과제 1 (김대솔 김소민)

Introduction

- 가이드라인에 집착x 가이드라인의 목적인 코드의 가독성/이해도를 최우선시
- 호환성 주의

Code Lay-out

1. Indentation (=4 spaces/ tab과 다름 섞이면 안됨)

-vertical for distinguishing arguments

Cf) if – 주석,이어지는 줄,다중구조 등

2. Maximum Line Length

- 한 줄 최대 글자: 79자
- 주석 등은 72자까지
- 문장이 길때는?: (), \

3. Should a Line Break Before or After a Binary Operator?

NO

4. Blank Lines

- 2: 최상위 함수, def문
- 1: 클래스 내부 메소드의 def문
- 추가: 연관된 함수 구분, 함수 내에 논리적인 구분
- 생략: 더미 코드 사이

5. Source File Encoding

- UTF-8 NOT 인코딩 선언
- standard library MUST use ASCII-only identifiers
- only English

6. Imports

- 각각 한줄씩, 사이에 공백 한줄씩

- 표준 -> 연관된 third party -> local/specific

- if 패키지 내부 디렉토리가 sys.path안에 있는 경우에는 from mypkg import형태로

(너무 길면 from. Import로)

-모듈에서 클래스를 불러올 때는: from myclass import MyClass, from foo.bar.yourclass import YourClass

- 로컬 이름과 충돌시: from myclass 후에 **"myclass.MyClass"로**

7. Module Level Dunder Names

from __future__ import barry_as_FLUFL

__all__ = ['a', 'b', 'c']

__version__ = '0.1'

__author__ = 'Cardinal Biggles'

import os

import sys

8. String Quotes: ‘,”: 구분x

Whitespace in Expressions and Statements

-{,{,;,함수 변수 전달(spam (1) ->x)안에서 공백x

-,사이 공백,x

-인덱스, 슬라이싱 x

- 맨뒤에 넣지 말기

- +-뒤에는 한칸씩

- 명령문은 한줄씩(한줄에 몰아넣지 말기)

When to Use Trailing Commas

FILES = ('setup.cfg',)

```
FILES = [

    'setup.cfg',

    'tox.ini',

    ]
```

Comments

-완전한 문장, 첫번째는 대문자

-코드와 동일한 들여쓰기

- 코드와 같은 줄에서는 어지간하면 쓰지 말기

-documentation strings: 모든 클레스,메소드,모듈,함수에서 쓰기 """ 여러줄로

Naming Conventions

라이브러리의 네이밍의 "일관성"을 유지하기 위한 권장표준(기준), 새로운 모듈에 한해서 적용되며 기존의 라이브러리가 있다면 기존의 내부 일관성이 우선시된다

1. **Overriding Principle**

API의 공개적인 부분으로 사용자에게 보이는 이름은 구현보다는 사용법을 반영하는 규칙을 따름

2. **Descriptive: Naming Styles**

일반적으로 네이밍 되는 스타일

- b (단일 소문자)
- B (단일 대문자)
- lowercase
- lower_case_with_underscores
- UPPERCASE
- UPPER_CASE_WITH_UNDERSCORES
- CapitalizedWords (또는 CapWords, CamelCase - 이는 그 글자의 특징적인 모양 때문에 붙여진 이름, 또는 StudlyCaps로도 알려져 있음)

**CapWords에서 약어를 사용할 때는 약어의 모든 글자를 대문자로 써야 함. 그래서 HTTPServerError가 HttpServerError보다 더 좋음.*

- mixedCase (첫 글자가 소문자, CapitalizedWords와 다름)
- Capitalized_Words_With_Underscores (이상함, 거의 안쓰임)

그룹화를 위한 고유 접두어를 사용하는 스타일

(Ex. Os.stat() 함수 -> st_mode / st_size 등 이름을 가진 튜플로 반환 가능)

선행 또는 후행 밑줄을 사용하는 다음과 같은 특수 스타일

- _single_leading_underscore: 약한 "내부 사용" 지시자. (Ex. from M import -> 밑줄로 시작하는 이름의 객체를 가져오지 않음)
- single_trailing_underscore_: Python 키워드와의 충돌을 피하기 위해 사용됨. (Ex. tkinter.Toplevel(master, class_='ClassName')
- __double_leading_underscore: 클래스 속성을 네이밍할 때 이름을 변경 (FooBar 클래스 내부에서, __boo는 _FooBar__boo가 됨)
- double_leading_and_trailing_underscore: 사용자가 제어하는 네임스페이스에서 존재하는 "마법" 객체 또는 속성. (Ex. init, import 또는 file. 이런 이름을 새로 만들어서는 안 되며, 문서에 명시된 대로만 사용해야 함)

3. Prescriptive: Naming Conventions

(1) Names to Avoid

변수 이름으로 'l' (소문자 엘), 'O' (대문자 오), 'I' (대문자 아이) 사용하지 않기 -> 구별이 안감

lIili oO0o)Oo ㅇ 처럼 이게 뭐가 뭔지 구별이 아예안감

(2) ASCII Compatibility

표준 라이브러리에서 사용되는 식별자들은 PEP 3131의 정책 섹션에 설명된 대로 ASCII 호환성을 가져야 함 -> 이식성과 가독성을 높이기 위해

(3) Package and Module Names

모듈들은 짧고, 모두 소문자로 이루어진 이름이여야 함, 가독성을 향상시키기 위해 밑줄 사용을 사용해도 된다, 하지만 파이썬은 밑줄 사용지향, C 또는 C++로 작성된 확장 모듈은 파이썬에서 밑줄이 붙음 (예: _socket).

(4) Class Names

일반적으로 클래스 이름은 CapWords 규칙을 사용, 인터페이스가 문서화되고 주로 호출 가능한 것으로 사용되는 경우, 함수의 네이밍 규칙을 대신 사용해도 됨.

**주의 대부분의 내장 이름은 단일 단어(또는 두 단어를 연결한 것)이고, CapWords 규칙은 예외 이름과 내장 상수에만 사용됨.*

(5) Type Variable Names

PEP 484에서 도입된 타입 변수의 이름은 일반적으로 CapWords를 사용하되, 짧은 이름을 선호

또한, T, AnyStr, Num. 공변 혹은 반공변 행동을 선언하는 데 사용되는 변수에 _co 또는 _contra 접미사를 추가하는 것이 권장

(6) Exception Names

예외 클래스를 정의할 때는 알맞은 이름을 선택하는 것이 중요 "Error" 접미사를 사용하면 그 예외가 실제로 오류를 나타내는 것임을 명확하게 표현할 수 있음 -> 가독성 UP

(7) Global Variable Names

글로벌 변수는 함수처럼 소문자와 밑줄을 통해 네이밍해야함 하지만, 이 변수들은 하나의 모듈에서만 사용되어야 함, 모듈을 'from M import *' 형태로 사용하려면, 전역 변수를 내보내지 않기 위해 __all__을 사용하거나, 변수 이름 앞에 밑줄을 추가요망.

(8) Function and Variable Names

함수와 변수 이름은 모두 소문자를 사용하고, 단어 사이는 밑줄로 구분, mixedCase는 특정 경우에만 허용 (호환성을 유지하기 위해)

(9) Function and Method Arguments

인스턴스 메소드는 첫 번째 인자로 'self'를, 클래스 메소드는 'cls'를 사용. 이름이 예약어와 겹칠 때는 이름 뒤에 밑줄을 추가.

(10) Method Names and Instance Variables

메소드와 인스턴스 변수의 이름은 소문자와 밑줄을 사용. 비공개 항목에는 선행 밑줄을 하나, 하위 클래스와 이름 충돌을 피하기 위해선 두 개의 선행 밑줄을 사용. 이중 선행 밑줄은 이름 충돌을 피하기 위한 경우에만 사용.

(11) Constants

상수는 대문자와 밑줄을 사용하여 표현하며, 모듈 전체에서 일관되게 적용

(12) Designing for Inheritance

클래스를 설계할 때, 메소드와 인스턴스 변수(속성)가 공개될지 아닐지 결정해야 함

공개 속성: 클래스를 사용하는 다른 클라이언트가 사용하도록 의도된 속성. 이에 대해 후방 호환성을 유지할 책임이 있음. 공개 속성은 밑줄 없이 정의.

비공개 속성: 다른 사용자가 사용하도록 의도되지 않은 속성, 변경되거나 제거될 수 있음.

하위 클래스 API의 일부인 속성: 확장하거나 수정하기 위해 상속될 수 있음. 이때 어떤 속성이 공개되고, 어떤 것이 하위 클래스 API의 일부인지, 어떤 것이 기본 클래스에서만 사용되는지 구분만 잘 하면 됨

몇가지 가이드 라인으로

밑줄 시작 X

공개 속성 이름이 충돌되면 밑줄 추가

사용하지 않을 속성은 두개의 밑줄

4. Public and Internal Interfaces

하위 호환성 보장은 공개 인터페이스에만 적용, 사용자가 공개 인터페이스와 내부 인터페이스를 명확하게 구분해야함.

문서화된 인터페이스는, 공개 인터페이스로, 문서화되지 않은 인터페이스는 모두 내부 인터페이스로 간주, 모듈은 all 속성을 사용해 공개 API의 이름을 명시적으로 선언해야 함. (Ex. __all__을 빈 리스트로 설정하면 공개 API가 없다는 것을 의미)

Programming Recommendations

Python의 다양한 구현체에 작용을 주지 않도록 코드를 작성하며, 문자열 연결에는 ''.join()을 사용

싱글톤과의 비교는 항상 'is'나 'is not'으로 수행하라며, 코드에서 'not ... is' 보다는 'is not' 연산자가 표현이 좋음

비교를 통한 순서 지정 구현 시 모든 비교 작업을 구현하라며, 데코레이터는 누락된 비교 메서드를 생성하는데 도움을 줌

Python 시스템은 반사성 규칙을 가정하므로, 이를 이용해서 여섯 가지 작업을 모두 구현하라는 것이 좋음

기능 정의에는 'def'를 사용하며, 람다 표현식이 아닌 할당문을 사용하라는 것이 좋음

BaseException에서 직접 상속받는 것보다는 Exception에서 예외를 파생. BaseException에서 직접 상속받는 것은 거의 항상 잘못된 방법

예외를 발생시키는 위치보다는 예외를 잡을 코드가 필요로 하는 구별에 기반한 예외 계층을 설계하는게 포인트. "문제가 발생했다"라고만 말하는 대신에 "어떤 문제가 발생했는가?"를 프로그래밍적으로 대답하려는 목표를 가져야함

모든 return 문에서 일관성을 유지해야함

Function Annotations

함수 주석은 PEP 484 문법을 사용해야 함

Python 표준 라이브러리는 이러한 주석을 채택하는 데 있어 보수적이어야 하지만, 새로운 코드와 큰 리팩토링에는 그 사용이 허용

타입 검사기는 선택적인 독립 도구로, 기본적으로 Python 인터프리터는 타입 검사로 인해 메시지를 발행하거나 주석에 기반한 동작을 변경해서는 안 됨

Variable Annotations

모듈 수준 변수, 클래스와 인스턴스 변수, 그리고 로컬 변수에 대한 주석은 콜론(:) 뒤에 공백 한 칸을 가져야 함

콜론 앞에는 공백이 없어야 함

할당이 오른쪽 수식을 가지는 경우, 등호(=)는 양쪽에 정확히 공백 한 칸을 가져야 함

소감: 생각보다 헌법보다 더 세세하게 규제되어있는 것 같아서 놀라웠다, 확실히 매우 광범위한 내용들을 규칙과 규율을 만들려고 하니 이 모든 것을 외우는 것 보단 센스로 항시 지켜야 하는게 좋다고 판단되었다, 또한 지금까지 프로그래밍 작업을 하면서, 이런 규칙들을 많이 무시하고 왔다, 코드가 인풋을 받아서 아웃풋을 제대로 출력만 하면 됐다는 생각에 사로잡혀 있었기 때문이다. 하지만 이번 기회에 "모든 프로세스를 쉽고 가독성 좋게 이해시키는 것" 또한 매우 중요하다는 사실을 깨달았다, 마지막으로 협업 상황에서 코드의 가독성이 더욱 중요하다는 것을 알게 되었다, 각자의 생각과 로직을 코드에 담는 것이지만, 그 코드를 다른 사람도 이해할 수 있도록 작성하는 것이 중요하다는 것을 깨달았다.

소감: 지금가지 아무 생각없이 했던 부분들이 내 생각 이상으로 정리가 안되있고 보기 힘들었다는 생각이 들었다. 예전에 작성했던 코드들이 대체로 해당 규칙에 어긋나는 것을 보고, 지금까지 많은 사람들이 읽기 힘들었을 거라는 생각에 미안해졌다. 또한 예전에 프로젝트 할 때, 단순히 주석을 덧붙이는 것만으로는 다른 사람들이 이해를 못해서 일일이 한줄씩 설명해야 했던 일이 생각나며 해당 규칙을 활용하여 조금 더 협업에 용이한 코드를 짜야겠다는 생각이 들었다.

과제 2 보고서

클래스 디자인 고려사항

명확하게 BPETokenizer 클래스를 정의함으로 확장성과 각 메소드의 역할을 나누었음으로 유지보수를 쉽게 할 수 있었다.

독립적인 기능을 가진 메소드로 분리하여 모듈화를 강화함으로 코드의 가독성과 재사용성을 높였습다.

Python의 typing 모듈을 사용하여 각 메소드의 매개변수와 반환 값에 타입 힌트를 제공함으로써, 누구나 코드를 더 쉽게 이해하고 사용할 수 있도록 했다.

에러를 방지하기 위하여 조건문을 활봘하게 이용했다

각 메소드의 구조와 작동 원리

(1) __init__:

클래스의 생성자로, 선택적으로 코퍼스와 어휘 크기를 매개변수로 받았다 (*어휘 크기를 제한한 이유는 너무 커버리면 학습시간이 무한 LOOP이 됨으로 적절하게 제한하였다). vocab은 단어의 빈도를 저장하는 데 사용되며, bpe_codes는 병합된 문자 쌍과 그 순서를 저장했다. 제공된 코퍼스가 있는 경우, add_corpus 메소드를 호출하여 어휘를 초기화하도록 하였다.

(2) add_corpus, add_line_to_vocab:

코퍼스를 단어 단위로 분할하고, 각 단어를 어휘에 추가하는 역할

단어는 문자 단위로 분해되고, 단어의 끝을 나타내는 </w> 토큰이 추가 되도록 설계.

(3) get_stats:

어휘 내 모든 단어들을 순회하면서 연속적인 문자 쌍의 빈도수를 계산.

(4) merge_vocab:

가장 빈번한 문자 쌍을 실제로 어휘에서 병합

정규 표현식을 사용하여 병합을 수행하고, 어휘를 업데이트

(5) train:

지정된 반복 횟수만큼 또는 어휘 크기에 도달할 때까지 가장 흔한 문자 쌍을 병합

각 병합 과정에서 update_vocab_token_index를 호출하여 토큰과 인덱스 매핑을 갱신.

(6) tokenize:

입력된 텍스트를 토큰화. 텍스트는 문자 단위로 분해되고, 학습된 어휘를 바탕으로 토큰의 인덱스로 변환. 또한 padding과 max_length 옵션을 사용하여 토큰화된 결과의 길이를 조정할 수 있도록 했음

(7) __call__:

클래스 인스턴스가 함수처럼 호출될 수 있도록 하여, tokenize 메소드를 간편하게 사용할 수 있음

제시된 조건 충족 방법

typing을 사용하여 입력 타입에 대한 명확한 지정과 에러 처리를 강화

코드의 가독성을 높이기 위해 메소드를 명확하게 구분하고, 주석을 통해 설명을 추가.

상속 가능, BPE 알고리즘을 구현한 베이스 클래스로 사용가능.

데코레이터나 functools는 이 코드 예제에서 직접적으로 사용되지 않았습니다.. .. (감점 요인..? ㅎ)

내장 라이브러리만 사용하여 외부 의존성을 아예 제거

협업한 방식

과제를 시작하기 앞서 서로 내용도 모르면 협업이 안되니 약 3일의 시간동안 여유롭게 과제 1,2의 링크를 정독후 과제 1은 앞뒤로 파트를 나눠서 요약을 했으며, 과제2의 코딩은 서로 막히는 부분에 있어서 토의하고 서로의 궁금증을 해결했습니다.