Python Enhancement Proposal (PEP) 8 Tutorial

이 책은 Python Enhancement Proposal (PEP)는 파이썬을 개선하기 위한 제안서, 즉 PEP에 대한 전반적인 개념을 설명하고 그 중에서도 Python 코딩 스타일에 대한 권고사항에 대한 PEP 8에 대한 튜토리얼을 제공하기 위해 작성되었습니다.

PEP는 보다 나은 파이썬을 위한 다양한 의견을 체계적으로 정리한 문서를 의미합니다.

파이썬을 배우는 사람들은 먼저 문법을 배우고 다양한 예제 실습을 통해 프로그래밍 능력을 키워 갑니다. 파이썬 문법을 배우고 다양한 예제를 실습하는 것은 매우 중요합니다. 하지만 어느 수준 이상 올라가게 되면 더 이상 혼자서 코딩을 할 수 없게 됩니다.

일반적인 소프트웨어 개발의 경우 다양한 기능을 구현해야 하므로 다음과 같은 과정을 거치게 됩니다.

- 기능 단위로 소프트웨어를 구조화하고 각 기능을 분할합니다.
- 개발자 개인별로 구현해야할 기능을 나눠 갖습니다.
- 각자 단위 기능 개발을 끝내면 하나의 프로그램으로 통합합니다.
- 전체 기능을 테스트합니다.
- 프로그램을 배포(서비스 개시)하고 운영유지 단계로 진입합니다.
- 운영유지 단계에서는 다양한 변경 요구사항을 반영하여 지속적인 업그레이드를 합니다.

위 과정에서 일관된 코딩 스타일의 필요성이 등장합니다.

왜냐하면 다음과 같은 사건들이 끊임없이 발생하기 때문입니다.

> 일관된 코딩 스타일이 필요한 경우
>
> - 다른 팀원이 구현해서 보내준 소스코드를 받아보니 코드 분석이 어렵다.
> - 모든 팀원의 코드를 통합(하나의 프로젝트로 완성) 해보니 코딩 스타일이 중구난방이다.
> - Github에서 필요한 소스코드를 받았는데 해석이 어렵다.
> - 우리 회사에서 개발한 소프트웨어를 다른 회사에서 유지보수하게 되었는데, 유지보수 개발자들이 우리 코드를 이해해지 못해 계속 컴플레인 한다.
> - 내가 짠 코드인데 몇 달이 지나고 나니 나도 해석하기 어렵다.

위와 같은 이유로 파이썬 개발자들은 공통된 Convention을 필요로 하게 되었습니다. 코딩 Convention을 쉽게 이야기 하면 우리가 영어공부를 할 때 관용적 표현을 배우는 것과 비슷합니다. 관용이란 습관으로 굳어진 것을 의미합니다. 한자로 보면 '버릇 관 (慣)'과 '쓸 용(用)'이 합쳐서 오랫동안 써서 굳어진 대로 늘 사용하는 것을 뜻합니다. 관용적 표현이라는 것은 관용적으로 굳어진 것들을 사용한다는 의미가 되겠습니다.

파이썬은 '프로그래밍 언어' 입니다. '언어'는 어떤 객체(object) 사이에서 소통하기 위한 수단입니다. 프로그래머라는 객체가 컴퓨터라는 객체와 소통하기 위해서는 당연히 언어가 필요합니다. 프로그래머가 컴퓨터와 소통하기 위한 언어는 다양합니다. Java, C, C++, Fortran, Go 등등 수없이 많습니다. 그 중에서 여러분들은 Python이라는 언어를 가지고 컴퓨터와 소통하고 있을 겁니다.

Python도 언어이기 때문에 문법과 다양한 표현법이 존재합니다. 똑같은 기능을 구현하더라도 코딩하는 방법(코딩 스타일)은 다양하게 존재합니다. 프로그래머 입맛에 따라 거의 무한대의 방법이 존재할 겁니다. 하지만 영어(언어)의 관용구처럼 Python(언어)도 관용적 표현, 즉 관용적 코딩 스타일이 자연스럽게 생기게 됩니다. 이렇게 생긴 파이썬 관용적 표현들을 따라 코딩하는 것을 'Pythonic (파이쏘닉)' 하다고 말합니다.

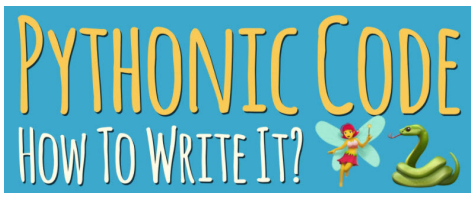


Fig. 1 Pythonic Image (source: https://tali.tistory.com/1356)

파이쏘닉한 소스코드는 다른 개발자들이 편하게 소스코드를 읽고 해석할 수 있기 때문에 개발 속도와 유지보수 효율성을 높일 수 있는 중요한 방법 중 하나입니다. 우리가 그걸 알면서 굳이 사용하지 않을 필요는 없는 겁니다.

그래서 등장한 것이 통일된 코딩 규칙을 갖자는 것입니다. 그 통일된 규칙을 체계적으로 정리한 것이 PEP 8입니다. 파이썬을 배우는 사람들은 문법과 실습에 집중하면서 자연스럽게 Pythonic 코드를 간접적으로 배우게 됩니다.

하지만 어떤 경우는 전혀 그렇지 못한 경우도 있을 것입니다. Pythonic 하게 코딩하는 것은 그리 어려운 것이 아닙니다. Python 문법을 공부한 사람이라면 한두번 설명만 들으면 누구라도 할 수 있습니다.

PEP는 수많은 종류가 있습니다. 이번 튜토리얼은 PEP의 개념 'Stype Guide for Python Code'로 불리는 PEP 8에 대하여 설명하겠습니다.

튜토리얼은 크게 2개의 Chapter로 구성되어 있습니다.

- Chapter 1: PEP의 개념과 종류에 대한 소개
- Chapter 2: PEP 8 구체적 설명

PEP 8은 파이썬의 코딩 스타일을 제안하는 내용입니다. 전문 개발자가 되기 위해 알아두어야 합니다. 파이썬 전문 개발자가 아니도라도 프로그래머들이 사용하는 코딩 스타일을 알고 있다면 협업이나 코드 분석을 쉽게 할 수 있는 장점이 있습니다.

다시 말하면 프로그래밍 생산성을 높일 수 있다는 의미입니다. 파이썬 문법을 배우고 다양한 예제를 풀어보는 등 공부는 많이 하지만 코딩 스타일을 정확하게 배우는 경우는 드물게 됩니다. 이번 튜토리얼을 통해 보다 전문적인 지식을 얻기 바랍니다.

> **Note**
>
> 튜토리얼은 파이썬 공식 홈페이지에 게시된 문서를 참고하여 작성되었습니다.
>
> **Python PEP**: https://www.python.org/dev/peps

> 저자소개
>
> 청주대학교 소프트웨어융합학부 인공지능소프트웨어전공
>
> 노기섭 교수
>
> Contact
>
> - E-mail: kafa46@cju.ac.kr
> - Phone: 043-229-8496 (유선)
> - Mobile: Not open to public (private, 비공개)
>
> 
>
> ***Fig. 2*** 청주대 노기섭 교수

---

1. Python Enhancement Proposal (PEP)

1.1. PEP가 도대체 뭔가요?

- PEP는 'Python Enhancement Proposal' 이라는 말 그대로 파이썬 개선하기 위한 제안서입니다.
- 파이썬 사용자들에게 정보를 제공하는 설계 문서로 볼 수도 있고, 파이썬의 작동 또는 환경에 대한 새로운 특징(기능)을 표현하는 문서로 볼 수도 있습니다.
- PEP는 파이썬의 특징이나 원리를 간결하게 표현한 기술 문서(스펙)입니다.

- PEP는 파이썬에 새로운 기능이나 특징을 제안하는 가장 중심적인 매커니즘(작동방식)입니다.
- PEP를 통해 파이썬 커뮤니티의 이슈를 수집하고, 파이썬이 나아갈 미래의 방향을 결정합니다.
- PEP의 저자(작성자)는 파이썬 커뮤니티로부터 동의를 이끌어 내고 관련 의견을 문서화할 책임을 갖게 됩니다.
- PEP는 버전 번호가 부여된 텍스트 파일 형태로 유지되기 때문에 변경 이력을 기록하고 관리해야 합니다.

> **Note**
>
> 본 튜토리얼은 PEP 1을 참고로 작성되었습니다.
>
> **PEP 1**: https://www.python.org/dev/peps/pep-0001/

1.2. PEP를 보는 사람들은 누가인가요?

- 당연히 PEP를 보는 사람은 모든 파이썬 개발자입니다.^^

- 하지만 PEP는 전문 지식과 경험이 필요하므로 PEP 제정, 결정, 유지보수와 관련된 인원들은 별도로 존재합니다.
- PEP를 보는 사람들은 CPython (우리가 일반적으로 사용하는 파이썬) 인터프리터의 핵심 기능을 개발하는 사람들과 그 중에서 선발된 운영위원회 멤버들 입니다.
- 파이썬 언어 스펙을 이용하여 다른 기술 문서(스펙)을 만드는 개발자도 포함됩니다.
- 하지만, 위에서 설명한 것과 다른 성격의 파이썬 커뮤니티라도 희망하는 API 사용에 대한 의견을 제시하려는 경우에 해당하는 사람들도 PEP의 독자가 될 수 있습니다.
- 또한 다수 프로젝트에 걸친 협업이 필요한 복잡한 문제에 대한 관리가 필요한 경우에도 PEP 프로세스를 활용할 수 있습니다.

아래에 PEP와 관련된 이해관계자(Stakeholders)를 별도로 정리하였습니다.

1.2.1. 파이썬 운영위원회 (Python's Steering Council)

- 파이썬 운영위원회는 다양한 표현을 사용합니다.
- 현재는 'Python's Steering Council' 이라는 표현을 씁니다.
- 파이썬 운영위원회의 역할은 PEP를 최종적으로 선택할지 또는 기각할지를 결정합니다.
- PEP와 관련해서 가장 강력한 권한입니다.
- 운영위원회 멤버를 선출하는 절차는 PEP 13을 따릅니다.

> **PEP 13 - Python Language Governance**
>
> PEP 13: https://www.python.org/dev/peps/pep-0013/
>
> 파이썬 언어를 통치(종합관리)하기 위한 일련의 절차를 정의 운영위원회 관련 선발, 투펴, 이력 등 포함 현재 운영위원회 투표자, 선발자, 투표방식 등은 PEP 8012에 명시되어 있으니 관심있는 사람은 내용을 확인해 보기 바랍니다.

1.2.2. 파이썬 핵심 개발자 (Python's Core Developers)

- 파이썬 핵심 개발자 "core developers"는 PEP 8012에 따라 선출되어 현재 활동중인 멤버를 의미합니다.

1.2.3. 파이썬 BDFL

- PEP 1에 명시되어 있는 DBFL은 예전에는 'BDFL-Delegate'라는 용어를 썼습니다.

- BDFL은 'Benevolent Dictator for Life (자비로운 종신 독재자)'라는 의미입니다.

> **BDLF (**자비로운 종신 독재자란**)?**
>
> 오픈 소스 소프트웨어를 창시하거나 개발한 리더에게 부여되는 호칭
>
> 주로 커뮤니티 내에서 논쟁이 있을 경우 최종 결론을 내려주는 사람
>
> 해당 프로젝트의 창시자인 경우가 대부분임
>
> Online Wiki: Benevolent dictator for life (영문), BDFL (한글)

- 파이썬의 BDFL은 과거 귀도 반 로섬이었지만 2018년 7월 12일 부로 사임하였음

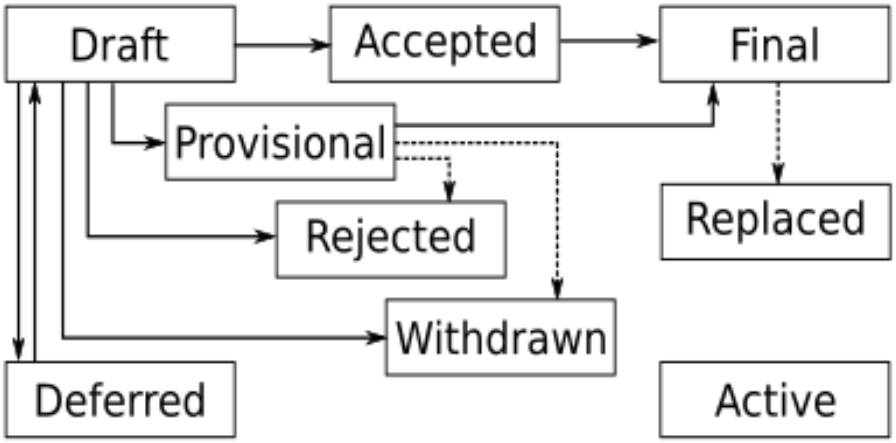
> 귀도 반 로섬**:** 파이썬 창시자
>
> 1995년 파이썬을 창시한 귀도 반 로섬을 가리키는 호칭으로 처음 **BDFL**이 사용됨
>
> 파이썬 창시 이후 2018년 7월까지 BDFL로 지내다 사임하였으나, 2019년에 다시 투표로 추대되었으나 2020 선거에서 후보로 추대되는 것에 스스로 사임하였음.
>
> 

- 현재 활동중인 '파이썬 핵심 개발자 (Python's core developers)'들에 PEP 13에 투표로 선출된 인원들을 **PEP-Delegate (PEP** 대리인**)** 이라고 부릅니다.
- 현재는 **PEP-Delegate**와 **BDFL-Delegate**는 같은 의미입니다.

1.2.4. PEP 편집자 (Editors)

- PEP 편집자는 PEP 작업 흐름을 관리하고 책임지는 사람입니다.
- 작업 흐름의 예로는 다음과 같은 것이 있습니다.
  - PEP 번호 부여
  - PEP 관리상태(내용 변경 등) 관리 등

PEP workflow & process



1.3. PEP에도 종류가 있나요?

PEP는 파이썬 성능 개선에 필요한 모든 사항들을 포함하고 있기 때문에 다양한 형태의 문서가 있습니다.

크게 문서는 세가지 분야로 정리할 수 있습니다.

1.3.1. Standard Track PEP (표준 트랙 PEP)

- Standard Track은 파이썬의 새로운 기능이나 그 기능을 구현한 것을 다룹니다.
- 현재 파이썬 표준에는 포함되어 있지 않지만, 향후 추가될 경우 기존 표준 파이썬 라이브러리와의 연동에 관한 사항을 기술하는 문서(PEP)입니다.

1.3.2. Informational PEP (정보 제공 PEP)

- 파이썬 디자인 이슈, 일반적인 가이드라인을 파이썬 커뮤니티에 제공합니다.
- 새로운 기능에 대하여 논의하지는 않습니다.
- 파이썬 커뮤니티 멤버들의 동의나 추천을 받을 필요는 없습니다.
- 따라서 파이썬을 이용해 구현하려는 사람들은 Informational PEP의 권고 사항을 반드시 따를 필요는 없습니다.

1.3.3. Process PEP (절차 PEP)

- 파이썬과 관련한 일련의 절차를 제공합니다.
  - 파이썬과 관련된 변경사항을 어떻게 처리할 것인지,
  - 어떻게 PEP를 처리할 것인지,
  - Standard Track PEP에 관한 내용 중 기능 이외의 것들을 어떻게 처리할 것인지
- 코딩 기반 구현에 직접적으로 해당하지 않지만 구현에 관련된 사항이 여기에 포함될 수 있습니다.
- Process PEP는 가끔씩 커뮤니티의 동의나 추천이 필요합니다. (Informational PEP와 차이점)
- 파이썬 사용자들은 Process PEP를 무시하면 안됩니다. (Informational PEP와 차이점)
- 주로 절차(procedures), 가이드라인(guidlines), 의사결정 과정의 변경, 파이썬 개발에 반영되어 있는 툴(tools) 또는 개발환경을 변경하려는 내용이 Process PEP에 해당됩니다.
- meta-PEP의 경우도 Process PEP에 해당합니다.

meta-PEP란?

PEP 프로세스 자체에 대한 PEP를 의미합니다.

1.4. PEP도 종류가 많은데요… 내가 원하는 것은 어디에서 찾나요?

- PEP 목록은 PEP 0에 체계적으로 정리되어 있습니다.

- PEP 0: Index of Python Enhancement Proposals (PEPs)
- 아래 링크를 클릭하여 어떤 종류의 PEP가 있는지 직접 확인해 보세요.

PEP 0 Link: https://www.python.org/dev/peps

> **PEP overview**를 마치며
>
> 이제 PEP의 개념과 구조에 대한 공부를 마쳤습니다. 본격적으로 우리가 목표로 하는 **'PEP 8. Style Guide for Python Code'**에 대하여 공부해 보도록 하겠습니다.

2. PEP 8 소개 (Introduction to PEP 8)

PEP:	8
Title:	Style Guide for Python Code
Author:	Guido van Rossum <guido at python.org>, Barry Warsaw <barry at python.org>, Nick Coghlan <ncoghlan at gmail.com>
Status:	Active
Type:	Process
Created:	05-Jul-2001
Post-History:	05-Jul-2001, 01-Aug-2013

Fig. 2.1 PEP 8 정리한 표

> **Note**
>
> PEP 8은 아래 링크를 참고하세요.
>
> **PEP 1**: https://www.python.org/dev/peps/pep-0008

- PEP 8은 main 파이썬 배포판에서 표준 라이브러리를 준수하는 파이썬 코드에 대한 컨벤션(convention)을 제공합니다.
- 파이썬은 C 언어로 만들어졌기 때문에 파이썬을 만든 C 언어로 코딩하는 스타일은 PEP 7을 참고할 수 있습니다.
  - C 언어는 파이썬과 친구 관계지만, 이번 튜토리얼에서는 C 언어 관련 사항은 생략합니다.
  - 관심있는 사람은 아래 링크를 참고하기 바랍니다.
  - PEP 7: https://www.python.org/dev/peps/pep-0007

> 컨벤션**(convention)**
>
> 많은 개발자들이 관용적으로 사용하여, 하나의 스타일로 정형화된 코딩 방식

- PEP 8은 귀도 반 로섬에 의해 최초로 제안된 Docstring Convention (PEP 257)과 Barry's GNU Mailman style guide의 일부가 반영되어 작성된 코딩 스타일 가이드입니다.
- 파이썬 코딩 스타일은 추가적인 컨벤션(convention)이 식별될 때마다 지속적으로 진화하고 있습니다.
- 파이썬 과거 버전은 언어 자체의 변화가 있는 경우 더 이상 반영하지 않습니다.
  - 종결처리(obsolete) 예시
    - Python 2 에 있고, Python 3에 반영되지 않는 경우
    - Pyton 3가 진화하면서 없어지는 문법들
- 대부분의 프로그래밍 언어는 그 자신만의 코딩 스타일을 가지고 있습니다.
- 다른 언어와 파이썬 사이에서 코딩 스타일의 충돌(conflicts)가 존재하는 경우는 해당 프로젝트에서 정한 규칙을 우선적으로 사용하는 것이 좋습니다.

3. A Foolish Consistency is the Hobgoblin of Little Minds

- "A Foolish Consistency is the Hobgoblin of Little Minds"을 문자 그대로 번역하면 “어리석은 일관성은 리틀 마인드의 홉고블린이다.” 이 된다. (구글 번역기)
- Hobgoblin은 영국의 민담에서 나오는 요정의 일종. 가장 대중적으로 알려진 홉고블린은 《한여름 밤의 꿈》에 나오는 장난꾸러기 요정 퍽(Puck)이다.

Fig. 3.1 홉고블린 이미지 (이미지 출처: 나무위키 홉고블린)

- "A Foolish Consistency is the Hobgoblin of Little Minds"은 영국 속담으로 "어리석은 고집은 소인배에게나 들러붙는 말성쟁이"라는 의미라고 한다. 주로 낡은 방식이나 사상을 고집하는 수구 정치인, 종교인, 학자들을 비난할때 쓰입니다. (참고문헌: 나무위키 홉고블린)
- 파이썬 프로그래머로 볼 경우 과거의 방식을 고집하는 개발자에게 자주 나타나는 것이 어리석은 고집을 부리거나 과거 방식을 답습하는 것으로 해석할 수 있습니다.
- 다르게 표현한다면 새로운 효율성을 무시하고 과거의 습관만을 고집하는 개발자로 해석할 수도 있습니다.
- 프로그래머들에 대한 Top 10 계명 중 6가지를 정리한 블로그를 소개합니다.
- 한 번 읽어보는 것도 좋을 것 같습니다. 아래 링크를 참고하세요
  - Top 6 List of Programming Top 10 Lists

3.1. 파이썬 창시자인 귀도 반 로섬의 통찰(Insight)

- 코드는 쓰는 것보다 훨씬 더 자주 읽힌다는 것입니다. 프로그래밍 과정에서 소스코드를 만드는 것보다는 소스코드가 만들어진 이후 읽히는 경우가 훨씬 많다는 것이 귀도의 생각이었습니다.
- 처음 프로그래밍(파이썬) 언어를 배우는 사람들은 아마도 감이 안잡히는 표현일 수 있습니다.
- 하지만, 구현(코딩) 능력이 올라갈수록 코드를 읽는 경우가 더 많다라는 의미를 자연스럽게 깨닫게 될 것입니다.
- 이러한 표현은 PEP 20 ‘The Zen of Python’ (파이썬의 선(禪))에서 언급한 바와 맥락이 같습니다.

3.2. 어차피 결론은 가독성(Readability)

- 파이썬을 처음 공부하는 사람들은 잘 느끼지 못할 수도 있지만, 가독성은 코딩에 있어서 매우 중요한 요소입니다.
- 가독성의 중요성은 여러가지가 있지만 대표적인 것들을 정리하면 다음과 같습니다.

> 💡 가독성이 중요한 경우
>
> 대부분의 프로그램 개발은 다양한 기능으로 분할하고 개발자별로 담당 기능을 구현한다.
>
> 단위 기능 구현이 끝나면 통합하고 테스트하여 하나의 프로그램을 완성한다.
>
> 다른 팀원으로 코드를 받았는데 해석이 어려운 경우가 있다.
>
> 물론 내 코드를 전달해 줬는데 어떤 로직인지 파악하기 힘들어 하는 경우도 있다.
>
> 여러 기능을 통합했더니 코딩 스타일이 중구난방이어서 일관성을 찾기 힘들다.
>
> 개발이 끝난 코드를 유지보수 업체에 넘겼는데 코드 해석이 어렵다는 질문이 계속 날아온다.

- PEP 8은 코딩 스타일에 대한 통일성을 추구하기 위한 제안서입니다.
- 특정 프로젝트에서 코딩의 일관성(consistency)의 중요성은 매우 큽니다.
- 특히 단일 모듈(module)이나 함수(function)에서의 일관성은 매우 중요합니다.
- 하지만, 일관성이 없는 코드를 발견한 경우 PEP 8에서 추천하는 코딩 스타일을 정확하게 적용하기 어려운 경우도 있습니다.
- 이런 경우에는 여러분 각자가 다른 샘플을 찾아보고 최선이라고 생각하는 것을 적용해야 합니다.
- 제발 모르면 물어보는 것을 주저하지 말도록 합시다!

> **Note**
>
> 이전 버전과의 호환성 PEP 8에서 추천하는 코딩 스타일을 최대한 따라야 하지만, 이전 버전의 Python에서 제공하던 스타일과의 일관성을 완전히 무시해서는 안된다는 점을 기억하자.

3.3. 코딩 스타일을 무시해야 하는 경우

- 일관된 코딩 스타일을 유지해야 하는 수많은 이유가 있지만,
- PEP 8의 권고사항을 준수하지 말아야 하는 경우도 있습니다.
- 그 중에서 PEP 8에서 명시적으로 기록한 내용을 정리하면 다음과 같습니다.

> 파이썬 코딩 스타일을 무시해야 하는 상황**(**가이드라인**)**
>
> 1. 코드를 분석하는 사람들이 PEP에 익숙하다 하더라도, PEP 8을 준수했는데 가독성이 떨어지는 경우
> 2. 여러분이 작성한 코드를 감싸는 다른 코딩 환경 때문에 가독성이 떨어지는 경우
> 3. 여러분이 확보한 코드가 PEP 8 이전에 작성되었고, 굳이 수정하지 않아도 되는 경우
> 4. PEP 8을 준수하지 않는 이전 스타일로 작성되었지만, PEP 8에서 지원하지 않는 코딩 스타일을 가지고 있는 경우

---

4. Code Lay-out

4.1. Indentation (들여쓰기)

- 모든 들여쓰기는 공백 4칸을 사용

4.1.1. 괄호(대괄호 [ ], 중괄호 { }, 소괄호( ))에서 줄바꿈이 일어나는 경우,

- 다음 줄에 나오는 구성 요소들은 괄호 단위에서 수직으로 정렬되어야 한다.
- 좋은 예

```
# 들여쓰기를 통해 함수의 인자를 표시한 것을 확인할 수 있음
foo = long_function_name(var_one, var_two,
                         var_three, var_four)
```

- 나쁜 예

```
# 함수의 전달 인자인지, 아니면 다른 변수인지 혼동할 수 있음
foo = long_function_name(var_one, var_two,
    var_three, var_four)
```

4.1.2. 함수의 여는 괄호 다음에 argument가 없는 경우 이어지는 줄에서 argument 표현의 명확성을 위해 추가 들여쓰기를 해야 한다.

- 좋은 예

```
# 들여쓰기를 통해 인자의 연속적 표현을 명확히 하고
# 함수 내부의 다른 명령어(print문)와의 구별이 명확함
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)
```

- 나쁜 예

```
# 괄호안 인자들과 함수 내부의 명령어를 혼돈할 우려가 있음
# 함수의 인자와 함수 내부의 첫번째 줄과 구분이 어려움
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

4.1.3. 함수의 호출에서 인자를 여러 줄에 걸쳐 사용해야 하는 경우

- 좋은 예

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

- 4칸 들여쓰기를 하지 않았지만 여전히 괜찮은 예

```
# 여러줄에 걸쳐 인자를 사용하는 경우 공백 4칸이 기본이지만
# 아래 예시는 2칸 공백을 사용했음
# 여전히 가독성은 확보할 수 있으므로
# 프로그래머의 성향에 따라 선택적으로 활용 가능
foo = long_function_name(
  var_one, var_two,
  var_three, var_four)
```

4.1.4. if문 들여쓰기

- if문에 사용할 내용이 길어서 여러 줄에 걸쳐서 코딩해야 하는 경우
  - 두 글자로 구성된 if 다음에 한 칸 공백을 주고 괄호를 열어주면 다음 줄에서는 자연스럽게 4칸 공백 이후 코딩을 할 수 있습니다.
  - 이 경우 들여쓰기 가독성이 확보될 수 있지만 한 줄짜리 if문 이후에 자동으로 4칸 들여쓰기가 생성된 것으로 혼동될 여지가 있습니다.
- PEP 8은 이 경우 어떤 규칙을 제시하지는 않고 있지만 3가지 정도의 대안을 제시하고 있습니다. 3가지 대안 모두 가능하지만, 다른 방법이 있다면 프로그래머가 적절히 선택해서 코딩해도 무방합니다.

- 옵션 1. 여러줄에 걸친 if문에서 들여쓰기를 하지 않는 경우

```
if (this_is_one_thing and
    that_is_another_thing):
    do_something()
```

- 옵션 2. 여러 줄에 걸친 if문이 끝난 다음에 주석을 활용하여 명시적으로 표시

```
# 주석의 경우 색깔이 다르게 표현되므로
# if문 들여쓰기와 if문 종료후 들여쓰기 구분 가능
if (this_is_one_thing and
    that_is_another_thing):
    # Since both conditions are true, we can frobnicate.
    do_something()
```

- 옵션 3. 여러 줄에 걸친 if문에 추가적 들여쓰기 적용

```
# if 문에 의한 들여쓰기와 if문 이후 실행 코드를 구분할 수 있음
if (this_is_one_thing
        and that_is_another_thing):
    do_something()
```

4.1.5. 괄호(대괄호 [ ], 중괄호 { }, 소괄호 ( )) 닫는 방법

- 괄호 안의 내용을 여러줄에 걸쳐 작성한 경우
- 괄호 내용이 끝나는 다음 줄에서
- 괄호 내용이 시작되는 위치에서 공백 없이 괄호를 닫아도 됩니다.

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

result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
    )
```

- 옵션: 괄호 내용이 끝나는 다음 줄의 첫번째 위치에 공백 없이 괄호를 닫는 방법도 가능

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

result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
)
```

4.2. 탭(Tab) 또는 공백(space)

- 들여쓰기는 공백(space) 사용하는 것이 좋습니다.
  - 일반적으로 탭을 사용하면 공백 4칸이 자동으로 생성되지만
  - 운영체제나 에디터에 따라 탭의 공백수 설정이 달라질 수 있기 때문입니다.
- 탭을 한번이라도 사용한 코드라면 일관성 유지를 위해 모든 들여쓰기 공백은 탭으로 사용해야 합니다.
- 파이썬은 탭과 공백을 섞어서 들여쓰기 하는 것을 허용하지 않습니다.
  - 간혹 탭과 공백을 섞어써도 작동하는 것은 에디터(VS code, PyCharm 등)가 내부적으로 탭을 공백으로 자동 변환 해주기 때문입니다.

4.3. 한 줄 최대 길이(Maximum Line Length)

- 한 줄 최대 글자수는 79 글자 이내로 작성해야 합니다.
- 특히, 글자가 연속적으로 계속되는 docstring 또는 주석의 경우는 72 글자로 제한해야 합니다.

- 한 줄 최대 글자수를 제한하는 이유는 다음과 같습니다.
  - 프로그래머들이 창을 여러 개 띄워서 코딩하는 경우 긴 줄은 작업을 어렵게 한다.
    - 코드 리뷰를 해야 하는 경우 여러 개 코딩 작업창을 띄워야 한다.
  - 대부분의 개발 도구들은 default wrapping 설정이 되어 있습니다. Default wrapping이 작동하는 경우 아주 긴 한줄의 코드가 있는 경우 그 코드를 이해하는 것을 더욱 어렵게 합니다.
  - Default wrapping의 경우 윈도우 폭을 80으로 설정하는 경우가 있습니다. 이런 환경에서도 가독성과 코드 분석에 혼선이 없도록 하기 위해서는 단어수를 최대 79글자로 제한해야 합니다.

> **Default Wrapping**이란**?**
>
> 텍스트 에디터, 워드 프로세서와 같은 편집기에서 한 줄 길이가 특정 글자수 이상에 도달한 경우 엔터(Enter) 키를 사용하지 않아도 다음 줄로 커서가 이동하는 것을 말합니다.

- 파이썬 표준 라이브러리의 경우 한 줄 최대 길이는 79글자, docstring/주석에서 한 줄 최대 길이는 72 글자로 제한하고 있습니다.

- 긴 줄을 처리하는 방법으로 괄호를 사용하는 것이 있습니다.
- 파이썬은 괄호 안에서 줄바꿈을 하는 것은 같은 줄이 계속 이어지는 것으로 해석하기 때문입니다.

- 긴 줄을 여러줄로 분할하기 위한 방법으로 백슬래시(\ )를 사용할 수 있습니다.
- 다음 예제는 암묵적 줄바꿈을 사용할 수 없는 with 구문에서 백슬래시(\ )를 적용한 사례입니다.

```
with open('/path/to/some/file/you/want/to/read') as file_1, \
     open('/path/to/some/file/being/written', 'w') as file_2:
    file_2.write(file_1.read())
```

4.4. 이항 연산자 (Binaray Operator) 줄바꿈은 어떻게 하지?

4.4.1. 과거의 이항 연산자 코딩 스타일

- 과거에는 이항 연산자 다음에 줄바꿈을 하는 것이었습니다.
- 그러나 이런 코딩 스타일은 가독성 문제가 제기되었습니다.
  - 화면상에 있는 연산자가 여러 줄에 걸쳐 흩어지는 문제
  - 각각의 이항 연산와 피연산자가 멀리 떨어지게 되는 문제

> 이항 연산자 **(Binary Operator)?**
>
> 연산자(operator)는 사칙연산 등과 같이 어떤 계산하기 위한 기호를 의미합니다.
>
> - (덧셈), - (뺄셈), * (곱셈), / (나눗셈) 등이 대표적입니다.
>
> 피연산자(operand) 연산자가 계산을 하기 위해 필요한 데이터를 말합니다.
>
> 덧셈을 하기 위해서는 2개의 피연산자가 필요합니다.
>
> 두 개의 피연산자 2와 3을 더하기 연산하기 위해서는 2 + 3 으로 표현합니다.
>
> 어떤 연산자는 피연산자가 하나만 필요한 경우도 있습니다.
>
> C 언어에서 ++, --, ! 등과 같은 경우입니다.

- 따라서 이항 연산자는 아래 예제와 같이 사용하여 가독성을 높여줍니다.
  - 과거의 이항 연산자 컨벤션(convention)과 반대로 코딩합니다.
  - 도널드 크누스는 그의 저서 'Computer and Typesetting Series'에서 다음과 같이 설명합니다.
  - “하나의 단락 내부에서 이항 연산자와 관계 연산자는 다음에 줄바꿈을 하지만, 보여주는 공식은 이항 연산자 이전에 줄바꿈을 한다”

- 좋은 예

```
# 이항 연산자와 피연산자를 쉽게 연관지을 수 있음
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

- 나쁜 예

```
# 이항 연산자와 피연산자가 멀리 떨어져 있어(줄바꿈) 가독성이 떨어짐
income = (gross_wages +
          taxable_interest +
          (dividends - qualified_dividends) -
          ira_deduction -
          student_loan_interest)
```

4.5. 빈 줄 (Blank Lines)의 활용

- 최상위 함수와 클래스는 2개의 빈 줄을 추가하여 가독성을 높여줍니다

```
# 최상위 함수 또는 클래스는 2개의 빈줄을 사용하여 구분


def foo(args):
    some code
    some code


class Foo():
    some code
    some code
```

- 클래스 내부에 선언되는 메서드(method)는 한개의 빈 줄을 추가하여 가독성을 높여줍니다.

```
class Foo():
    def __init__(self):
        pass

    # 메서드 사이에는 1개의 빈 줄로 구분
    def foo1(self, args):
        some code

    def foo2(self, args):
        some code
```

- 유사한 기능들을 구분하기 위해 빈 줄을 추가할 수 있습니다.

```
def foo():
    # 기능이 유사한 코드 영역
    code_1
    code_2
    code_3

    # 기능이 유사항 코드 영역
    code_4
    code_5
    code_6
```

- 함수 내부에서 논리적 영역의 분리를 표현하기 위해 빈 줄을 추가할 수 있습니다.

```
def foo():
    # 로직 A를 구현한 블록
    code_1
    code_2
    code_3

    # 로직 B를 구현한 블록
    code_4
    code_5
    code_6
```

- 일부 에디터의 경우 Control-L을 사용하면 공백문자로 인식하는 경우가 있습니다.
- Control-L을 사용하여 논리적 영역을 구분하는 특수문자로 사용할 수 있지만, 이러한 기능을 지원하지 않는 에디터도 있을 수 있기 때문에 개인적으로 사용하지 말 것을 추천합니다.

4.6. 소스 파일 인코딩

- 파이썬 배포판은 UTF-8 인코딩을 기본으로 지원합니다.
- 따라서 파이썬 소스코드에 인코딩을 별도로 선언하지 않으면 UTF-8을 기본 인코딩으로 인식합니다.
- UTF-8이 아닌 인코딩 사용을 최대한 자제해야 합니다.
    - UTF-8 이외의 인코딩 방식은 테스트 목적인 경우 등에 한해 제한적으로 사용하는 것이 좋습니다.
    - ASCII 계열이 아닌 인코딩 방식은 사용을 자제해야 합니다.
    - 사람 이름을 표현하기 위한 특수한 경우에 제한적으로 UTF-8을 사용할 수 있습니다.

ASCII 인코딩이란?

ASCII는 'American Standard Code for Information Interchange'의 약자입니다. '미국 정보 교환 표준 부호'라는 정식 명칭이 있지만 개발자들은 '아스키투'라고 발음합니다. 아스키 코드는 1963년 미국 ANSI에서 표준화한 정보교환용 1바이트(7비트 부호체계)입니다. 8비트 컴퓨터에서도 활용되어 오늘날 문자 인코딩의 근간을 이루고 있습니다. 영어 키보드로 입력할 수 있는 모든 글자가 숫자로 지정(인코딩)되어 있습니다. 1 바이트면 8비트이지만, 문자 표현을 위해 7 비트만 사용하고 나머지 1개 비트는 에러 검출용으로 사용합니다. $\(2^7=128\)$개 문자를 표현할 수 있습니다. ASCII는 영어를 주로 표현하고 7비트만 사용하기 때문에 전세계 다양한 언어를 표현할 수 없습니다. 이를 보완하기 위해 등장한 것이 유니코드입니다.

> **UTF-8** 인코딩이란**?**
>
> 먼저 유니코드(Unicode) 개념을 알아야 합니다. 유니코드는 전 세계의 모든 문자(다국어 환경)를 컴퓨터에서 일관되게 표현되도록 설계된 표준입니다.
>
> UTF-8 (Universal Coded Chacter Set + Transformation Format - 8 bit)는 유니코드 표준을 준수하는 글자 표현 방법 중 하나입니다. 특징은 한 개의 글자를 표현하기 위해 최대 4 바이트까지 사용할 수 있습니다. 따라서 정해진 비트수로 더 많은 글자를 표현할 수 있습니다. UTF-8 방식으로 문자를 표현하면 2,097,151개 글자 표현 가능하다. 당연히 한국어도 UTF-8의 코드북 어딘가에 정의되어 있습니다.

- PEP 8에서는 가급적 영어를 이용하여 코딩할 것을 권고하고 있습니다.
  - 영어를 사용하면 전세계 공용어를 사용하게 되므로 의사소통 및 코드분석에 효율적입니다.
  - 게다가 혹시 있을수 있는 인코딩 문제에서 보다 자유롭게 됩니다.

4.7. 임포트 (Imports)

파이썬 모듈이나 패키지를 임포트 하는 것도 정해진 규칙이 있습니다.

4.7.1. 개별 모듈은 가급적 서로 다른 줄에 표시해야 합니다.

- 좋은 예

```
import os
import sys
```

- 나쁜 예

```
import sys, os
```

4.7.2. 동일한 모듈에서 여러 개를 임포트하는 경우에는 한 줄 사용도 괜찮습니다.

- 아래와 같은 경우는 subprocess라는 하나의 모듈에서 Popen과 PIPE 라는 기능을 임포트 하는 경우입니다.
  - 이런 경우라면 한 줄에 여러개를 임포트 하는 것도 괜찮습니다.

```
from subprocess import Popen, PIPE
```

4.7.3. 임포트 하는 모듈의 종류에 따라 기록 순서를 다르게 합니다.

기본적으로 임포트는 소스 파일의 맨 위에 적습니다. 여러분이 임포트하는 모듈의 종류는 아래 3가지에 속하게 됩니다. 임포트 순서는 아래 순서대로 하는 것이 파이썬 컨벤션 입니다.

1. Standard library: 파이썬에서 제공하는 기본 라이브러리
2. Related third party: 다른 누군가가 만들어 놓은 라이브러리
3. Local application/library specific: 개인 또는 조직(회사)에서 자체적으로 만든 라이브러리

4.7.4. 절대 경로를 통해 임포트하는 것을 추천합니다.

절대 경로로 임포트하면 가독성이 높아지고 임포트 에러를 줄일 수 있습니다. 아래와 같이 임포트 하면 됩니다.

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

절대 경로로 임포트하는 것을 추천하지만, 복잡한 패키지 구조를 갖는 경우 명시적 상대 경로 임포트도 가능합니다. 복잡한 코드 이름을 생략하여 가독성을 향상시킬 수 있는 경우에 사용합니다. 다음 예시를 참고하세요.

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

파이썬 표준 라이브러리의 경우는 항상 절대 경로 임포트를 합니다.

4.7.5. 클래스를 포함하고 있는 모듈은 아래와 같이 임포트 합니다.

```
from myclass import MyClass
from foo.bar.yourclass import YourClass
```

하지만 현재 작성하고 있는 소스 파일에서 이름 공간(namespace) 충돌이 우려된다면 다음과 같이 임포트하여 사용하는 것도 괜찮습니다.

```
import myclass
import foo.bar.yourclass
```

위 예제와 같이 모듈을 임포트할 경우 “myclass.MyClass” 또는 "foo.bar.yourclass.YourClass"와 같이 사용하게 되므로 현재 소스 파일에 동일한 클래스 이름이 있더라도 충돌을 피할 수 있습니다.

4.7.6. 와일드카드 임포트는 자제해야 합니다.

와일드카드 임포트는 아스테리크(*) 문자를 이용한 임포트 방법입니다. 아래 예시를 참고하세요.

```
from foo import *
```

위와 같이 foo 모듈을 임포트하면 foo 모듈의 모든 내부 기능을 통째로 불러오게 됩니다. 이 경우 다음과 같은 심각한 문제가 있습니다.

- 현재 작성하고 있는 소스 파일과 이름 공간(namespace) 충돌 확률이 높습니다.
  - foo 모듈 내부에 어떤 이름을 가진 변수, 상수, 함수, 클래스가 있는지 명시적으로 파악하기 매우 어렵습니다.
- 코드 분석하는 사람에게 애매함(ambiguous)을 주게 됩니다.

4.8. 모듈 수준의 던더 이름 (Module Level Dunder Names)

언더스코어(_) 문자가 앞뒤로 2개씩 붙어 있는 함수를 던더(Dunder, Double Underscores) 메서드라고 부릅니다. 파이썬은 모든 것이 객체로 표현되는 "객체지향언어"입니다. 파이썬 인터프리터(윈도우의 경우 python.exe)에서 객체를 생성하거나, 표현하거나, 연산을 하는데 도움을 주기 위하여 미리 만들어 놓은 함수를 던더 메서드라고 합니다.

Built-in 함수들을 미리 만들어 놓고 객체 생성/연산 등을 할 때 마법과 같이 처리한다고 하여 'Magic Method (매직 메서드)'라는 별명이 있습니다. 파이썬 공식 문서에는 Special Method (스페셜 메서드)라고 명시하고 있기 때문에 '스페셜 메서드'라고 부르기도 합니다. **init**, **repr**, **new** 등과 같은 다양한 던더 메서드가 있습니다.

```
a = 1
print(dir(a))
```

```
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__',
'__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor__', '__floordiv__',
'__format__', '__ge__', '__getattribute__', '__getnewargs__', '__gt__', '__hash__',
'__index__', '__init__', '__init_subclass__', '__int__', '__invert__', '__le__',
'__lshift__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__',
'__pos__', '__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__',
'__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__',
'__ror__', '__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__',
'__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__',
'__subclasshook__', '__truediv__', '__trunc__', '__xor__', 'bit_length', 'conjugate',
'denominator', 'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
```

위 예제는 a라는 객체(변수)에 정수 1을 할당하고, 객체 a가 가지고 있는 메소드를 dir() 함수를 이용하여 출력한 것입니다. 객체 a 역시 다양한 던더 메서드를 가지고 있는 것을 확인할 수 있습니다. 프로그래머가 직접 만들지 않았지만 마법처럼 자동으로 만들어 졌습니다.

던더 메서드 역시 코딩 스타일이 존재합니다.

프로그래머가 던더 메서드를 작성할 경우 임포트(import)를 선언하기 이후에 작성합니다. 파이썬에서 docstring은 반드시 모듈의 맨 앞에 나와야 하므로 docstring이 있다면 docstring과 import 사이에 와야 합니다. 다만 **future** 모듈을 임포트하는 경우에는 from **future** import 구문 다음에 나와야 합니다.

> **future** 모듈이란**?**
>
> 파이썬은 다양한 built-in 기능들을 업그레이드 하면서 진화해 왔습니다. 'Python 2.xx'로 표현되는 2세대 파이썬과 'Python 3.xx'로 알려진 3세대 파이썬이 있습니다. 지속적인 성능개량을 하다 보니 2세대 파이썬에 정의된 built-in 기능들이 3세대 버전에서는 작동하지 않는 경우가 생겼습니다. 어떤 상황에서는 구세대 파이썬을 사용해야 하는 경우도 있습니다(구 버전의 인공지능 프레임워크로 개발했거나 개발 환경이 구세대에 최적화되어 있는 경우 등등). 이 경우 구세대 파이썬에서 최신 파이썬 기능을 사용하고자 할 경우에 사용하는 것이 **future** 모듈입니다.
>
> 예를 들어 Python 2.7 개발환경에서 다음과 같이 코딩한 경우
>
> from **future** import print_function
>
> Python 3.xx에서 사용하는 기능을 사용할 수 있습니다.
>
> - Python 2 Style: print “hello world”
> - Python 3 Style: print(“hello world”)

위 상황을 반영한 던더 사용 예제는 다음과 같습니다.

```
"""This is the example module.

docstring 내용 작성
"""

from __future__ import barry_as_FLUFL

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

import os
import sys
```

---

5. String Quotes (문자열 따옴표 사용)

파이썬 문자열 처리에서 활용하는 작은 따옴표(')와 큰 따옴표(")는 아무런 차이가 없습니다. PEP 8에서는 따옴표 사용에 대한 어떠한 추천 사항도 없습니다. 개발자 각자 문자열 처리에서 따옴표를 사용하는 규칙(습관)을 정하고 일관성 있게 사용하는 것이 중요합니다.

하나의 문장에서 작은 따옴표나 큰 따옴표를 사용할 경우 서로 다른 따옴표를 사용하는 것이 좋습니다. 동일한 따옴표를 사용할 경우 백슬래시(\ )를 사용해야 하므로 가독성을 떨어뜨릴 수 있습니다.

아래 코드는 정확히 같은 역할을 합니다.

- 동일한 따옴표를 사용한 예

```
print('오늘의 메뉴는 \'갈비탕\' 입니다.')
```

```
오늘의 메뉴는 '갈비탕' 입니다.
```

- 문자열 처리에서 서로 다른 따옴표를 혼합한 예

```
print("오늘의 메뉴는 '갈비탕' 입니다.")
```

```
오늘의 메뉴는 '갈비탕' 입니다.
```

```
print('오늘의 메뉴는 "갈비탕" 입니다.')
```

```
오늘의 메뉴는 "갈비탕" 입니다.
```

동일한 결과를 얻는 코드지만 두 번째, 세 번째 예제가 가독성이 더 높아집니다.

따옴표 세 개를 사용하는 트리플 따옴표(triple quote, ‘’’)의 내부에서 인용을 하는 경우는 쌍따옴표를 사용하는 것이 좋습니다. 왜냐하면 docstring 컨벤션인 PEP 257에서 이미 정의한 내용이기 때문입니다. 어떤 함수의 docstring에서 따옴표를 사용하는 예를 들면 다음과 같습니다.

```
def complex(real=0.0, imag=0.0):
    """Form a complex number.
    Keyword arguments:
    real -- the real part (default 0.0)
    "이 부분은 쌍따옴표를 썼습니다."
    imag -- the imaginary part (default 0.0)

    """
    if imag == 0.0 and real == 0.0:
        return complex_zero
    ...
```

6. White Space (공백 문자)

6.1. Pet Peeves

Pet Peeves는 애완동물(pet)을 입양했더니 짜증(peev)을 유발한다는 것에서 유래한 말이라고 합니다. 강아지를 입양했더니 여기저기 어지르는 상황이 예가 될 것 같습니다. 의역하면 “내가 싫어하는 것”, “불쾌한 것”, “화나는 것” 정도로 해석하면 될 것 같습니다.

파이썬에서 기본 원칙으로 제시하는 것은 “공백을 남용하지 말자” 입니다. 공백은 꼭 필요한 곳에 필요한 만큼만 쓰면 됩니다. 몇가지 예제를 살펴보도록 하겠습니다.

6.1.1. 괄호를 사용하는 경우는 괄호 안에 불필요한 공백을 사용하지 않습니다.

- 좋은 예

```
spam(ham[1], {eggs: 2})
```

- 나쁜 예

```
spam( ham[ 1 ], { eggs: 2 } )
```

6.1.2. 괄호에서 콤마를 마지막에 사용하는 경우 공백을 사용하지 않습니다.

- 좋은 예

```
foo = (0,)
```

- 나쁜 예

```
bar = (0,  )
```

6.1.3. 콤마(,), 세미콜론(;), 콜론(:) 바로 앞에는 공백을 사용하지 않습니다.

- 좋은 예

```
if x == 4: print x, y; x, y = y, x
```

- 나쁜 예

```
if x == 4 : print x , y ; x , y = y , x
```

6.1.4. 슬라이싱에서 콜론이 이항 연산자와 같이 사용된다면 콜론 기준으로 양쪽에 동일한 스페이스를 줍니다.

- 좋은 예

```
ham[1:9], ham[1:9:3], ham[:9:3], ham[1::3], ham[1:9:]
ham[lower:upper], ham[lower:upper:], ham[lower::step]
ham[lower+offset : upper+offset]
ham[: upper_fn(x) : step_fn(x)], ham[:: step_fn(x)]
ham[lower + offset : upper + offset]
```

- 나쁜 예

```
ham[lower + offset:upper + offset]
ham[1: 9], ham[1 :9], ham[1:9 :3]
ham[lower : : upper]
ham[ : upper]
```

6.1.5. 함수의 인자를 표시하는 괄호의 경우 공백을 사용하지 않습니다.

- 좋은 예

```
spam(1)
```

- 나쁜 예

```
spam (1)
```

6.1.6. 인덱싱이나 슬라이싱에 괄호를 사용하는 경우 시작 부분에 공백을 사용하지 않습니다.

- 좋은 예

```
dct['key'] = lst[index]
```

- 나쁜 예

```
dct ['key'] = lst [index]
```

6.1.7. 대입 연산자(=) 한칸의 공백만 줍니다. 변수 정렬을 위해 불필요한 공백은 사용하지 않습니다.

- 좋은 예

```
x = 1
y = 2
long_variable = 3
```

- 나쁜 예

```
x             = 1
y             = 2
long_variable = 3
```

6.2. Other Recommendations

6.2.1. 연산자와 관련한 공백

코딩 파일 전반에 걸쳐 명령어의 마지막에 불필요한 공백을 남겨놓지 않도록 합니다. 왜냐하면 한 줄의 마지막에 공백이 있을 경우 눈에 보이지 않기 때문에 혼동을 유발할 수 있습니다.

대입 연산자 (=), 증강 대입 연산자 (+=, -= 등), 비교 연산자(==, <, >, !=, <>, <=, >=, in, not in, is, is not), 불 연산자 (and, or, not)와 같은 이항 연산자는 전후에 공백을 하나씩 추가합니다.

만약 연산자가 괄호와 같은 우선순위를 표시한 구문과 연산자가 같이 쓰인다면 가장 낮은 연산 순위를 갖는 연산자 주변에 공백을 추가합니다. 하지만 이 경우에도 불필요하게 2개 이상의 공백을 추가하지 않는 것이 좋습니다.

- 좋은 예

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

- 나쁜 예

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

6.2.2. 함수 인자 및 리턴값 관련 공백

함수 인자의 자료형을 설명하는 콜론과 함수의 리턴 자료형을 명시하는 설명 연산자 (->) 사이에는 공백을 씁니다.

- 좋은 예

```
def foo(input: AnyStr):
    ...

def foo() -> PosInt:
    ...
```

- 나쁜 예

```
def foo(input:AnyStr):
    ...

def foo() ->PosInt:
    ...
```

함수의 키워드 인자값을 지정하기 위한 등호 (=) 주변에는 공백을 사용하지 않습니다.

- 좋은 예

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

- 나쁜 예

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

함수 인자의 설명과 디폴트(기본)값을 동시에 사용하는 경우는 등호 (=) 주변에 공백을 사용합니다.

- 좋은 예

```
def foo(sep: AnyStr = None):
    ...

def foo(input: AnyStr, sep: AnyStr = None, limit=1000):
    ...
```

- 나쁜 예

```
def foo(input: AnyStr=None):
    ...

def foo(input: AnyStr, limit = 1000):
    ...
```

여러개의 문(statement, 명령어)을 한줄에 몰아서 쓰는 것은 추천하지 않습니다.

- 좋은 예

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

- 나쁜 예

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

아주 짧은 if문, for문, while문을 사용할 경우 한줄로 코딩하는 것은 괜찮습니다(코드가 간결해지기 때문입니다). 그러나 다수의 조건과 결합되는 코딩을 하는 경우는 긴 라인을 한줄로 표현하면 안됩니다. 가독성이 매우 떨어지게 됩니다.

- 나쁜 예

```
if foo == 'blah': do_blah_thing()
for x in lst: total += x
while t < 10: t = delay()
```

- 더 나쁜 예

```
if foo == 'blah': do_blah_thing()
else: do_non_blah_thing()

try: something()
finally: cleanup()

do_one(); do_two(); do_three(long, argument,
                             list, like, this)

if foo == 'blah': one(); two(); three()
```

---

7. When to Use Trailing Commans (콤마를 마지막에 사용하는 경우)

파이썬 코딩에서 어떤 문(statement)에서 콤마를 사용하는 것은 선택사항입니다.

다만, 튜플을 만들때 콤마를 사용하는 것은 선택이 아니라 필수 입니다.

일반적으로 명료함을 추구하기 위해 괄호로 묶어서 사용하는 것을 추천합니다.

- 좋은 예

```
FILES = ('setup.cfg',)
```

- 나쁜 예

```
FILES = 'setup.cfg',
```

값을 여러개 나열하거나, 여러개의 인자를 선언하는 등 마지막 콤마가 여러개 이어질 경우, 유지보수 단계에서 값을 추가하거나 삭제하고 디폴트 값을 변경해야 하는 경우가 발생합니다. 이 경우 한줄로 코딩해 놓으면 유지보수가 힘들고 가독성이 떨어지게 됩니다.

이 경우에는 괄호를 분할하여 여러줄에 걸쳐 사용하는게 좋습니다.

- 좋은 예

```
FILES = [
    'setup.cfg',
    'tox.ini',
    ]

initialize(FILES,
           error=True,
           )
```

- 나쁜 예

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

---

8. Comments (주석)

코드의 실행과 모순되는 주석은 없는 것보다 못합니다. 그리고 코드를 변경할 경우 주석을 항상 업데이트 해야 합니다.

코드의 실행과 모순되는 주석은 다음과 같은 예를 들 수 있습니다.

- 더하기 연산을 실행하는 함수에 빼기 함수라고 주석을 다는 경우

```
# Substract two integers
def add(a, b):
    return a + b
```

주석은 반드시 하나의 완성된 문장이어야 합니다. 첫 글자가 소문자로 시작하는 고유 식별자가 아니라면 주석의 첫 글자는 대문자를 사용합니다. 문장의 시작하는 첫 글자는 대문자라는 영어 문법을 생각하면 이해가 쉽게 됩니다.

블록 주석은 하나 이상의 문단으로 구성합니다. 각 문단은 완전한 문장으로 구성되어야 합니다. 그리고 각 문장이 끝날때 마침표를 찍어줍니다. 이것도 영어 문법과 동일합니다.

여러 문장으로 구성된 주석을 연속적으로 구성할 경우 각 문장이 끝나면 2줄 공백을 주는 것이 좋습니다.

여러분이 작성한 주석은 다른 사람들이 쉽게 이해할 수 있도록 작성해야 합니다. 본인 혼자만 알아볼 수 있는 알송달송한 단어나 기호를 사용하는 것은 매우 나쁜 주석입니다.

> 영어를 사용하지 않는 개발자들에게 **PEP**에서 전하는 말
>
> 제발 주석은 영어로 작성해 주세요.
>
> 여러분의 모국어를 아는 사람들만 코드를 읽고 이해할 수 있습니다.
>
> 영어로 작성하지 않은 코드는 읽혀지지 않을 확률이 120% 입니다. ㅠㅠ

8.1. Block Comments (블록 주석)

블록 주석은 코드와 동일한 들여쓰기 위치에 씁니다. 블록 주석은 당연히 코드와 연관된 주변(관련 코드 직전 또는 바로 다음 위치)에 작성합니다.

블록 주석의 개별 문장들은 # 표시로 시작하고 한칸 공백을 준 다음 주석을 작성합니다.

문단 내에서 주석 내용을 구분해야 하는 경우 (블록 주석에서 문단이 2개 이상인 경우)는 # 하나만으로 주석을 만들고 새로운 문단에 해당하는 블록 주석을 작성합니다.

8.2. Inline Comments (문장 내 주석)

문장내 주석은 꼭 필요한 경우에만 사용하도록 합니다.

문장내 주석은 말 그대로 명령어와 주석을 한 줄에 같이 사용하는 주석입니다. 문장 내 주석은 문장(statemnt, 명령어)와 2칸 이상의 공백으로 분리하고 # 표시로 시작합니다.

문장 내 주석은 너무나 자명한 경우에는 사용하지 않는 것이 좋습니다.

- 다음은 \(x\) 값을 1 만큼 증가시키는 코드입니다. 굳이 주석이 필요없는 경우입니다.

```
x = x + 1  # Increment x
```

- 아래와 같은 경우는 값을 증가시키더라도 내부적 의미를 가지고 있다면 사용해도 됩니다

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

8.3. Document Strings (설명문)

Document string은 줄여서 docstring이라고 부르기도 합니다. Docstring은 PEP 257 Docstring Conventions에 별도로 정의되어 있습니다. Docstring은 모듈, 함수, 클래스 또는 메서드를 설명할때 사용합니다. VS Code와 같은 편집기에서 함수나 클래스 이름을 클릭하면 호버링 되거나 팝업되는 설명이 바로 docstring 입니다. 파이썬의 멋진 기능 중 하나입니다.

- PEP 257은 모든 퍼블릭 모듈, 함수, 클래스, 메서드에 작성할 것을 권고하고 있습니다. Non-public (비공개 또는 은닉) 메서드에는 docstring을 작성하지 않아도 됩니다. 하지만 해당 메서드가 하는 역할에 대해서는 docstring으로 작성해 놓아야 합니다.
- Docstring은 def 선언이 있는 바로 다음 줄에 나와야 합니다.
- Docstring은 큰 따옴표 3개 (“””) 또는 작은 따옴표 3개 (‘’’)를 사용하여 정의합니다.
- Docstring이 여러 줄에 걸쳐 작성될 경우 마지막은 따옴표 3개만 나오도록 작성합니다.

- 여러줄에 걸친 docstring 예

```
def foo():
    """Return a foobang
    Optional plotz says to frobnicate the bizbaz first.
    """
    your_code
        :
```

- 한 줄짜리 docstring 예

```
def foo():
    """Return a foobang"""
    your_code
        :
```

---

9. Naming Conventions (이름 짓기)

파이썬에서 이름짓기 규칙은 약간 골치아플 정도록 복잡합니다. 파이썬 개발자는 기존에 존재하는 어떤 이름 짓기 convention이라도 사용할 수 있습니다. 모든 이름 짓기 convention을 모두 다룰 수 없지만 현재 기본적으로 많이 사용하는 이름 짓기 표준에 관련된 사항 위주로 설명하겠습니다.

새롭게 개발되는 모듈이나 패키지는 당연히 여기서 설명하는 이름 짓기 convention 사용을 강력히 권고합니다.

만약, 여러분이 기존에 가지고 있는 코드가 다른 스타일이라면 전체를 뜯어고치지 말고 기존 코드의 convention을 유지하면서 관리하는 것이 좋습니다.

9.1. Overriding Principle (오버라이딩 원칙)

> **Important**
>
> API의 공개된 부분 중 사용자가 볼 수 있는 이름은 어떻게 구현했는지 보다는 어떤 작업을 하는지를 보여주는 것이 중요합니다.

9.2. Descriptive Naming Style (서술적 이름 짓기 스타일)

서술적 이름 짓기 스타일은 어떤 관행이나 규칙을 따르는 것이 아니라 개발자들의 편리에 의해 개별적으로 사용되는 이름 짓기 스타일 입니다. 일정한 규칙이 없다보니 수많은 이름 짓기 스타일이 존재합니다. 대표적 서술적 이름 짓기를 정리하면 다음과 같습니다.

- b (하나의 소문자를 이용해 이름 붙여주기)
- B (하나의 대문자를 이용해 이름 붙여주기)
- lowercase (소문자를 결합해서 이름 붙여주기)
- lower_case_with_underscores (소문자와 언더스코어를 결합해서 이름 붙여주기)
- UPPERCASE (대문자를 결합해서 이름 붙여주기)
- UPPER_CASE_WITH_UNDERSCORES (대문자와 언더스코어를 결합해서 이름 붙여주기)

> **CapWord (캡워드 스타일)**
>
> - CapitalizedWords (이름 중간에 대문자를 적절히 사용하기 때문에 중간이 툭툭 튀어나온 모양을 갖는 이름 짓기 스타일입니다.
> - ‘캡워드(CapWords)’, ‘스터들리캡(StudlyCap)’ 또는 '카멜케이스(CamelCase)'라고 부르기도 합니다.
> - 파이썬에서는 클래스(class) 이름을 지어줄때 사용하는 컨벤션(convention) 입니다.

- mixedCase (캡워드 스타일과 유사하지만 첫 글자를 소문자로 씁니다.)
- Capitalized_Words_With_Underscores (캡워드와 언더스코어 스타일이 혼합된 형태)
  - 가장 난잡하고 복잡해서 사용하는 것을 추천하지 않습니다.

> **Note**
>
> 캡워드 스타일에서 약어 사용 캡워드 스타일을 사용하는 이름 짓기에서 약어를 포함해야 하는 경우에는 약어는 모두 대문자로 쓰는 것이 좋습니다.
>
> - 좋은 예: HTTPServerError
> - 나쁜 예: HttpServerError

연관있는 이름을 그룹화 하기 위하여 독특한 문자를 사용하는 이름 짓기 스타일도 있습니다.

파이썬의 경우 그리 많이 사용하지 않지만, 가끔 사용하는 경우가 있습니다.

예를 들어 os.stat() 함수는 튜플을 리턴하게 되는데 튜플의 이름을 st_mode, st_size, st_mtime 등과 같이 짓는 경우가 있습니다. 이 경우는 관례적으로 사용하는 리턴값과의 일관성을 유지하기 위해 사용하는 경우입니다.

X11 라이브러리는 관습적으로 모든 public 함수의 이름 맨 앞에 X를 붙여주기도 합니다. 하지만 이런 이름 짓기 스타일은 파이썬에서 필수적인 것은 아닙니다. 왜냐하면 속성 및 메서드 이름은 객체이름이 앞에 붙여지고 함수 이름은 모듈 이름이 앞에 붙기 때문입니다.

> **X11**에 대하여
>
> X 윈도우(X Window) 또는 X 윈도 시스템(X Window System, 흔히 X11, X라고 알려져 있음. 문화어: X Window체계)은 주로 유닉스 계열 운영체제에서 사용되는 윈도 시스템 및 X 윈도우 GUI 환경입니다.
>
> X 윈도 시스템은 디스플레이 장치에 창을 표시하며 마우스와 키보드 등의 입력 장치의 상호작용 등을 관리해 GUI 환경의 구현을 위한 기본적인 프레임워크를 제공합니다.
>
> 자료 출처: X윈도 시스템

앞쪽 혹은 뒤쪽에 언더스코어(_)를 사용하는 경우를 볼 수 있습니다. 이 경우는 일반적으로 특수한 목적을 가진 경우에 사용합니다. 몇 가지 예를 들면 다음과 같습니다.

- _single_leading_underscore: 기능을 내부 목적으로 사용하고자 하는 경우에 해당합니다. Java 또는 C++에서 private 멤버와 조금 비슷한 느낌입니다.
- 아래 코드 중 M에 포함된 객체(함수, 클래스, 변수 등) 중에서 하나의 언더스코어(_)가 앞에 붙은 것들은 임포트 되지 않습니다.

```
# Follow line does not import objects whose names start with an underscore.
from M import *
```

- ingle_trailing_underscore_: 파이썬에서 이미 사용하고 있는 키워드와 겹치는 것을 방지하기 위해 이름 마지막에 언더스코어(_)를 붙여줍니다. 아래 예제는 class 라는 이름을 지어주고 싶은데, class는 파이썬의 예약어이므로 사용할 수 없어 맨 끝에 언더스코어를 붙여준 경우입니다.

```
tkinter.Toplevel(master, class_='ClassName')
```

- __double_leading_underscore: 클래스 속성에 이름을 지어줄 경우 언더스코어 2개를 앞에 붙여주면 해당 이름에 맹글링(mangling)이 수행됩니다.
  - FooBar 라는 클래스가 __boo라는 속성을 가지고 있을 경우
  - __boo 속성은 _FooBar__boo로 자동 변환 됩니다.

> 맹글링**(mangling)**
>
> 맹글링은 사전적 의미는 '짓이기다'라는 뜻입니다. 파이썬에서 맹글링은 원래 이름을 짓이겨서 다른 이름으로 바꿔버리는 것을 의미합니다. 맹글링이 왜 필요하냐고요? 두 가지 필요성 때문에 맹글링이 필요합니다.
>
> 1. 클래스 속성값을 외부에서 접근하지 못하도록 할 때 사용합니다. (private 멤버로 만드는 효과)
> 2. 하위 클래스가 상위 클래스 속성을 오버라이딩 하는 것을 막을 때 사용합니다.

- 다음은 클래스 속성값에 접근할 수 있는 예제입니다.

```
class TestClass:
    def __init__(self):
        self.name = "왕춘삼"
        self.age = 30
        self.hobby = "인형놀이"


man = TestClass()
print(man.name, man.age, man.hobby)

# print output
# 왕춘삼 47 인형놀이
```

- 다음은 맹글링을 통해 클래스 속성값에 접근하지 못하는 예제입니다.

```
class TestClass:
    def __init__(self):
        self.name = "왕춘삼"
        self.age = 47
        self.__hobby = "인형놀이"


man = TestClass()
print(man.name, man.age, man.__hobby)
# AttributeError: 'TestClass' object has no attribute '__hobby'
```

이름 앞과 뒤에 더블 언더스코어 (언더스코어 2개, __)가 붙은 경우는 매직 매서드 입니다. 대표적으로 아래와 같은 것들이 있습니다. 절대로 매직 메서드와 같은 이름을 만들지 마세요.

```
__init__
__import__
__file__
    :
    :
```

9.3. Prescriptive Naming Style (관행적 이름 짓기 스타일)

9.3.1. Names to Avoid (피해야할 이름)

영어 대문자 아이(I), 대문자 오(O), 소문자 엘(l)은 단일문자 이름으로 사용하면 절대 안됩니다.

일부 폰트에서는 숫자 1, 0과 구분이 되지 않습니다. 가독성에 매우 큰 문제가 될 수 있습니다.

알파벳 소문자 엘(l)을 꼭 사용해야 한다면, 소문자 대신 대문자 엘(L)을 사용하기 바랍니다.

9.3.2. ASCII Compatibility (아스키 코드 호환성)

파이썬 표준 라이브러리(standard library)에서 사용된 식별자는 반드시 ASCII 코드와 호환성이 있어야 합니다.

다른 이름 짓기에는 다음과 같은 규칙을 적용해야 파이썬이 알아듣습니다.

- 영어 소문자 (a to z)
- 영어 대문자 (A to Z)
- 숫자 (0 to 9)
- 언더스코어 (_)
- 식별자는 숫자로 시작해서는 안됩니다.
- 특수문자(!, @, #, $, %, .)를 사용하면 안됩니다.
- 파이썬 예약어를 사용하면 안됩니다.
- 최대 79 글자를 넘기면 안됩니다.

- 파이썬에서 사용하는 예약어(키워드)는 다음과 같이 확인할 수 있습니다.

```
import keyword
print(keyword.kwlist)
```

```
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class',
'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global',
'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return',
'try', 'while', 'with', 'yield']
```

9.3.3. Package & Module Names (패키지와 모듈의 이름)

모듈의 이름은 모두 소문자로 구성하고 간결하게 짓습니다.

만약 가독성에 도움이 된다면 모듈 이름에 언더스코어를 사용할 수 있습니다.

파이썬 패키지 역시 모두 소문자로 구성하고 간결하게 이름을 지어야 합니다.

파이썬 패키지는 언더스코어 사용하지 않을 것을 추천합니다.

> 패키지와 모듈의 차이
>
> - 모듈: 각종 클래스, 함수, 변수 등을 포함하고 있는 파일입니다. 일반적으로 xxx.py 파일을 의미합니다.
> - 패키지: 특정 기능과 관련된 모듈(xxx.py)들을 묶어놓은 폴더라고 생각하면 됩니다. 참고로 과거 파이썬 3.3 이전 버전에서는 해당 폴더가 패키지인 것을 파이썬에게 알려주기 위해 __init__.py 라는 빈 파일을 폴더에 넣어주어야 했습니다. 파이썬 3.4 이상에서는 __init__.py 라는 빈 파일이 없어도 작동은 하지만 여전히 명시성과 과거 버전과의 호환성을 위해 사용하는 경우도 많습니다. 어떤 폴더에 __init__.py 파일이 있다면, 해당 폴더에 있는 모듈들을 from 또는 import 등을 이용하여 편리하게 불러올 수 있습니다.

> **Note**
>
> __init__.py 또 다른 역할
>
> 특정 모듈이 위치한 디렉토리에 *_init_*.py 파일이 있다면 공통으로 적용 가능한 기능이나 모듈을 포함할 수 있습니다. 예를 들어 __init__.py와 top_module.py 파일이 같이 있고, 하위 디렉토리가 여러개 있을 경우 한번의 선언으로 모두 공통적으로 사용할 수 있습니다.
>
> - top_module.py 모듈이 있는 하위 디렉토리에 aaa.py, bbb.py, ccc.py 라는 모듈이 있다고 가정합니다.
> - aaa.py, bbb.py, ccc.py 3개의 모듈은 모두 base.py 라는 모듈 기능이 필요하다면 각각의 파일 내부에서 import를 해야 합니다.
> - aaa.py, bbb.py, ccc.py 3개 모듈 외에 __init__.py 라는 모듈을 만들고 __init__.py 모듈 내부에서 import base 선언하면 나머지 3개에 파일에서는 별도의 import 없이 base.py를 활용할 수 있습니다.
> - 이와 같은 모듈 초기화를 통해 더 적은 코드를 사용할 수 있고, 결과적으로 효율적인 코드 작성이 가능해 집니다.

9.3.4. Class Names (클래스 이름)

클래스의 이름은 반드시 캡워드 컨벤션을 따라야 합니다.

인터페이스가 정의되어 있고 우선적인 호출 객체일 경우 함수 이름 붙여주는 컨벤션을 적용할 수 있습니다.

> **Note**
>
> - 다양한 빌트인(builtin, 이미 정의된, 사전에 만들어서 내장해 놓은) 이름이 있을 수 있습니다.
> - 대부분의 빌트인 이름은 하나 또는 두 개의 단어가 결합된 형태가 대부분입니다.
> - 빌트인 이름 중 유일하게 캡워드 컨벤션을 사용하는 것은 에러(Exception) 이름과 빌트인 상수입니다.

9.3.5. Type Varialbe Names (자료형 변수 이름)

자료형 변수의 이름은 PEP 484 Type Hints에 정의되어 있습니다. 자료형 변수 이름은 주로 캡워드 컨벤션을 따르고 있으며 짧은 이름을 선호합니다.

T, **AnyStr**, **Num** 등이 자료형의 이름입니다.

자료형 변수는 해당 변수 이름 마지막에 특징을 부여하는 단어를 붙여서 사용하는 것을 추천합니다.

- 아래 예제는 TypeVar 이라는 자료형을 임포트하고 VT_co 라는 이름을 갖는 TypeVar 자료형 변수를 선언한 예시입니다.
- `covariant=True`라는 옵션을 설정하였기 때문에 변수 이름 끝에 **co**를 붙여서 `VT_co` 라는 자료형 변수를 선언했습니다.

```
from typing import TypeVar

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

9.3.6. Exception Names (예외 이름)

예외(`Exception`)는 기본적으로 클래스(class) 입니다. 따라서 클래스 이름 짓기 컨벤션인 캡워드를 사용하면 됩니다.

다만 여러분이 만든 에러 이름이 진짜 에러를 표현하는 것이라면 예외 이름 지어준 마지막에 "Error"라고 붙여주면 좋습니다.

9.3.7. Global Variable Names (전역 변수 이름)

전역변수는 단일 모듈 내에서만 사용할 것을 강력히 권장합니다. 전역변수 하나를 설정하고 여러 모듈에서 사용하면 절대로 안됩니다.

전역 변수를 남용하면 코드 작동 결과가 이상해도 버그를 찾아내는 것이 매우 어렵게 됩니다.

전역변수 이름 짓기는 함수(function)와 동일한 컨벤션을 사용합니다.

모듈의 모든 내용을 불러오는 와일드(`*`) 임포트를 사용할 경우엔는 전역 변수가 같이 `import` 되는 것을 막기 위해 `__all__` 메커니즘을 활용하는 것이 좋습니다.

> **Note**
>
> `__all__` 메커니즘은 해당 디렉터리의 `__init__.py` 파일에 `__all__`이라는 변수를 설정하고 import할 수 있는 모듈을 정의해 주는 방식입니다. `__all__` 변수에 정의해 주지 않는 변수들은 와일드(`*`) 임포트를 해도 불러오지 않습니다.
>
> - `game`이라는 패키지 하위 디렉토리에 `sound.py`라는 모듈이 있고,
> - `sound.py`가 존재하는 하위 디렉토리에 `echo.py`라는 모듈이 있는 상황입니다.
> - `echo.py`는 `echo_test()`라는 함수가 정의되어 있으며, `echo_test()`는 `echo`라는 전역 변수를 출력하는 함수입니다.
> - 만약 다음과 같이 실행하면 에러가 납니다.
>
> ```
> from game.sound import *
> echo.echo_test()
> Traceback (most recent call last):
>     File "<stdin>", line 1, in <module>
> NameError: name 'echo' is not defined
> ```
>
> - 하지만 `__init__.py`에 `__all__`을 다음과 같이 정의할 수 있습니다.
>
> ```
> # __init__.py
> __all__ = ['echo']
> ```
>
> - 다시 위 코드를 실행하면 정상적으로 작동합니다.
>
> ```
> from game.sound import *
> echo.echo_test()
> # >>> Output -> echo
> ```

전역 변수 import를 막는 다른 방법은 앞에서 설명한 것과 같이 이름 앞에 하나의 언더스코어를 사용하는 컨벤션을 사용하는 것입니다.

앞에서 설명한 단일 언더스코어 사용법을 참고하면 됩니다.

9.3.8. Function & Variable Names (함수와 변수 이름)

함수 이름은 소문자로 구성하여 짓습니다.

가독성을 높일 수 있다면 함수 이름 중간에 언더스코어를 사용하는 것도 좋습니다.

변수 이름 짓는 것은 함수 이름 짓는 방법과 동일합니다.

mixedCase 컨벤션을 사용하는 것도 가능합니다. 이미 과거에 `threading.py`와 같은 모듈에서 사용한 적이 있기 때문에 과거 컨벤션과의 호환성을 유지할 수 있습니다.

9.3.9. Method Names & Instance Variables (메서드 이름과 객체 변수)

인스턴스 메서드의 첫 번째 인자는 항상 `self`를 적어 줍니다.

인스턴스 메서드 **(instance method)**

- 가장 흔히 쓰이는 것으로, 인스턴스 변수에 엑세스할 수 있도록 첫 번째 인자에 항상 객체 자신을 의미하는 `self` 파라미터를 갖습니다.(self이외에도 여러개의 파라미터를 가질 수 있습니다.)
- 해당 메서드를 호출한 객체에만 영향을 미칩니다.
- 객체 속성에 접근이 가능하다.
- 호출 방법
  - 해당 클래스 안에서는 `self.메서드명`
  - 클래스 밖에서는 `객체.메서드명`

클래스 메서드의 첫번째 인자는 `cls`를 적어 줍니다.

클래스 메서드 **(class method)**

- `self` 파라미터 대신 `cls`라는 클래스를 의미하는 파라미터를 갖습니다.
- 해당 클래스로 생성된 객체로 부터 호출 되는 것이 아니라, 클래스 자체에서 직접 호출됩니다.
- 객체의 속성/메서드에는 엑세스가 불가능합니다.
  - 그러나, `cls.클래스속성명`으로 클래스 속성에는 접근 가능합니다.
- `cls`를 사용하면 클레스 메서드 내부에서 현재 클래스의 인스턴스를 만들 수도 있다. (`cls() = 현재클래스명()`를 의미합니다.)
- 호출 방법
  - `클래스명.클래스메서드명`
  - `객체명.클래스메서드명`

```
class Person:
    count = 0 # 클래스 속성

    def __init__(self):
        Person.count += 1

    @classmethod
    def print_count(cls):
        # 클래스 속성에는 엑세스가 가능합니다.
        print('{0}명 생성되었습니다.'.format(cls.count))

    @classmethod
    def create(cls):
        # 메서드 내부에서 cls()로 현재 클래스의 인스턴스를 만들 수도 있습니다.
        # 즉, cls() = Person()을 의미합니다.
        p = cls()
        return p

ryan = Person()
apeach = Person()

Person.print_count()
# 결과값 : 2명 생성되었습니다.

print(Person.create())
# 결과값 : <__main__.Person object at 0x000001BA0AE143D0>
# Person클래스로 부터 생성된 인스턴스임을 확인할 수 있습니다.
```

참고 블로그: [Python] 인스턴스 메서드 / 정적 메서드 / 클래스 메서드

파이썬 예약어를 함수 인자 이름으로 지어주게 되면 에러가 나게 됩니다.

예약어와 동일한 인자 이름을 써야 한다면 이름 뒤쪽에 언더스코어를 붙여 줍니다.

예를 들어 `class`라는 인자 이름을 사용해야 한다면 `clss`와 같은 줄임말 보다는 `class_` 같이 이름을 지어주는 것이 훨씬 가독성이 높고 코드 분석을 할때 혼선을 줄일 수 있습니다.

9.3.10. Constants (상수)

상수(constant)는 모듈 레벨에서 정의하고 사용하도록 합니다.

상수는 일반적으로 모두 대문자로 표기합니다.

가독성을 높이기 위해 언더스코어를 적절히 혼합하여 사용할 수 있습니다.

예를 들면 `MAX_OVERFLOW` 또는 `TOTAL` 같은 형태 모두 가능합니다.

9.3.11. Designing for Inheritance (상속 설계)

이 내용은 어렵기 때문에 생략합니다.

좀 더 깊은 이해를 원하는 독자는 Method Names and Instance Variables를 참고하기 바랍니다.

9.4. Public & Internal Interface

이 내용은 어렵기 때문에 생략합니다.

좀 더 깊은 이해를 원하는 독자는 Public and Internal Interfaces를 참고하기 바랍니다.

---

10. Programming Recommendations

- Pfoo2Pfoo2, Jfoo2thon, IronPfoo2thon, Cfoo2thon, Psfoo2co와 같은 다른 파이썬 구현에 방해가 되지 않도록 코딩 스타일을 유지하도록 해야 합니다.
  - 예를 들어 두 개의 문자열 `a`와 `b`가 있는 경우 문자열을 합쳐서 기존의 문자열에 대입하는 **inplace** 연산을 사용하는 것을 자제해야 합니다.
  - `a += b` 또는 `a = a + b`와 같은 연산은 `Cfoo2thon`에서는 속도가 빠를 수 있지만, 다른 파이썬 구현에 모두 적용되지 않을 수 있습니다.
  - 속도가 중요한 고려사항이라면 `''.join()` 형태로 코딩하는 것이 좋습니다.
- `None`과 같은 싱글톤(singleton)와 비교할때는 항상 `is` 또는 `is not`을 사용해야 합니다. 절대로 비교 연산자 `==`를 사용하지 않도록 주의합니다.

> 싱글톤**(singleton)**
>
> 사용자가 여러 번 객체 생성을 하더라도 클래스로부터 오직 하나의 객체만 생성되도록 하는 디자인 패턴입니다.
>
> 이러한 싱글턴 패턴은 오직 유일한 객체를 통해서만 어떤 리소스에 접근해야하는 제약이 있는 상황에서 유용하게 사용할 수 있습니다.
>
> 클래스를 사용하는 입장에서는 실수로 여러 번 객체 생성을 시도하더라도 내부적으로는 오직 하나의 객체만 생성되고 사용됩니다.
>
> 참고 블로그: 레벨업 파이썬 01) 싱글톤 패턴

- 싱글톤 사용 시 예상되는 문제점
  - `if foo1 is not None`을 표현하는 경우에 한해서 `if foo1` 같이 코딩할 수 있습니다.
  - 변수나 함수 인자값이 다른 어떤 값으로 설정된 상태를 테스트하는 과정에서 의도와 상관없이 `if foo1` 구문을 통과할 수 있습니다.
    - `if foo1`는 `if foo1==True`로 해석될 수 있습니다.
    - `None`과 `True`는 다르게 해석되어야 합니다.
- `not ...` 보다는 `is not`을 사용하도록 합니다.
  - 두 가지 모두 문법적으로 맞지만 `is not`이 보다 가독성이 높습니다.
- 좋은 예

```
if foo is not None:
```

- 안좋은 예

```
if not foo is None:
```

10.1. 객체의 순서를 비교하려고 할 때

객체 비교는 프로그래머가 다양한 코딩 방식을 동원해 구현할 수 있습니다. 그러나 대부분은 파이썬 빌트인으로 지원하는 던더(dunder) 메서드를 활용하는 것이 가장 바람직한 파이썬 코딩 스타일입니다. 객체 비교를 위한 빌트인 던더 메서드는 6가지입니다. 구체적으로 다음과 같습니다.

- **lt**: less than (작다), `<`
- **le**: less than or equal (작거나 같다), `<=`
- **gt**: greater than (크다), `>`
- **ge**: greater than or equal (크거나 같다), `>=`
- **eq**: equal (같다), `==`
- **ne**: not equal (같지 않다), `!=`

`FooBar` 라는 클래스가 속성 `value`를 가지고 있을 때

`__eq__` 메서드를 재정의 하여 객체간 비교를 가능하게 할 수 있습니다.

다음은 예시 입니다.

```
class FooBar():
    def __init__(self, value):
        self.value = value

foo1 = FooBar(10)
foo2 = FooBar(20)

# Check built-in members
print(f'Dunder methods in foo1: {dir(foo1)}')
print()
print(f'Dunder methods in foo1: {dir(foo1)}')
```

```
Dunder methods in foo1: ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'value']

Dunder methods in foo1: ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'value']
```

- `foo1`, `foo2` 객체는 모두 `__gt__` 던더 메서드를 가지고 있습니다. 하지만 `greater than (>)` 비교연산을 하면 작동하지 않습니다.

```
foo1 > foo2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_1216014/1785668163.py in <module>
----> 1 foo1 > foo2

TypeError: '>' not supported between instances of 'FooBar' and 'FooBar'
```

- 위 빌트인 멤버들 중에서 `__gt__` 함수를 오버라이딩(재정의) 하면 다음과 같습니다.

```
class FooBar():
    def __init__(self, val) :
        self.value = val

    def __gt__(self, other):
        return self.value >= other.value
```

```
foo1 = FooBar(10)
foo2 = FooBar(20)
```

```
foo1 > foo2
```

```
False
```

실제로 비교 연산자는 다음과 같이 작동하게 됩니다.

1. 연산자 앞쪽 객체(`foo1`)는 자신 메서드 중 해당되는 던더 메서드를 호출하고,
2. 그 던더 메서드의 인자(argument)로 연산자 뒤쪽에 있는 객체(`foo2`)를 전달합니다.
3. 앞쪽 객체(`foo1`)의 해당 던더 메서드는 함수 정의에 따라 결과를 리턴합니다.

```
# fool에서 __lt__ 함수를 호출하면서 인자로 foo2 전달
foo1 < foo2

# fool에서 __gt__ 함수를 호출하면서 인자로 foo2 전달
foo1 <= foo2

# fool에서 __gt__ 함수를 호출하면서 인자로 foo2 전달
foo1 > foo2

# fool에서 __ge__ 함수를 호출하면서 인자로 foo2 전달
foo1 >= foo2

# fool에서 __eq__ 함수를 호출하면서 인자로 foo2 전달
foo1 == foo2

# fool에서 __ne__ 함수를 호출하면서 인자로 foo2 전달
foo1 != foo2
```

Note

`__eq__`, 즉 `==` 연산자는 객체간 비교연산을 객체가 가지고 있는 값들이 모두 같으면 `True`, 다르면 `False`를 리턴합니다. 그러므로 동일한 클래스에서 생성한 객체들은 별도의 `==` 연산자를 별도로 오버라이팅하지 않더라도 정상적으로 작동합니다.

```
>>> foo1 == foo2

False
```

그러나 속성이 여러 개인 경우 그 중에서 특정 값이나 조건을 기준으로 `==` 연산하려면 추가적으로 `__eq__` 메서드를 오버라이딩 해줘야 합니다.

10.2. 단일 식별자 함수 할당

하나의 식별자에 람다(`lambda`) 표현을 사용해 직접적으로 할당하는 방식보다 `def`문을 이용해 명시적으로 함수를 정의해 주어야 합니다.

- 좋은 예

```
def f(x): return 2*x
```

- 나쁜 예

```
f = lambda x: 2*x
```

10.3. 예외 처리 (Exception Handling)

- 예외처리를 할 경우 단순히 에러가 났다는 것만 표시하는 것 보다는 어떤 에러인지 명시적으로 코딩해 주는 것이 좋습니다.

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

위 예제에서 `except ImportError` 대신 `except`만 사용할 경우 `SystemExit`과 `KeyboardInterrupt` 예외를 모두 잡아냅니다. 이 경우 키보드의 `Control-C`를 쳐서 생긴 예외인지 다른 문제가 있는 예외인지 구분하기 어렵게 됩니다. 만약 프로그램에서 발생하는 모든 에러를 잡아내고 싶다면 `except Exception` 구문을 사용하면 됩니다.

- 만약 `try-except` 구문을 사용하려고 한다면 명시적으로 잡아내고 싶은 에러를 최소화하는 것이 좋습니다.
- 좋은 예

```
try:
    value = collection[key]
except KeyError: # Catch only one specific error
    return key_not_found(key)
else:
    return handle_value(value)
```

- 나쁜 예
  - `try` 구문에서 발생할 수 있는 예외가 너무 다양한 경우
  - 정확히 어디에서 예외가 존재하는지 해석이 어려울 수 있습니다.

```
try:
    # Too broad!
    return handle_value(collection[key])
except KeyError:
    # Will also catch KeyError raised by handle_value()
    return key_not_found(key)
```

10.4. 컨텍스트 매니저 (context manager, 주로 with 구문으로 사용)를 사용하는 경우

컴퓨팅 자원을 확보하거나 해제하는 경우가 아니라면 함수나 메서드를 이용하여 실행시키는 것이 좋습니다.

- 좋은 예

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

- 나쁜 예
  - 아래 예제는 `with` 구문을 부적절하게 사용한 예를 보여줍니다.
  - `with` 구문은 해당 컨텍스트 영역의 시작(`__enter__`)과 끝(`__exit__`)을 자동으로 처리해 줍니다. 파일이나 소켓 연결같이 `close()` 메서드가 필요한 코드를 자동으로 처리해 주기 때문에 코딩 에러를 줄여주는 유용한 기능입니다.
  - 다음 예제는 `conn`이 시작되었다는 정보만 알려줄 뿐 어떤 정보도 알 수 없는 구조입니다.
  - 위의 예제는 `conn.begin_transaction()`을 명시적으로 코딩함으로써 어떤 일을 처리하겠다는 건지 명시적으로 나타나지 않습니다.
  - 명시적으로 어떤 기능을 컨택스트 매니저의 해당 블록에서 처리하는 건지에 대하여 표현하는 것이 보다 바람직합니다.

```
with conn:
    do_stuff_in_transaction(conn)
```

10.5. 명시적 리턴(`return`) 표현

- 어떤 함수에서 리턴(`return`)을 사용할 경우 모든 경우에서 명시적으로 리턴이 되도록 코딩하는 것이 바람직합니다.
- 만약 조건문(`if - else`) 중에서 `if`에서 리턴이 있을 경우, 그 이외의 경우(`else`)에 어떤 값을 리턴하는지 명시적으로 표현해야 코드 분석이 용이하고 가독성(readability)가 높아집니다.
- 좋은 예

```
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)
```

- 나쁜 예
  - 아래 예제의 `foo` 함수에서 `if`문이 실행되는 경우 리턴 값을 알지만, 실행되지 않을 경우 어떤 값이 명시적으로 리턴 되는지 알 수 없습니다.
  - `bar` 함수에서 `return math.sqrt(x)`는 `if`문이 실행되는 경우 어떤 값이 리턴 되는지 명시적으로 표현하지 않아 코드 분석에 혼선을 줄 수 있습니다. 위의 '좋은 예'처럼 명시적으로 `None`을 리턴해 주는 것이 좋은 코딩 스타일 입니다.

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

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

10.6. 문자열 슬라이싱(Slicing)을 안전하게 하는 법

문자열에서 특정 문자를 기준으로 앞 또는 뒤쪽으로 슬라이싱을 하는 경우는 `True`/`False` 연산을 통해 처리하기 보다는 `''.startwith()` 또는 `''.endwith()`를 활용하는 것이 좋습니다. `''.startwith()` 또는 `''.endwith()`이 인덱스 기반 슬라이싱보다 깔끔하고 에러가 적은 것으로 알려져 있습니다.

- 좋은 예

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

- 나쁜 예

```
if foo[:3] == 'bar':
```

10.7. 객체의 자료형(type) 비교

객체와 객체 자료형(type)을 비교해야 하는 경우는 직접 비교하는 코딩 스타일 보다는 `isinstance()` 함수를 이용하는 것이 바람직합니다.

- 좋은 예

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

- 나쁜 예

```
if type(obj) is type(1):
```

10.8. 시퀀스 자료형의 Empty 여부 확인

문자열(string), 리스트(list), 튜플(tuple)과 같은 시퀀스 자료형에 데이터가 있는지 여부를 확인할 경우, 빈(empty) 자료형은 `Fasle`를 반환한다는 것을 이용하여 코딩하는 것이 좋습니다.

- 좋은 예

```
if not seq:
if seq:
```

- 나쁜 예

```
if len(seq):
if not len(seq):
```

10.9. 문자열 마지막의 공백 사용

마지막에 지나치게 많은 공백을 덧붙여진 문자열을 사용하는 것을 지양해야 합니다. 가독성이 떨어질 뿐만 아니라 어떤 에디터는 자동으로 문자열 마지막 공백을 잘라내는 경우도 있기 때문에 조심해야 합니다.

10.10. Boolean 자료형의 비교

Boolean 자료를 비교해야 하는 경우 `==` 연산자를 사용하지 않는 것이 좋습니다.

- 좋은 예

```
if greeting:
```

- 나쁜 예

```
if greeting == True:
```

- 아주 나쁜 예

```
if greeting is True:
```

10.11. `try - finally` 구문에서의 흐름제어 명령 사용

`try - finally` 구문 내부에서 `finally` 밖으로 빠져나가는 흐름제어 명령(`return`, `break`, `continue`)을 사용하는 것은 가급적 사용하지 말아야 합니다. 이러한 표현법은 `finally` 구문에서 지속될 수 있는 예외를 묵시적으로 무시할 수 있기 때문입니다.

- 나쁜 예

```
def foo():
    try:
        1 / 0
    finally:
        return 42
```

10.12. Function Annotations

PEP 484 - Type Hints에 따라서 함수 설명 방식이 변경되었습니다.

- 함수 설명은 PEP 484 - Type Hints에서 제시하는 코딩 스타일을 따라야 합니다.
- PEP 8에서 제시하는 함수 설명 방식은 더 이상 사용하지 않을 것을 추천합니다.
- 그러나 표준 라이브러리(`stdlib`) 이외의 코딩에서 PEP 484 - Type Hints 코딩 스타일을 사용하는 것은 추천합니다. 이때 PEP 484 - Type Hints를 사용하는 것이 함수 설명을 추가하는 것에 필요한 노력이나 가독성 증대를 검토한 후 사용할 것을 추천합니다.
- 파이썬 표준 라이브러리에서 PEP 484 - Type Hints와 같은 코딩 스타일(함수 설명)을 채택하는 것에는 아직 논란이 있습니다.
- 그러나 새로운 개발을 위한 코딩이나 대규모 리팩토링(refactoring)이 필요한 경우에 사용하는 것은 가능합니다.
- PEP 484 - Type Hints에서 제시하는 함수 설명 방식을 다른 용도로 사용하고 싶다면 아래와 같은 코드를 소스파일 시작 부분에 삽입해 주면 됩니다.

```
# type: ignore
```

위와 같은 코드는 PEP 484 - Type Hints에서 제시하는 모든 함수 설명과 관련된 체크 기능을 사용하지 않도록 해줍니다.

린터(linter)와 같은 것들을 이용해 함수 주석에 대한 정확성을 확인하는 것은 선택사항입니다. 파이선 기본 인터프리터는 기본적으로 함수 설명에 대한 코딩 스타일에 대해서는 어떠한 에러도 발생시키지 않습니다.

> **Linter란?**
>
> - Online wiki: lint는 컴퓨터 프로그래밍에서 의심스럽거나, 에러를 발생하기 쉬운 코드에 표시(flag)를 달아 놓는 것을 말합니다. 원래는 C 언어에서 사용하던 용어였으나 지금은 다른 언어에서도 일반적으로 사용됩니다.
> - python으로 작성된 모듈, 패키지 등을 손쉽게 PEP8 스타일 가이드 및 구문에러 등을 통해 분석해 채점하고 올바르게 수정할 수 있도록 도와주는 작은 프로그램입니다. VS code의 경우 Extention을 이용해 쉽게 설치하고 사용할 수 있다. 대표적인 익스텐션은 `[pylint](https://pylint.org/)`이 있다.

10.13. Variable Annotations

변수에 대한 설명 방법은 PEP 526 - Syntax for Variable Annotations 에 별도로 정의되어 있습니다.

- 변수 설명 방법은 꽤 복잡한 부분이 있어서 초기 학습자들에게 혼선을 줄 수 있어 구체적 내용에 대한 설명은 생략합니다.
- 변수 설명에 대한 자세한 내용은 PEP 526을 참고하기 바랍니다.

By Giseop Noh
Last updated on 03 Jan 2022.

본 책자에 적용되는 라이센스는 Creative Commons Attribution 4.0 International .