
  - **설정 가능성:** 코퍼스를 초기화할 수 있는 생성자와 같이 클래스를 쉽게 설정할 수 있도록 고려하였습니다.
- `BPETokenizer` **클래스:**
  - **BPE Tokenizer의 핵심 아이디어를 반영:** BPE (Byte Pair Encoding) 알고리즘을 기반으로 Tokenizer를 디자인하였습니다. 이는 많은 언어 모델에서 효과적으로 사용되는 Tokenizer 중 하나입니다.
  - **데이터 구조 활용:** `vocab` 및 `merges`와 같은 데이터 구조를 활용하여 통계 및 병합 과정을 효과적으로 관리할 수 있도록 디자인되었습니다.
  - **확장성:** 상속받은 메서드를 구현하면서, 일반적인 기능에 대한 특수화 및 확장이 가능하도록 고려되었습니다.
- `WordTokenizer` **클래스:**
  - **단어 기반 토크나이저:** 단어를 기반으로 하는 토크나이저로서, 기본 텍스트 분석에 활용될 수 있도록 설계되었습니다.
  - **간결한 구조:** 상대적으로 간단한 구조로 표현되어, 단어 수준에서의 토크나이징에 중점을 두었습니다.
  - **문장 길이 관리:** 패딩 및 최대 길이 제한과 같은 텍스트의 길이를 관리하는 옵션을 제공하여 다양한 상황에서 유연한 사용이 가능하도록 고려되었습니다.

각 method의 구조와 작동 원리:

- `BaseTokenizer` :
  - `add_corpus` : 파생 클래스에서 구현해야 하는 메서드로, 코퍼스를 토크나이저에 추가하는 역할을 합니다.
  - `get_stats` : 통계 정보를 추출하는 메서드로, 파생 클래스에서 구현해야 합니다.
  - `merge_vocab` : 어휘를 병합하는 메서드로, 파생 클래스에서 구현해야 합니다.
  - `train` : 토크나이저를 학습시키는 메서드로, 파생 클래스에서 구현해야 합니다.
  - `tokenize` : 주어진 텍스트를 토큰화하는 메서드로, 파생 클래스에서 구현해야 합니다.
- `BPETokenizer` **클래스:**
  - `add_corpus` : 주어진 코퍼스에서 BPE 토크나이저의 초기 어휘를 구축합니다.
  - `get_stats` : 주어진 어휘에서 바이그램 통계를 계산합니다.
  - `merge_vocab` : 주어진 바이그램을 사용하여 어휘를 병합합니다.
  - `train` : 주어진 반복 횟수만큼 BPE 토크나이저를 학습시킵니다.

    - `tokenize`: 주어진 텍스트를 BPE 토큰으로 변환합니다.
- `WordTokenizer` **클래스:**
    - `add_corpus`: 주어진 코퍼스에서 단어 토크나이저의 초기 어휘를 구축합니다.
    - `tokenize`: 주어진 텍스트를 단어로 토큰화합니다. 이때, 어휘에 없는 단어는 `<unk>`로 처리합니다. 패딩 및 최대 길이 제한 옵션을 고려하여 텍스트를 관리합니다.