DS 코딩 캠프 1주차 과제 보고서

PEP 8 – Style Guide for Python Code

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 distingui
sh 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_na
me(
    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 edi
tors


# 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 continuatio
n 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 section 을 나타내도록 blank line 사용

Source File Encoding

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

noisy 유니코드 사용 X

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

module level의 dunder는 Docstring 뒤에 그리고 import문 앞에 와야 한다.

단, from future import는 Docstring 바로 뒤에 그 어떤 코드보다 먼저 위치해야 한다. 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 공백 사용

```
# 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

파이썬을 취급하는 실력과는 무관하게, 파이썬을 접하게 된 지 적지 않은 시간이 흘렀음에도 코드 스타일링 관련해서 모르거나 잊고 있었던 부분이 너무 많아 읽길 잘 했다는 생각을 하면서 과제에 임했던 것 같습니다.

Tokenizer 구현 보고서

Class 디자인 시 고려한 부분

상위 클래스를 구현하여 상속 메커니즘을 사용하려 했지만, 발생하는 오류를 해결하지 못해 일단 각 클래스의 개별적 구현에 힘썼습니다.

https://velog.io/@gwkoo/BPEByte-Pair-Encoding
알고리즘의 효율적인 변경에 관해서는 좋은 인사이트를 얻지 못해, 해당 링크에서 BPE를 위해 사용한 함수들을 이용해 작동하는 알고리즘을 구현하려 했습니다.

WordTokenizer class의 경우 단순 .split()함수를 이용한 tokenize 메커니즘만 구현하였습니다.

method 구조와 작동 원리 및 제시된 조건 충족 방법

get_dictionary method를 이용해 어절 별 character 단위로 구분된 후보를 사전 생성합니다.
그 후 get_pairs method를 이용해 바이그램 페어를 생성합니다.
마지막으로 merge_dictionary method를 이용해 가장 자주 등장한 바이그램 페어를 merge하여 딕셔너리를 업데이트하는 일련의 과정을 생성하고, train method로 주어진 반복 횟수만큼 훈련을 진행시킵니다. get_vocab method를 통해 훈련된 딕셔너리를 기반으로 최종 vocabulary를 생성하고, 이를 통해 tokenize method를 구현해야 했지만 train method 실행 과정에서
ValueError: max() arg is an empty sequence
라는 오류가 발생하여 vocabulary 생성 자체에서 문제가 생겼습니다. 각 method의 실행 과정에서 도출된 결과값이 호환되지 않아 발생하는 문제라고 생각했지만 관련 해결 방법에 대해 알아내지 못해 그대로 제출하게 되었습니다.
과제 만들어 주시느라 너무너무 고생하셨을 텐데 죄송합니다…

협업한 방식

과제 1의 요약은 전반부와 후반부를 나눠 분담하는 방식으로 작성하였고, (건설적인 해결책을 나누지는 못했지만) BPETokenizer의 train method 설계 관련 문제와 이전에 서술한 ValueError 관련 논의를 진행하였습니다.