Report

과제 1

소감

박정양:

이러한 문서가 나온 이유 자체가 결국 복잡한 코드를 짤 때는 많은 사람들이 함께 코드를 작성하게 되기 때문이라고 생각하였습니다. 결국에 코딩을 배우는 이유가 복잡한 코드를 작성하기 위함이고, 그 코드를 작성하기 위해 여러 사람이 협업을 진행하게 됩니다. 이 때 모든 사람이 코드를 작성하는 방법이 다르고, 코드의 가독성이 좋지 않다면 협업의 전체적인 효율성이 떨어지게 될 것입니다. 따라서 처음 배울 때부터 코드를 가독성 좋게 구현하는 습관을 만

들어야 나중에 복잡한 코드를 작성할 때도 가독성이 좋은 코드를 작성할 수 있음을 깨달았습니다.

또한 가독성이 좋은 코드의 장점으로는 유지보수를 진행할 때 효율성이 좋다는 것입니다. 프로젝트를 진행한 뒤 여러가지 이유로 코드를 수정해야 하는 상황이 발생할 수 있는데, 이 때 가이드라인을 따르지 않아서 가독성이 나쁜 코드의 경우 가독성이 좋은 코드보다 어떤 부분에서 문제가 생긴것인지 확인하는 것이 어렵습니다. 따라서 유지보수를 할 때 더 많은 비용이 들게 됩니다.

다만, 협업으로 진행될 때는 해당 프로젝트의 코딩 스타일을 먼저 따라가라는 말도 있었고, 가이드라인을 따르기 어려울 때 다른 방법을 이용하라는 말도 있었습니다. 그만큼 완전한 정답은 없지만, 코딩을 하는 목적은 결국 다른 사람들과 공유하는 코드를 작성하기 위함이기 때문에 이러한 가이드라인이 존재한다는 것을 깨달았습니다.

김채현:

과제 이전에는 파이썬 공식 문서를 정독해본 적이 없었는데, 이번 기회로 협업을 할 때 지켜야 할 파이썬 규칙과 같은 부분을 알게 되었습니다. 예를 들어 코드가 복잡해지고, 여러 사람이 공동으로 작업을 하는 상황에서 변수명은 어떻게 지정해야 하는지, 그리고 어떤 코딩 스타일을 지켜야 하는지 알 수 있었습니다. 적절한 규칙을 따르면 여러 사람이 함께 복잡한 작업을 수행하더라도 유지보수를 하는 시간을 효율적으로 줄일 수 있고, 더 가독성 좋은 코드를 만들 수 있을 것 같습니다. 향후 와이빅타에서 여러 프로젝트를 할 때 지금 배웠던 내용이 매우 유용하게 쓰일 것 같습니다.

명시된 모든 규칙을 외우기는 힘들겠지만, 이러한 규칙이 있구나를 인지함으로써 무조건 결과만 도출되는 코딩이 아닌 효율적으로 가독성 있게 코드를 짜는 과정도 중요하다는 것을 알 수 있었습니다. 그리고 파이썬과 관련된 모르는 부분을 공식 문서를 정독하면서 읽는 방법도 매우 유용하다는 것을 알게 되었씁니다.

Link

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

Python Enhancement Proposals (PEPs)

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

Intro

- 많은 프로젝트는 각자의 코딩 스타일이 존재한다
    - 프로젝트의 스타일과 이 document의 스타일에 conflict가 존재할 때는 프로젝트의 스타일이 우선이다.
    - 즉 이 가이드라인에 너무 빠지지 말고 유연성을 가지라는 뜻

A foolish consistency is the Hobgoblin of Little minds

- 사람들이 고집부리거나 변화를 거부하는 습관을 비판하는 말
    - 과거의 생각에 고수하는 것이 얼마나 좁은 시야를 만들어내는지를 강조
- 이 가이드라인을 따를 수 있을 때는 따르되, 외에도 좋은 방법이 있거나 쓰기 어려운 경우 최선의 선택을 하라
    - 위에서 했던 말과 일맥상통

Code Lay-out

요약

- indentation 별로 4개의 Space (탭보다는 space)
- 한 줄의 코드가 너무 길어서 여러 줄에 나타내야 할 때
    - 너무 길다의 기준
        - 코드는 79자, docstring과 comment는 72자
        - 합의하에 코드는 99자로 늘릴 수 있음
    - 코드 작성 방법
        - Hanging indents
            - 첫 번째 줄에는 괄호 열기만 하고, 그 뒤로 4개의 space로 구별하여 argument 작성

        - \(백슬래시)
            - 줄의 끝부분에 백슬래시를 입력
        - 괄호가 존재할 경우 괄호 여는 부분에서 다음 줄이 시작하게 할 수도 있음
    - 사칙연산 기호
        - 사칙연산 기호가 맨 앞으로 오게 자르기
- 빈 줄
    - Top-level function과 class definitions은 2줄의 빈 줄을 준다
    - Class 안의 메서드들 사이에 1줄의 빈 줄을 준다
    - 서로 관련된 함수들 그룹을 구별해주기 위해 1줄의 빈 줄을 준다
    - 자동으로 빈 줄을 만들어주는 단축키로 control-L이용
        - editor별로 다를 수는 있는데, 보통 이걸 쓴다고 함
- import
    - 한줄에 import는 한개만
        - 안좋은 형식: import os, sys
        - 단, from library import ___, ___, ___ 이런 형식은 좋음
    - 파일의 가장 위쪽에 module comments and docstrings 직후 선언
- 기타
    - ASCII가 아닌 문자는 최대한 피하기
    - dunder name
        - 이건 아래쪽을 봐야할 것 같음

Indentation

- indentation level 별로 4개의 space 이용
    - for, while 과 같이 안에 들어가는 것 의미
- Continuation lines
    - 원래 한줄로 써야하는 코드인데, 너무 길어서 여러 줄로 작성하는 경우
    - 2가지 방법
        - 괄호 여는 부분에서 다음 줄이 시작하게 하기

```
foo = long_function_name(var_one, var_two,
                         var_three, var_four)
```

- Hanging indents
  - 첫 줄만 처음에서 시작, 그 뒤로는 4개(이상)의 space로 구별
  - 이 때, 첫번째 줄에는 argument가 없어야 하며, 다음에 오는 줄과 헷갈려서는 안됨

```
# Add 4 spaces (an extra level of indentation) to
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)
# 2번째, 3번째 줄이 한칸 앞으로 오면 명령어와 위치가 헷갈릴

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

- 닫는 괄호
  - 선택

```


# 1번 옵션
my_list = [
    1, 2, 3,
    4, 5, 6,
    ]

# 2번 옵션
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
)
```

Tabs or Spaces?

- Indentation 기법으로 Spaces를 선호
- Space와 Tab을 번갈아 쓰는 것만 피하기

Maximum line length

- 제한 사항(Python standard library)
  - 79자로 제한
  - docstrings과 comments처럼 구조적 제한이 적은 block은 72자로 제한
- 제한하는 이유
  - 여러개의 파일을 동시에 띄워두고 수정이 용이
  - 코드 리뷰 툴을 이용할 때 편함
- 팀에 따라 더 긴 코드를 선호할 수 있음
  - 이럴 경우 코드 길이 제한은 99자로 증가시켜서 쓸 수 있음
  - 단, 여전히 docstrings과 comments는 72자
- 제한보다 긴 코드
  - 괄호를 이용하여 여러줄로 나타냄
  - \ (백슬래시)를 이용해서 여러줄로 나누어 쓸 수 있음

Should a line break before or after a binary operator?

- Binary operator
  - +, -, *, /
  - 사칙연산을 나타내는 연산자들을 말함
- Binary operator에서 줄을 나눠야할 때는 Binary operator 앞에서 자른다.
  - 즉, 새로운 줄을 Binary operator로 시작한다.
  - 이유

        - Binary operator가 세로방향으로 정리되어 있어 가독성이 좋음
        - Binary operator가 적용되는 operand가 무엇인지 확인이 쉬워 가독성이 좋음

Blank lines

- Top-level function과 class definitions은 2줄의 빈 줄을 준다
- Class 안의 메서드들 사이에 1줄의 빈 줄을 준다
- 서로 관련된 함수들 그룹을 구별해주기 위해 1줄의 빈 줄을 준다
- control-L (i.e. ^L)
    - 자동으로 빈 줄을 만들어주는 단축키
    - 많은 editor에서 이 단축키를 이용함

Source file encoding

- 파이썬 코드 배포판은 UTF-8만 쓰고 인코딩 선언이 따로 있어서는 안됨
- ASCII가 아닌 문자는 최대한 적게 사용
    - 가급적이면 장소, 사람 이름만
- 파이썬 표준 라이브러리의 식별자는 ASCII 전용 식별자
    - 가능하면 영단어 사용

Imports

- import 하는 방법
    - 한줄에 1개씩만 import
    - from ____ import ____, _____, ___ 이런 형식은 가능
- import 하는 위치
    - 파일의 가장 위쪽 부분
    - module comments and docstrings 직후
    - module globals and constants 전

Module level dunder names

- dunder (Double under의 약자)
  - 앞뒤에 __(언더바 2개)가 붙으면 던더 메소드
    - 클래스 안에서 __init__()으로 가장 많이 봤을 듯
  - 변수 앞에만 붙으면(이게 dunder name…?) 실수로 수정되지 않도록 name mangling 이 진행된다고 함
- import 하기 전에 선언
  - 단, from __future__ imports __ 문은 예외적으로 dunder 앞으로
    - (__future__ 이라는 라이브러리가 존재함)
  - 따라서 이런 형식

```
"""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

- 문자열을 나타내는 작은따옴표(’ ‘)와 큰따옴표(” “)는 파이썬에서는 똑같음
- 둘 중 하나로 고정하는 규칙은 없으나, 둘을 동시에 쓰지는 말 것
  - 단, 문자열 안에 작은따옴표 또는 큰따옴표가 있으면, 없는 것으로 감싸기

        - 아니면 백슬래시를 써야하는데, 가독성을 해침

Whitespace in expressions and statements

Pet peeves

- 띄어쓰기 규칙
    - 괄호 바로 앞 뒤에 띄어쓰기 하지 않기
        - **Correct**: (egg = {1})
        - **Wrong**: (egg = { 1 })
    - 콤마와 닫는 괄호 사이에는 띄어쓰기 하지 않기
        - **Correct**: foo = (0,)
        - **Wrong**: foo = (0,)
    - 콤마, 세미콜론, 콜론 바로 앞에는 띄어쓰기 하지 않기
    - slice에서 쓰이는 콜론의 경우, binary operator 처럼 양쪽에 같은 양의 띄어쓰기 필요
        - `ham[1:9]` 또는 `ham[1 : 9]`
        - 단, slice parameter가 생략된 경우 띄어쓰기도 생략
            - `ham[:: 9]` 가능
    - 소괄호, 대괄호 열기 전 띄어쓰기 하지 않기
        - **Wrong**: `spam (1)` || `dct['key'] = lst[index]`
    - operator 앞뒤로 2개 이상의 띄어쓰기 하지 않기
        - **Wrong**: x = 1

Other recommendations

- 줄 끝날 때 띄어쓰기 하지 말기
    - 잘 안보이는데, 결과적으로 에러를 만들 수도 있기 때문
        - 예) 줄바꿈 표시인 백슬래시 뒤에 띄어쓰기 있으면 줄바꿈 표시로 인식 안됨

- 연산자들 앞뒤로 항상 띄어쓰기 하기
  - assignment (`=`)
  - augmented assignment (`+=`, `-=` etc.),
  - comparisons (`==`, `<`, `>`, `!=`, `<>`, `<=`, `>=`, `in`, `not in`, `is`, `is not`),
  - Booleans (`and`, `or`, `not`)
- 우선순위가 다른 연산자를 함께 이용할 경우 낮은 우선순위의 연산자 앞뒤로 띄어쓰기 하기
  - 가독성을 위해 각자 판단할 것
  - 단, 연산자 앞과 뒤에는 같은 양의 띄어쓰기를 이용해야 함
- Function annotations
  - 파이썬은 자료형 선언이 없기 때문에 해석이 어렵다는 단점을 해결하는 것
    - 함수의 매개변수에 :expression 의 형태로 매개변수마다 annotation 작성 가능
    - function 옆에 ->expression 의 형태로 function의 return 값 작성 가능
  - 콜론에 대해서는 콜론 앞에는 띄어쓰기 안하고 뒤에 한개 두는 것
  - 화살표의 경우 양쪽에 띄어쓰기

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

- =(등호)가 keyword argument 또는 unannotated function parameter의 default value 할당을 위해 쓰였다면 띄어쓰기 하지 않기
  - annotated function parameter에 쓰였을 경우 등호 양쪽에 띄어쓰기
- 여러개의 줄에 거쳐서 쓰여져야 하는 코드를 한줄에 쓰는 것은 비추천
  - wrong example

```
if foo == 'blah': do_blah_thing()
do_one(); do_two(); do_three()
```

When to use trailing commas

- Trailing commas는 보통 optional
  - 1개의 element로 된 tuple을 생성할 때는 필수
- 추후에 list of value, argument, imported item이 추가될 수 있을 때 유용하게 이용
  - 이 경우 각 줄에 한개의 element를 두어야 하고, 닫는 괄호는 따로 써줘야 함

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

Comments

- 코드와 맞지 않는 주석은 없는것보다 안좋음
  - 코드가 바뀔 때마다 주석도 업데이트 해주기
- 주석은 문장형태여야 함
  - 소문자로 시작하는 identifier가 아닌 이상 대문자로 시작
- Block comment는 보통 완전한 문장으로 구성된 paragraph로, 각 문장은 마침표로 끝나야 함
  - 여러 문장으로 구성될 경우, 각 문장 사이에 1~2개의 띄어쓰기로 구별해줘야 함
- 이 주석을 읽을 사람들이 이해하기 쉽게 쓸 것
  - 영어로.
    - ~~(코드를 읽는 사람이 무조건 우리 언어를 쓸 수 있을 경우가 아니면)~~

Block comments

- 이 주석 뒤에 오는 코드들을 설명
- #(샵) 뒤에 띄어쓰기 한번 하고 그 뒤로 내용 작성
- block 안에 paragraph를 나눠야 하는 경우

- #만 있는 한줄을 두면 됨

```
# 안녕하세요
#
# 와빅 24기입니다.
```

Inline comments

- Inline comment
  - Statement와 같은 줄에 작성하는 comment
- 최대한 안쓰기
  - 당연한 내용은 쓰지 말고, 만약 어떤 의미인지 알 필요가 있는 코드일 때만 작성하기

```
x = x + 1                 # Increment x -> Bad

x = x + 1                 # Compensate for border -> Ne
```

Documentation strings

- Public modules, functions, classes, and methods에 대한 docstrings은 모두 작성하는 것이 좋음
  - non-public methods에 대해서는 안써도 됨
  - 단, 어떤 method인지에 대한 주석인 필요
- 함수를 define하는 줄 다음에 작성한다
- 한줄이든 여러줄이든 “”” 큰따옴표 3개로 둘러싼다. “””
  - 여러줄일 경우 뒤쪽 큰따옴표 3개는 줄 넘김을 해줘야 한다.

Naming Conventions

Overriding Principle

- api의 공개 부분으로 user에게 보여지는 이름은 구현보다는 사용법을 반영하는 규칙을 따라야 한다.

Descriptive: Naming Styles

- 어떤 Naming Style이 사용되고 있는지 알아볼 수 있도록 도와준다.
- 일반적으로 다음과 같은 Naming Style이 있다.
  - b (single lowercase letter)
  - B (single uppercase letter) lowercase
  - lower_case_with_underscores UPPERCASE
  - UPPER_CASE_WITH_UNDERSCORES
  - CapitalizedWords
  - mixedCase
  - Capitalized_Words_With_Underscores

또한 선행 또는 후행 밑줄을 사용하는 다음과 같은 특수 형식이 인식된다.

- _single_leading_underscore: weak “internal use” indicator. E.g. from M import * does not import objects whose names start with an underscore.
- single_trailing_underscore_: used by convention to avoid conflicts with Python keyword, e.g. : tkinter.Toplevel(master, class_='ClassName')
- __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.

Prescriptive: Naming Conventions

Names to Avoid

단일 문자 변수 이름으로 'l', 'O', 또는 'I' 문자를 사용하지 말기.

Package and Module Names

모듈은 짧은 소문자의 이름을 가져야 한다. 가독성을 높이기 위해 밑줄을 사용할 수 있다.

Class Names

클래스 이름은 일반적으로 CapWords 규칙을 사용해야 한다.

Type Variable Names

covariant 또는 contravariant behavior를 선언하는 데 사용되는 변수에는 각각 _co 또는 _contra 접미사를 추가하는 것을 권장한다:

```
from typing import TypeVar

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

Exception Names

예외는 class여야 하며, 여기에도 class naming 규칙이 적용된다. 단, 예외 name에는 "Error" 접미사를 사용해야 한다.

Global Variable Names

규칙은 함수에 대한 것과 거의 유사하다.

Function and Variable Names

- 함수 이름은 가독성을 높이기 위해 필요한 경우 underscore로 나누어 소문자로 작성되어야 한다.
- 변수 이름도 함수 이름과 동일한 규칙을 따른다.

Function and Method Arguments

- 인스턴스 메서드의 첫 번째 인자로는 항상 self를 사용하자.
- 클래스 메서드의 첫 번째 인자로는 항상 cls를 사용하자.
- 함수 인자의 이름이 예약된 키워드와 충돌하는 경우, 일반적으로 약어나 철자의 변형 대신에 하나의 underscore를 추가하는 것이 좋다.

Method Names and Instance Variables

- function naming 규칙 사용: 가독성을 높이기 위해 필요한 경우 단어 사이에 underscore로 소문자로 모두 작성하자.
- 비공개 메서드와 인스턴스 변수에 대해서는 하나의 선행 언더스코어만 사용하자.
- 하위 클래스와의 이름 충돌을 피하기 위해 Python의 name mangling 규칙을 활용하기 위해 두 개의 선행 언더스코어를 사용하자.

Constants

Constant는 일반적으로 모듈 수준에서 정의되며, 모두 대문자로 쓰여야 하며, 단어 사이에 언더스코어로 구분되어야 한다.

ex: MAX_OVERFLOW and TOTAL.

Designing for Inheritance

- 항상 클래스의 메서드와 인스턴스 변수가 public이어야 할지 아니면 non-public이어야 할지를 결정하자. 의심이 생기면 non-public을 선택하자.
- 또 다른 attribute 카테고리는 "subclass API"의 일부인 attribute들이다. (다른 언어에서는 종종 "protected"로 불린다). 일부 클래스는 상속을 받아 클래스의 동작을 확장하거나 수정하기 위해 설계된다. 이러한 클래스를 디자인할 때 어떤 속성이 public인지, 어떤 것이 subclass API의 일부인지, 어떤 것이 base class에서만 사용되어야 하는지에 대한 결정을 내리는 데 주의하자.
- Public attributes는 leading underscore를 가져서는 안된다.
- 만약 public attribute name이 예약된 키워드와 충돌한다면 attribute name 뒤에 하나의 밑줄을 추가하자.
- 단순한 public data attributes의 경우, 복잡한 접근자/변경자 메서드 없이 attribute name 자체만 노출하는 것이 가장 좋다.
- If your class is intended to be subclassed, and you have attributes that you do not want subclasses to use, consider naming them with double leading underscores and no trailing underscores.

Public and Internal Interfaces

In software development, backwards compatibility applies only to public interfaces, which are documented and explicitly declared. Public interfaces are defined using the __ all__ attribute, and an empty __ all__ list indicates no public API.

Undocumented interfaces are considered internal. Even with all set, internal interfaces should have a leading underscore. Any containing namespace (package, module, or class) is also considered internal. Imported names are implementation details, and reliance on indirect access should only occur if explicitly documented in the module's API.

Programming Recommendations

- Code should be written in a way that does not disadvantage other implementations of Python
- Comparisons to singletons like None should always be done with is or is not, never the equality operators.
- Use is not operator rather than not ... is.
- When implementing ordering operations with rich comparisons, it is best to implement all six operations (__ eq__, __ ne__, __ lt__, __ le__, __ gt__, __ ge__) rather than relying on other code to only exercise a particular comparison. To minimize the effort involved, the functools.total_ordering() decorator provides a tool to generate missing comparison methods.
- Always use a def statement instead of an assignment statement that binds a lambda expression directly to an identifier
- Derive exceptions from Exception rather than BaseException.
- Use exception chaining appropriately. raise X from Y should be used to indicate explicit replacement without losing the original traceback.
- When catching exceptions, mention specific exceptions whenever possible instead of using a bare except: clause

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

- Additionally, for all try/except clauses, limit the try clause to the absolute minimum amount of code necessary. Again, this avoids masking bugs

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

- When a resource is local to a particular section of code, use a with statement to ensure it is cleaned up promptly and reliably after use. A try/finally statement is also acceptable.
- Either all return statements in a function should return an expression, or none of them should.
- Object type comparisons should always use isinstance() instead of comparing types directly

```
try:
    if isinstance(obj, int):
```

- Don’t compare boolean values to True or False using ==:

```
# Correct:
if greeting:
```

Variable Annotations

- 모듈 레벨 변수, 클래스 및 인스턴스 변수, 그리고 지역 변수에 대한 주석은 콜론 뒤에 하나의 공백을 가져야 한다.
- 콜론 앞에는 공백이 없어야 합니다.
- assignment에 오른쪽 항이 있는 경우 등호 기호의 양쪽에 정확히 하나의 공백이 있어야 한다.

```
# Correct:
code: int

class Point:
```

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

Descriptive: Naming Styles

- 어떤 Naming Style이 사용되고 있는지 알아볼 수 있도록 도와준다.
- 일반적으로 다음과 같은 Naming Style이 있다.
  - b (single lowercase letter)
  - B (single uppercase letter) lowercase
  - lower_case_with_underscores UPPERCASE
  - UPPER_CASE_WITH_UNDERSCORES
  - CapitalizedWords
  - mixedCase
  - Capitalized_Words_With_Underscores

```
b (single lowercase letter)
    - B (single uppercase letter) lowercase
    - lower_case_with_underscores UPPERCASE
    - UPPER_CASE_WITH_UNDERSCORES
    - CapitalizedWords
    - mixedCase
    - Capitalized_Words_With_Underscores
```

또한 선행 또는 후행 밑줄을 사용하는 다음과 같은 특수 형식이 인식된다.

- _single_leading_underscore: weak “internal use” indicator. E.g. from M import * does not import objects whose names start with an underscore.
- single_trailing_underscore_: used by convention to avoid conflicts with Python keyword, e.g. : tkinter.Toplevel(master, class_='ClassName')
- __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.

Prescriptive: Naming Conventions

Names to Avoid

단일 문자 변수 이름으로 'l', 'O', 또는 'I' 문자를 사용하지 말기.

Package and Module Names

모듈은 짧은 소문자의 이름을 가져야 한다. 가독성을 높이기 위해 밑줄을 사용할 수 있다.

Class Names

클래스 이름은 일반적으로 CapWords 규칙을 사용해야 한다.

Type Variable Names

covariant 또는 contravariant behavior를 선언하는 데 사용되는 변수에는 각각 _co 또는 _contra 접미사를 추가하는 것을 권장한다:

```
from typing import TypeVar

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

Exception Names

예외는 class여야 하며, 여기에도 class naming 규칙이 적용된다. 단, 예외 name에는 "Error" 접미사를 사용해야 한다.

Global Variable Names

규칙은 함수에 대한 것과 거의 유사하다.

Function and Variable Names

- 함수 이름은 가독성을 높이기 위해 필요한 경우 underscore로 나누어 소문자로 작성되어야 한다.
- 변수 이름도 함수 이름과 동일한 규칙을 따른다.

Function and Method Arguments

- 인스턴스 메서드의 첫 번째 인자로는 항상 self를 사용하자.
- 클래스 메서드의 첫 번째 인자로는 항상 cls를 사용하자.
- 함수 인자의 이름이 예약된 키워드와 충돌하는 경우, 일반적으로 약어나 철자의 변형 대신에 하나의 underscore를 추가하는 것이 좋다.

Method Names and Instance Variables

- function naming 규칙 사용: 가독성을 높이기 위해 필요한 경우 단어 사이에 underscore로 소문자로 모두 작성하자.
- 비공개 메서드와 인스턴스 변수에 대해서는 하나의 선행 언더스코어만 사용하자.
- 하위 클래스와의 이름 충돌을 피하기 위해 Python의 name mangling 규칙을 활용하기 위해 두 개의 선행 언더스코어를 사용하자.

Constants

Constant는 일반적으로 모듈 수준에서 정의되며, 모두 대문자로 쓰여야 하며, 단어 사이에 언더스코어로 구분되어야 한다.

ex: MAX_OVERFLOW and TOTAL.

Designing for Inheritance

- 항상 클래스의 메서드와 인스턴스 변수가 public이어야 할지 아니면 non-public이어야 할지를 결정하자. 의심이 생기면 non-public을 선택하자.
- 또 다른 attribute 카테고리는 "subclass API"의 일부인 attribute들이다. (다른 언어에서는 종종 "protected"로 불린다). 일부 클래스는 상속을 받아 클래스의 동작을 확장하거나 수정하기 위해 설계된다. 이러한 클래스를 디자인할 때 어떤 속성이 public인지, 어떤 것이 subclass API의 일부인지, 어떤 것이 base class에서만 사용되어야 하는지에 대한 결정을 내리는 데 주의하자.
- Public attributes는 leading underscore를 가져서는 안된다.

- 만약 public attribute name이 예약된 키워드와 충돌한다면 attribute name 뒤에 하나의 밑줄을 추가하자.
- 단순한 public data attributes의 경우, 복잡한 접근자/변경자 메서드 없이 attribute name 자체만 노출하는 것이 가장 좋다.
- If your class is intended to be subclassed, and you have attributes that you do not want subclasses to use, consider naming them with double leading underscores and no trailing underscores.

Public and Internal Interfaces

In software development, backwards compatibility applies only to public interfaces, which are documented and explicitly declared. Public interfaces are defined using the __ all__ attribute, and an empty __ all__ list indicates no public API. Undocumented interfaces are considered internal. Even with all set, internal interfaces should have a leading underscore. Any containing namespace (package, module, or class) is also considered internal. Imported names are implementation details, and reliance on indirect access should only occur if explicitly documented in the module's API.

Programming Recommendations

- Code should be written in a way that does not disadvantage other implementations of Python
- Comparisons to singletons like None should always be done with is or is not, never the equality operators.
- Use is not operator rather than not ... is.
- When implementing ordering operations with rich comparisons, it is best to implement all six operations (__ eq__, __ ne__, __ lt__, __ le__, __ gt__, __ ge__) rather than relying on other code to only exercise a particular comparison. To minimize the effort involved, the functools.total_ordering() decorator provides a tool to generate missing comparison methods.
- Always use a def statement instead of an assignment statement that binds a lambda expression directly to an identifier
- Derive exceptions from Exception rather than BaseException.

- Use exception chaining appropriately. raise X from Y should be used to indicate explicit replacement without losing the original traceback.
- When catching exceptions, mention specific exceptions whenever possible instead of using a bare except: clause

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

- Additionally, for all try/except clauses, limit the try clause to the absolute minimum amount of code necessary. Again, this avoids masking bugs

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

- When a resource is local to a particular section of code, use a with statement to ensure it is cleaned up promptly and reliably after use. A try/finally statement is also acceptable.
- Either all return statements in a function should return an expression, or none of them should.
- Object type comparisons should always use isinstance() instead of comparing types directly

```
try:
    if isinstance(obj, int):
```

- Don’t compare boolean values to True or False using ==:

```
# Correct:
if greeting:
```

Variable Annotations

- 모듈 레벨 변수, 클래스 및 인스턴스 변수, 그리고 지역 변수에 대한 주석은 콜론 뒤에 하나의 공백을 가져야 한다.
- 콜론 앞에는 공백이 없어야 합니다.
- assignment에 오른쪽 항이 있는 경우 등호 기호의 양쪽에 정확히 하나의 공백이 있어야 한다.

```
# Correct:
code: int

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

과제 2

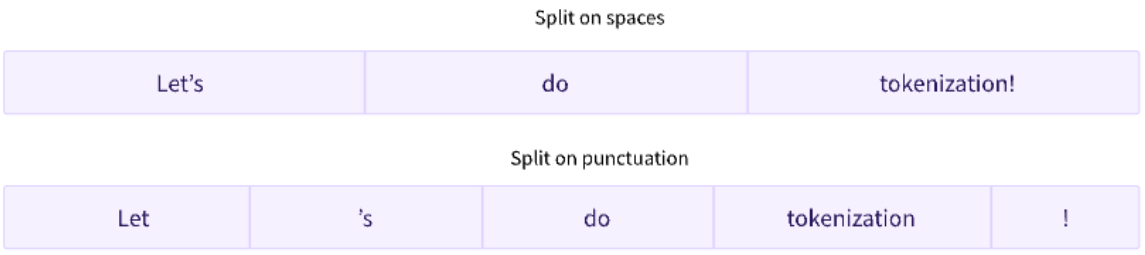
코딩 과제는 BPE, Word tokenizer를 구현하는 것이었습니다. 저희 팀은 tokenizers.py라는 파일에 BPE와 Word에서 공통적으로 동작하는 부분을 base tokenizer class로 구현한 다음, 이를 상속하여 BPE tokenizer class와 Word tokenizer class를 구현하였습니다.

BPE tokenizer class와 Word tokenizer class에 대한 원리는 아래에 적은 내용과 같고, 이러한 내용을 바탕으로 코드를 구현하였습니다.

Introduction

- Tokenizer
  - 입력된 텍스트를 모델에서 처리할 수 있게 변환하는 것
  - 모델은 숫자만 처리할 수 있으므로, 토크나이저는 텍스트 입력을 숫자 데이터로 변환하는 것이다.
- Space and Punctuation Tokenization
  - rule-based tokenization로도 불림
  - 방법

        - 띄어쓰기를 기준으로 나눈 뒤
        - punctuation이 나오면 정해진 규칙대로 나누는 방식
            - Don’t ⇒ Do 와 n’t로 나누기, 마침표는 따로 두기 등
    - 단점
        - Massive text corpora
            - vocabulary의 크기가 너무 크게 될 수도 있음
            - 이는 model이 너무 큰 embedding matrix를 갖게 만들기도 하는데, 이는 memory & time complexity에 안좋은 영향을 줌
            - 보통 한개의 언어를 이용한다면 50000보다 크지 않게 한다고 함
-
- 각 단어에는 0부터 시작하여 어휘집 크기 사이의 ID가 할당되어, 모델은 위에서 할당된 ID를 활용하여 단어를 식별한다.
- “dog”와 “dogs”가 다르게 식별 되기 때문에 엄청난 양의 토큰이 생성된다.
- 단어 사이의 similarity를 알 수 없다.
- 어휘집에 없는 단어는 “unknown” 토큰으로 표현한다.
- 어휘집을 만들 때 unkown 토큰을 최대한 적게 출력하는 것이 목표이다.
    - rule-based tokenization로도 불림
    - 방법
        - 띄어쓰기를 기준으로 나눈 뒤
        - punctuation이 나오면 정해진 규칙대로 나누는 방식
            - Don’t ⇒ Do 와 n’t로 나누기, 마침표는 따로 두기 등
    - 단점
        - Massive text corpora
            - vocabulary의 크기가 너무 크게 될 수도 있음
            - 이는 model이 너무 큰 embedding matrix를 갖게 만들기도 하는데, 이는 memory & time complexity에 안좋은 영향을 줌
            - 보통 한 개의 언어를 이용한다면 50000보다 크지 않게 한다고 함
- Word tokenizer



- 각 단어에는 0부터 시작하여 어휘집 크기 사이의 ID가 할당되어, 모델은 위에서 할당된 ID를 활용하여 단어를 식별한다.
- 단점: “cat”과 “cats”가 다르게 식별 된다. → 엄청난 양의 토큰이 생성
- 단어 사이의 similarity를 알 수 없다.
- 어휘집에 없는 단어는 “unknown” 토큰으로 표현한다.

Byte-Pair Encoding (BPE)

- Training data를 단어 단위로 쪼개는 pre-tokenizer 이용
  - Pretokenization은 space tokenization 정도로 생각하면 됨
  - 더 복잡한 pre-tokenization은 rule-based tokenization까지 포함하기도 함
  - Pretokenization 이후
    - 단어들의 set이 구성됨
    - 이 set의 단어들이 얼마나 많이 등장했는지도 확인 가능
- 단어 set에 나온 모든 symbol로 구성된 vocabulary를 생성
- 반복: 하이퍼파라미터로 정한 vocabulary size에 해당하는 크기가 될 때까지
  - 2개의 symbol을 하나의 새로운 symbol로 병합하는 규칙을 배워나감
  - 과정
    - 가장 많이 등장하는 symbol pair를 확인
    - 해당 symbol pair를 병합함
- 기본 symbol과 병합을 통해 생성된 symbol pair를 vocabulary에 포함한다.
  - 결과적으로, vocabulary의 크기는 (base symbol의 개수 + merge 횟수)가 된다
- 분석 대상 언어에 대한 지식이 필요 없다.

    - 말뭉치에서 자주 나타나는 문자열(서브워드)을 토큰으로 분석하기 때문에
- 언제까지 수행하게 되냐?: BPE 어휘 집합 구축은 어휘 집합이 사용자가 정한 크기가 될 때까지 반복해서 수행

▼ 예시1

1. Pretokenization 이후 - `("hug", 10), ("pug", 5), ("pun", 12), ("bun", 4), ("hugs", 5)`
2. 첫 vocabulary - `("h" "u" "g", 10), ("p" "u" "g", 5), ("p" "u" "n", 12), ("b" "u" "n", 4), ("h" "u" "g" "s", 5)`
3. u 다음 g가 나오는 빈도(10+5+10)가 가장 높음. 따라서 u와 g 병합 - `("h" "ug", 10), ("p" "ug", 5), ("p" "u" "n", 12), ("b" "u" "n", 4), ("h" "ug" "s", 5)`
4. u 다음 n의 빈도가 가장 높아서 병합 - `("h" "ug", 10), ("p" "ug", 5), ("p" "un", 12), ("b" "un", 4), ("h" "ug" "s", 5)`
5. h 다음 ug의 빈도가 가장 높아서 병합 - `("hug", 10), ("p" "ug", 5), ("p" "un", 12), ("b" "un", 4), ("hug" "s", 5)`

여기서 마친다면, 최종 vocabulary는 ["b", "g", "h", "n", "p", "s", "u", "ug", "un", "hug"]가 된다.

▼ 예시2

aaabdaaabac

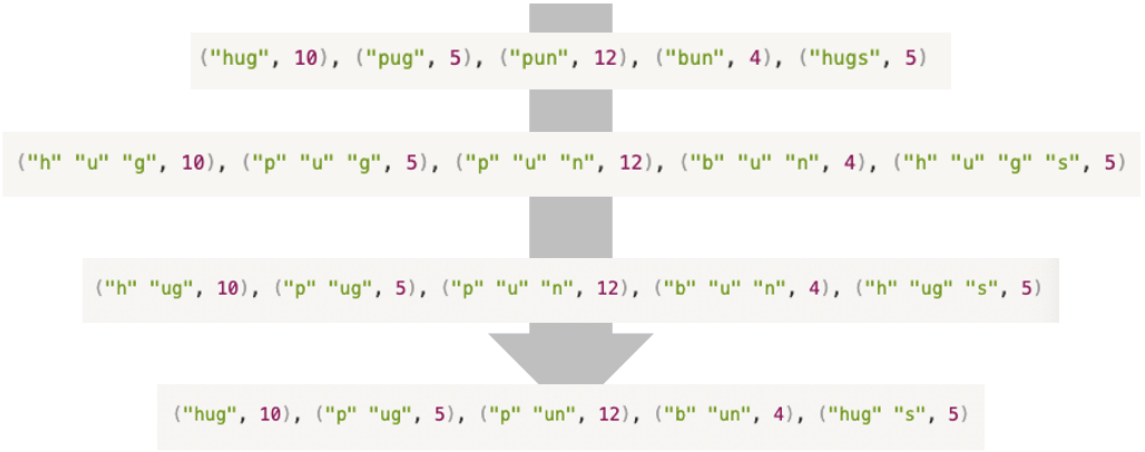
- 연속된 두 글자를 한 글자로 병합
    - aa → Z 치환
- **Z**abd**Z**abac: ab가 가장 많이 나타났으므로 이를 다시 Y로 치환
    - 단, 알파벳 순으로 앞선 것을 먼저 병합
- Z**Y**dZ**Y**ac: ZY 역시 X로 치환 가능. 한 번 더 병합 수행
- **X**d**X**ac
    - BPE를 수행하기 전: 사전 크기가 4개(a, b, c, d)
    - 수행 이후: 사전 크기가 7개(a, b, c, d, Z, Y, X) → 사전 크기 증가
    - 데이터의 길이: 11에서 5로 줄었음.

→ 사전 크기를 지나치게 늘리지 않고, 각 데이터 길이는 효율적으로 압축할 수 있도록 함

Byte-level BPE

- A base vocabulary that includes all possible base characters can be quite large
  - all unicode characters are considered as base characters 할 경우
- To have a better base vocabulary, GPT-2 uses bytes as the base vocabulary
  - 모든 base character를 포함해도 base vocabulary의 크기가 256이면 충분함
  - Punctuation을 위한 규칙을 추가하면, GPT2의 tokenizer는 <unk> symbol없이 모든 text를 tokenize할 수 있음
  - 50000번의 merge + 256 + 1 (End Of Text token)으로 vocabulary 크기는 50257이 됨

BPE tokenizer 알고리즘 요약



1. pre-tokenize 가정: 하나의 문장이 띄어쓰기 등으로 분리되어 있다.
2. 각 pre-tokenize로 나누어진 단어들의 빈도를 계산한다.
3. 각 단어들을 하나의 글자 (character) 단위로 나눈다.
4. 각 연속한 글자 pair들을 높은 빈도 순으로 merge한다.
5. 각 연속한 글자 pair들을 높은 빈도 순으로 merge한다.

6. 각 연속한 글자 pair들을 높은 빈도 순으로 merge한다.

출처)

- Tokenizer란?: https://huggingface.co/docs/transformers/tokenizer_summary
- Byte Pair Encoding이란?: https://en.wikipedia.org/wiki/Byte_pair_encoding
- BPE tokenizer?: https://arxiv.org/abs/1508.07909v5
- Byte pair encoding 설명 (BPE tokenizer, BPE 설명, BPE 예시) - 유니의 공부 (tistory.com)
- Byte Pair Encoding - ratsgo's NLPBOOK
- chatGPT

Summary of the tokenizers

We're on a journey to advance and democratize artificial intelligence through open source and open science.

🤗 https://huggingface.co/docs/transformers/tokenizer_summary