#######소감문#######

23기 DS 이성현
24기 박경욱

이성현
글로 정리된 것을 보니 기존에 알던 것, 다른 코드에서 자주 본 것, 그리고 새롭게 알게 된 것들이 있었다. 가독성의 중요성에 대해서 다시 한번 생각하게 되었고, 실제 코딩에도 제대로 적용해보고 싶다는 의욕이 들었다.

박경욱
Programming Recommendations 부분에서 is와 ==의 적절한 활용이 필요하다고 되어 있다. 저도 가끔씩 이 둘이 헷갈려 혼용해서 쓰기도 하는데 조심해서 써야겠다는 생각을 했습니다. 또한 Python에서 제공하는 함수가 뭐가 있는지도 찾아보며 공부해야겠다는 생각도 들었습니다. 저는 그동안 c++을 많이 사용하여 Python에서 제공하는 startswith, endswith와 같은 함수의 존재를 잘 모릅니다. 가독성을 위해, 또한 코드 작성의 편리함을 위해서라도 알아두는 것이 좋다고 생각합니다.

#######내용 정리#######

1. Readability
코드는 쓰는 것보다도 읽는데 더 많은 쓰임을 가지고 있다. 그러므로 코드를 작성할 때는 쓰기 편한 것보다도 읽기 편한 코드를 짤 줄 아는 사람이 훨씬 더 중요하다. 잘 쓴 코드의 첫 번째 할 일은 일관성 있는 스타일의 코드를 쓰는 것이다. 일관성이 없으면 가독성은 떨어지기 마련이다. 물론 상황에 따라 일관되지 않은 코드 작성 스타일이 있을 수 있고, 이는 명확한 기준이 있다고 볼 수는 없다. 그냥 적당히 보고 판단해야 한다.
- 해당 문서가 제시한 가이드라인이 오히려 가독성을 떨어트릴 때
- 다른 코드들과의 일관성을 유지해야 하는 상황일 때
- 해당 문서보다 더 이전의 코드였고, 더는 수정될 이유가 없는 코드일 때
- 과거 버전의 파이썬을 기반으로 작성된 다른 코드와의 호환성을 위해 스타일을 어떤 방식으로든 희생해야 할 때

2. 코드 레이아웃
들여쓰기는 4 칸으로 하자.
여러 argument에 대해서 들여쓰기를 할 경우, 첫 줄에는 argument가 없어야 하며, 서로 연속된 줄이라는 것을 알 수 있게 작성해야 한다.
============예시============
```
# 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) # 이런 것을 Continuation line이라고 부른다.
==================================
```

Continuation line은 4칸 들여쓰기가 강제되지는 않으나 4칸 들여쓰기가 권장된다.
if 문에 대해서도 몇 가지 권장되는 형태의 들여쓰기 방식들이 있지만, 여기에 형식이 반드시 한정되지는 않는다.

```
============예시============
# 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()
============예시============
```

괄호(' ( ) ', ' [ ] ', ' { } ')는 줄 띄움을 해주는 편이 더 놓은데 들여쓰기가 되어 있는 만큼 동일하게 해주거나 아예 들여쓰기를 하지 않는 방법도 있다.

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

# method 2
my_list = [
```

```
    1, 2, 3,
    4, 5, 6,
]
============예시============
```

tab과 space 중에서는 뭐가 들여쓰기로 사용하기 더 좋은가?
space가 들여쓰기의 방식으로서 더 적합하다.
(이유는 정확히 무슨 말인지 이해를 못했다.)
Tabs should be used solely to remain consistent with code that is already indented with tabs.

maximum line length
한 줄에 79글자를 넘지 않도록 하라. 다른 파일 형식이나 파일 내부 등에서 더 호환성을 높이고 싶으면 72글자로 줄이는 것이 낫다. 실제 작업시에 화면을 분할하여 작업하기에 적은 글자수를 지니는 것이 좋다. 줄을 나누는 좋은 방법 중 하나는 ' \ ' 기호를 쓰는 것이며, 변수에 긴 문자열(주로 파일 경로에 해당한다)을 저장하는 방법도 있다.

연산자를 줄 앞에 둘 것인가 뒤에 둘 것인가?
지금까지 자주 권장되어 온 작성 스타일은 연산자를 줄의 맨 뒤에 두는 것이다.
그러나 가독성의 측면에서 그리 좋지 않기 때문에 연산자 (+ , - 등) 는 각 줄의 맨 앞에 두는 편이 좋다.

```
============예시============
income = (gross_wages
                + taxable_interest
                + (dividends - qualitied_dividends)
                - ira_deduction
                - student_loan_interest)
============예시============
```

빈 줄
함수(function)은 한 줄을 띄운다.
클래스(class)는 두 줄을 띄운다.

소스 파일 인코딩
파일은 UTP-8로 인코딩되어야 하며, encoding declaration을 가져서는 안 된다.

import
import를 할 때 각 라이브러리를 한 줄씩 불러오는 것이 바람직하지만,
한 라이브러리에서 여러 함수를 불러오는 것은 한 줄로 작성하는 것이 좋다.

```
============예시============
import os
import sys
from subprocess import Popen, PIPE
============예시============
```

import는 반드시 파일의 맨 위에 둔다.

import의 순서는
1. 표준 라이브러리 (time, random 등)
2. 서드 파티 라이브러리 (pytorch 등)
3. 로컬 라이브러리 (ex - src.model 등)
이며, 각 그룹끼리는 한 줄을 띄워주는 것이 좋다.

Module Level Dunder Names (Dunder란 double underbar)
============예시============
```
from __future__ import barray_as_FLUFL

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

import os
import sys
```
============예시============

String Quotes
작은따옴표(' ')와 큰따옴표(" ")는 모두 파이썬에서는 동일하다.
세따옴표(''' ''') 안에서는 큰따옴표를 사용하는 것이 더 적합하다.

문구 별 띄어쓰기
괄호 안에 있는 값은 굳이 띄어쓰기를 하지 않는다.
쉼표(,) 뒤의 괄호는 붙여쓴다.
쉼표의 앞, 세미콜론, 콜론의 앞은 붙여쓴다.
slice로 쓰이는 콜론은 앞뒤 모두 붙여쓰지만, 연산이 포함되었으면 띄어쓴다.
============예시============
```
# correct:
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+offset : upper+offset]

# incorrect:
spam( ham[ 1 ], { eggs: 2 } )
bar = (0, )
if x == 4 : print(x , y) ; x , y = y , x
ham[1: 9], ham[1 :9], ham[1:9 :3]
ham[lower + offset:upper + offset]
```
============예시============

메서드와 괄호는 붙여쓴다(띄어쓰기 X)
데이터를 인덱스로 부를 때는 붙여쓴다(띄어쓰기 X)

Other Recommendations
띄어쓰기는 연산자에 대해서는 띄워주는 편이 더 좋다. 그러나 곱셈이나 괄호 안 연산 등은 붙여씀으로써 하나의 덩어리 속에 묶여서 계산되어야 한다는 점을 강조하는 것이 더 좋다.

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

function annotation의 경우 기본 규정들을 지키는 것이 좋고, ->를 사용한다면 앞뒤로 띄어쓰기를 붙여주는 편이 좋다.

```
============예시============
def munge(input: AnyStr):
def munge() -> PosInt:
============예시============
```

함수에서 기본값을 설정해줄 때는 띄어쓰기를 없애는 편이 더 좋다.

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

When to use Trailing commas
trailing comma(맨 마지막 인덱스의 값 뒤에 붙는 쉼표)는 쓰든 안 쓰든 자유지만, 튜플에서 한 요소만 있도록 만든다면 이때는 필수적으로 있어야 한다. 여러 개의 값을 한 변수에 저장한다고 하면, 이때는 각 줄을 띄워주는 편이 더 좋다.

```
============예시============
FILES = ('steup.cfg',)
FILES = ['setup.cfg',
             'tox.ini',
             ]
============예시============
```

Comments
코드 내용와 모순되는 코멘트는 최악이기에 코드를 바꾸면 꼭 코멘트도 그때그때 바꿔야 한다.
코멘트는 완전한 문장의 형식을 갖추도록 하여 첫 글자는 대문자로 해야 한다. Block comments는 일반적으로 하나 이상의 문단으로 구성되어 완전한 문장의 형식을 지닌다. 코드의 마지막 줄이 아니라면 한 줄

을 띄워주는 것이 좋다. 코드에 대한 설명은 반드시 영어로 사용해야 한다.

Block Comments
내용 생략

Inline Comments
같은 줄에서 코멘트를 다는 것은 그렇게 권장되지 않는데, 특히 그 코드의 기능만 설명하고 의미(역할)를 설명하지 않는 코멘트는 굳이 안 쓰는 편이 좋다.

```
============예시============
x = x + 1                 # Increment x (X)
x = x + 1                 # Compensate for border
============예시============
```

documentation Strings
""" """ 형태의 문자열. 한 줄로 작성할 때는 따옴표가 한 줄에 오도록 작성하고, 여러 줄에 걸쳐서 작성할 때는 첫 줄은 """와 함께 문자가 오도록, 나머지 """는 줄을 따로 띄워서 쓰는 것이 바람직하다.

Naming Conventions
b (single lowercase letter)
B (single uppercase letter)
lowercase
lower_case_with_underscores
UPPERCASE
UPPER_CASE_WITH_UNDERSCORES
CapitalizedWords
mixedCase

_single_leading_underscore : weak 'internal use' indicator. from M import * does not import ojects whose names start with an underscore.

_single_trailing_underscore_ : used by convention to avoid conflicts with Python keyword

__double_leading_underscore : when naming a class attribute, invokes name mangling (inside class FooBar, __boo becomes __FooBar__boo; see below).
__double_leading_and_trailing_underscore__: “magic” objects or attributes that live in user-controlled namespaces. E.g. __init__, __import__ or __file__. Never invent such names; only use them as documented.

Names to avoid
l, O, I 등 다른 문자 또는 숫자 등과 헷갈릴 수 있는 것들은 사용하지 않는다.

Class Names
클래스들은 CapWords 관습을 따르는 것이 일반적이다.

Type Variable Names
입력 변수명은 CapWords를 사용하되 이름을 줄이는 것이 더 좋다.

============예시============
VT_co = TypeVar('VT_co', covariant=True)
KT_contra = TypeVar('KT_contra', contravariant=True)
============예시============

Exception Names
Exception도 클래스여야 하므로 class name 선정과 동일한 방식으로 이름을 작성하며, 에러의 마지막에는'Error'를 꼭 붙여줘야 한다(error가 정말 error일 때).

Global Variable Names

Function and Variable Names
Function 이름들은 소문자여야 하고, 단어들은 [_] 로 구분하여 가독성을 높인다. 기존의 가독성을 일관되게 유지한다는 가정하에 mixedCase도 상황에 따라 사용한다.

Function and Method Arguments
인스턴스 메서드의 첫 argument로 꼭 self를 써야 한다.
cls를 항상 class 메서드의 첫 argument로 써야 한다.
class_는 clss보다 나은 것이다.

Method Names and Instance Variables
생략

Public and Internal Interfaces
사용자를 public과 internal interfaces로 명확히 분리시킬 수 있어야 한다. 문서화된 interface는 public으로 간주되는 편이다. __all__ attribute을 사용하여 배타적으로 public API의 이름을 선언할 수 있어야 한다.

Programming Recommendations
코드는 파이썬의 변형된 언어들(PyPy, Jython, IronPython 등)로 사용하여도 문제가 없도록 작성되어야 한다.
문법적으로 is와 not을 함께 사용할 일이 있으면, 가독성과 의미상의 해석 등을 고려해서 not ... is가 아니라 is not으로 작성하는 편이 더 좋다.

============예시============
if foo is not None:
============예시============

예외 처리를 해줄 때, 가능하면 예외에 대한 정확한 문구를 작성해주는 것이 좋다.
============예시============

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

또한 try를 여러 차례 쓰는 것보다는 except와 else등을 복합적으로 활용해서 try 문이 너무 많이 쓰이는 문제를 개선하는 편이 좋다.

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

Ceontext manager는 분리된 function 또는 method로 있어야 한다.

============예시============
```
with conn.begin_transaction():
    do_stuff_in_transaction(conn)
```
============예시============

return문을 만들 때 일관된 형태로 구성하는 것이 좋다. 예를 들어 if 문을 활용하여 return을 한다고 하면, if문이 거짓일 때 else:를 활용해주는 것이 좋다.

============예시============
```
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)
```
============예시============

prefix와 suffix를 이용하여 자를 때는 split() 메서드나 슬라이싱([:-2] 등)을 활용하기보다는 ''.startswith()와 ''.endswith()를 활용하는 것이 더 좋다.

============예시============
```
if foo.startswith('bar'):
```
============예시============

type을 비교할 때도 가독성을 위해서 type(variable_name)보다는 isinstance(variable_name, object)를 활용하는 것이 더 바람직하다.

```
============예시============
if isinstance(obj, int): # O
if type(obj) is type(1): # X
============예시============
```

sequence 자료형(strings, lists, tuples)은 빈 자료형이 false를 반환한다는 점을 활용하여 코딩하는 것이 len() 메서드를 활용하는 것보다 더 바람직하다.

```
============예시============
if not seq:
if seq:
============예시============
```

boolean value는 == 를 활용하여 비교하지 말고 직접 비교하는 것이 더 좋다.

```
============예시============
if greeting: # O
if gretting == True: # X
if greeting is True: # X
============예시============
```

Function Annotations
내용 생략

Variable Annotations
내용 생략

코드를 쓰는 것보다 코드를 읽는 작업을 많이 하기 때문에 일관성 있게 가독성이 높은 코드를 짜는 것이 중요하다.

코드 레이아웃과 관련된 스타일 가이드로는 다음과 같이 있다.

1) 기본 indentation level은 4 spaces이다. 배열의 closing bracket은 저는 개인적으로 두 번째(lined up under the first character of the line that starts the multiline construct)가 편해서 이걸 쓰려고 합니다.

2) 한 줄에는 최대 79개의 문자가 와야합니다. 저도 코드가 옆으로 너무 길면 가독성이 떨어진다는 것을 많이 인지해서 저도 최대한 길 경우, multiline으로 줄이려고 노력합니다. 아래로 내릴 때는 backslash '\'을 붙여야 합니다.

3) binary operator는 줄의 끝이 아닌 시작에 오도록 하는 것이 가독성이 좋다고 합니다. 그 전에는 저도 줄로 나눌 때 끝으로 가게 해야할 지 아니면 앞으로 오게 해야할 지 잘 몰라 헷갈리는 경우가 있었는데, 꼭 지켜야할 필요까지는 없지만 그래도 binary operator를 앞으로 오게끔 하는 것이 좋다는 것은 기

억해 놓으려고 합니다.

4) 빈줄의 존재 여부 역시 가독성에 큰 영향을 줍니다. 그냥 빈줄 없이 빼곡히 적는 것보다는 당연히 함수가 끝나고 아니면 클래스 정의가 끝나고 빈줄을 넣는 것이 읽기에 당연히 편하겠죠? 클래스 정의에는 2개의 빈줄을, 클래스 내부 함수 정의는 1개의 빈줄을 쓰는 것이 원칙이라고 하네요.

5) import도 어떤 규칙에 따라 그룹화 및 작성 순서를 정해야 합니다.

빈줄과 마찬가지로 공백 역시 가독성에 영향을 미친다. 나는 가끔씩 예를 들어, x = x*2 + 1을 x = x * 2 + 1이라고 가끔씩 쓰기도 한다. 엄청 짧은 식이기 때문에 가독성에 큰 문제가 없을 것이라고 생각했기 때문에 큰 신경을 쓰지 않았다. 하지만 이렇게 쓰는 것을 추천하지 않는 것을 이번에 처음 알게 되었고, 앞으로도 유의하여 코드를 작성해보고자 한다. 그리고 또한 그 동안 c++로 코드를 짤 때 항상 나는 여러 줄을 한 줄로 작성한다. 예를 들어, do_one(); do_two(); do_three()와 같이. 왜냐하면 코드가 아래로 너무 길어지는 것이 개인적으로 불편했기 때문에 짧은 것이나 연관성이 있는 것은 최대한 한 줄로 넣어 작성해왔다. 그러나, 이것이 가독성이 좋지 않다는 것을 이번에 깨달았고, 팀프로젝트 할 때만큼은 최대한 감안하여 작성해보고자 한다.

코멘트는 상황에 따라 block comments, inline comments, docstrings를 적절히 활용해야 한다.

변수, 함수, 클래스 등에 이름을 지정할 때 이름을 적절히 지어야 한다. 당연히 변수, 함수, 클래스의 역할에 따라 이름을 부여하는 것은 당연하며 lowercase, uppercase 및 underscore의 적절한 활용이 필요하다.

Algorithm Assignment Report

과제 완성 간 주요 활동 사항

2024.01.14. Meeting

대면회의 (15:00~17:00)

1. PEP8 문서 및 BPE 관련 문서 정독.
2. 문서 정독 후 알고리즘 완성 과정 논의.
3. Algorithm assignment 관련 역할 분담.
4. 1월 15일 대면 회의 전까지의 과제 확인

역할 분담

이성현 : BPE 논문 끝까지 읽어오기.
코드 작성 스타일 관련 글 소감문 완성.
preprocessor 구현.
보고서 간단히 작성.

박경욱 : parent class 구현(tokenizer).
decorator 추가
상황 판단 후 추가적인 역할 부담.

전체 알고리즘 작성 과정

Whole sequence Preprocessor / Tokenizer / Main.py / Word / BPE / (With Error Dealing) -> after these : Adding decorator and enhancing readability.
독립적으로 진행 가능한 경우 별개로 활용

2024.01.15. Meeting

대면회의 (19:30~21:00)

1. BPE tokenizer 관련 논문 강독 결과 공유.
2. 논문 내용을 바탕으로 한 과제 진행 간 핵심 과업 확인.
3. Tokenizer.py와 Tokenizers.py 파일의 분리로, 알고리즘 구현의 편의 증진.

역할 분담

이성현 : 보고서 작성 BPE
Word tokenizer 완성 (preprocessor를 거의 그대로 차용하는 방안 고려)
Preprocssor 완성 (for문 써서 white space 제거)

박경욱 : Tokenizer.py와 Tokenizers.py 파일 완성
main.py 파일을 고려하여 알고리즘 일부 수정

2024.01.16. Meeting

Zoom meeting (23:00~23:20)

1월 15일 당시 맡은 역할 분담 결과 확인.
개별 일정으로 인해 진행도가 낮았던 점 확인 후 추후 일정 공유
알고리즘 작성 관련한 진행 상황 공유
edge case와 pipeline 상 오류 없는 파일 간 연결을 위해 고려할 점 논의

역할 분담

이성현 : preprocessor.py 최종 완성 및 보고서/알고리즘 정리

박경욱 : tokenizer.py 파일 수정 후 공유.

2024.01.18.

최종 파일 제출