PEP: 8
Title: Style Guide for Python Code
타이틀: 파이썬 코드를위한 스타일 가이드
Version: $Revision$
Last-Modified: $Date$
Author: Guido van Rossum guido@python.org,
Barry Warsaw barry@python.org,
Nick Coghlan ncoghlan@gmail.com
Status: Active
Type: Process
Content-Type: text/x-rst
Created: 05-Jul-2001
Post-History: 05-Jul-2001, 01-Aug-2013

Introduction

This document gives coding conventions for the Python code comprising the standard library in the main Python distribution. Please see the companion informational PEP describing style guidelines for the C code in the C implementation of Python [1]_.

이 문서는 기본 Python 배포판에 표준 라이브러리를 포함하는 Python 코드에 대한 코딩 규칙을 제공합니다. Python의 C 구현에서 C 코드에 대한 스타일 지침을 설명하는 정보 PEP를 참조하십시오 [1] .

This document and PEP 257 (Docstring Conventions) were adapted from Guido's original Python Style Guide essay, with some additions from Barry's style guide [2]_.

이 문서와 PEP 257 (Docstring Conventions)은 Guido의 원래 Python 스타일 가이드 에세이에서 수정되었으며 Barry의 스타일 가이드 [2]에 추가되었습니다 .

This style guide evolves over time as additional conventions are identified and past conventions are rendered obsolete by changes in the language itself.

이 스타일 가이드는 추가 규칙이 식별되고 언어 자체의 변경으로 인해 과거 규칙이 더 이상 사용되지 않으므로 시간이 지남에 따라 발전합니다.

Many projects have their own coding style guidelines. In the event of any conflicts, such project-specific guides take precedence for that project.

많은 프로젝트에는 자체 코딩 스타일 지침이 있습니다. 충돌이있는 경우 해당 프로젝트 별 가이드가 해당 프로젝트에 우선합니다.

A Foolish Consistency is the Hobgoblin of Little Minds

어리석은 일관성은 작은 마음의 홉고블린이다.

One of Guido's key insights is that code is read much more often than it is written. The guidelines provided here are intended to improve the readability of code and make it consistent across the wide spectrum of Python code. As PEP 20 says, "Readability counts".

Guido의 주요 통찰력 중 하나는 코드가 작성된 것보다 훨씬 자주 읽히는 것입니다. 여기에 제공된 지침은 코드의 가독성을 향상시키고 광범위한 Python 코드에서 일관성을 유지하기위한 것입니다. 으로 PEP (20)는 "가독성 수를"말한다.

A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is the most important.

스타일 가이드는 일관성에 관한 것입니다. 이 스타일 가이드와의 일관성이 중요합니다. 프로젝트 내 일관성이 더 중요합니다. 하나의 모듈 또는 기능 내에서 일관성이 가장 중요합니다.

However, know when to be inconsistent -- sometimes style guide recommendations just aren't applicable. When in doubt, use your best judgment. Look at other examples and decide what looks best. And don't hesitate to ask!

그러나 일관성이없는시기를 알고 때로는 스타일 가이드 권장 사항을 적용 할 수없는 경우도 있습니다. 확실하지 않은 경우 최선의 판단을하십시오. 다른 예를보고 가장 적합한 것을 결정하십시오. 주저하지 말고 물어보십시오!

In particular: do not break backwards compatibility just to comply with this PEP!

특히,이 PEP를 준수하기 위해 이전 버전과의 호환성을 중단하지 마십시오!

Some other good reasons to ignore a particular guideline:

특정 지침을 무시해야 할 몇 가지 다른 이유 :

1. When applying the guideline would make the code less readable, even for someone who is used to reading code that follows this PEP.

이 지침을 적용하면이 PEP를 따르는 코드를 읽는 데 익숙한 사람에게도 코드를 읽을 수 없게됩니다.

2. To be consistent with surrounding code that also breaks it (maybe for historic reasons) -- although this is also an opportunity to clean up someone else's mess (in true XP style).

역사적인 코드와도 관련이있는 주변 코드와 일관성을 유지하기 위해 (역사적인 이유로) 다른 사람의 혼란을 제거 할 수도 있습니다 (진정한 XP 스타일).

3. Because the code in question predates the introduction of the guideline and there is no other reason to be modifying that code.

문제의 코드가 가이드 라인을 소개하기 이전에 해당 코드를 수정해야 할 다른 이유가 없기 때문입니다.

4. When the code needs to remain compatible with older versions of Python that don't support the feature recommended by the style guide.

코드가 스타일 가이드에서 권장하는 기능을 지원하지 않는 이전 버전의 Python과 호환되어야하는 경우.

Code Lay-out

코드 레이아웃

Indentation

들여 쓰기

Use 4 spaces per indentation level.

들여 쓰기 레벨 당 4 개의 공백을 사용하십시오.

Continuation lines should align wrapped elements either vertically using Python's implicit line joining inside parentheses, brackets and braces, or using a *hanging indent* [#fn-hi]_. When using a hanging indent the following should be considered; there should be no arguments on the first line and further indentation should be used to clearly distinguish itself as a continuation line.

연속 선은 괄호, 괄호 및 중괄호 안에 파이썬의 암시 적 선 결합을 사용하거나 매달린 들여 쓰기를 사용하여 줄 바꿈 된 요소를 수직으로 정렬해야합니다 [7] . 매달린 들여 쓰기를 사용할 때 다음을 고려해야합니다. 첫 번째 줄에는 인수가 없어야하며 연속 줄로 명확하게 구분하기 위해 들여 쓰기를 추가로 사용해야합니다.

Yes::

```
# Aligned with opening delimiter.
# 여는 분리 문자와 일치합니다.
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.
# 4 개의 공백 (추가 들여 쓰기 수준)을 추가하여 인수와 나머지 인수를 구분하십시오.
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)

# Hanging indents should add a level.
# 매달린 들여 쓰기는 레벨을 추가해야합니다.
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

No::

```
# Arguments on first line forbidden when not using vertical alignment.
# 세로 정렬을 사용하지 않을 때는 첫 줄의 인수가 금지됩니다.
foo = long_function_name(var_one, var_two,
    var_three, var_four)
```

```
# Further indentation required as indentation is not distinguishable.
# 들여 쓰기로 필요한 추가 들여 쓰기는 구별 할 수 없습니다.
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

The 4-space rule is optional for continuation lines.
4 줄 규칙은 연속 선의 경우 선택 사항입니다.

Optional::

```
# Hanging indents *may* be indented to other than 4 spaces.
# 매달린 들여 쓰기는 4 칸 이외의 공간에 들여 쓰기 될 수 있습니다.
foo = long_function_name(
  var_one, var_two,
  var_three, var_four)
```

.. _multiline if-statements:

When the conditional part of an `if`-statement is long enough to require that it be written across multiple lines, it's worth noting that the combination of a two character keyword (i.e. `if`), plus a single space, plus an opening parenthesis creates a natural 4-space indent for the subsequent lines of the multiline conditional. This can produce a visual conflict with the indented suite of code nested inside the `if`-statement, which would also naturally be indented to 4 spaces. This PEP takes no explicit position on how (or whether) to further visually distinguish such conditional lines from the nested suite inside the `if`-statement. Acceptable options in this situation include, but are not

`if` 문의 조건부 부분이 여러 줄에 걸쳐 쓰여질만큼 충분히 길면 두 문자 키워드 (예 : `if`)와 단일 공백, 여는 괄호의 조합이 자연스럽게 생성 된다는 점에 주목할 가치가 있습니다. 멀티 라인 조건부의 후속 라인에 대해 4 칸 들여 쓰기. 이는 내부에 중첩 코드의 오목 스위트 시각적 충돌 생성 할 수 있는 경우 도 당연히 4 개 공간으로 들여 쓰기를 -statement 것이다. 이 PEP는 더 시각적으로 내부에 중첩 된 스위트 룸에서 조건부 라인을 구별하는 방법 (여부)에 대한 명시적인 입장도 취하지 않는다 경우 -statement을. 이 상황에서 허용되는 옵션은 다음과 같습니다.

limited to::

```
# No extra indentation.
# 추가 들여 쓰기가 없습니다.
if (this_is_one_thing and
    that_is_another_thing):
    do_something()

# Add a comment, which will provide some distinction in editors
# 편집자를 구별 할 수있는 주석 추가
# supporting syntax highlighting.
# 구문 강조를 지원합니다.
```

```
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.
# 조건부 연속 줄에 들여 쓰기를 추가하십시오.
if (this_is_one_thing
        and that_is_another_thing):
    do_something()
```

(Also see the discussion of whether to break before or after binary operators below.)

(아래의 이진 연산자 앞이나 뒤를 나눌 지에 대한 설명도 참조하십시오.)

The closing brace/bracket/parenthesis on multiline constructs may either line up under the first non-whitespace character of the last line of list, as in::

여러 줄로 구성된 구문에서 닫는 중괄호 / 괄호 / 괄호는 다음과 같이 목록의 마지막 줄의 첫 번째 비 공백 문자 아래에 정렬 될 수 있습니다.

```
my_list = [
    1, 2, 3,
    4, 5, 6,
    ]
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
    )
```

or it may be lined up under the first character of the line that starts the multiline construct, as in::

또는 다음과 같이 여러 줄 구문을 시작하는 줄의 첫 문자 아래에 줄을 그을 수 있습니다.

```
my_list = [
    1, 2, 3,
    4, 5, 6,
]
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
)
```

Tabs or Spaces?

탭 또는 공백?

Spaces are the preferred indentation method.

공백이 선호되는 들여 쓰기 방법입니다.

Tabs should be used solely to remain consistent with code that is already indented with tabs.

탭은 이미 탭으로 들여 쓰기 된 코드와 일관성을 유지하기 위해서만 사용해야합니다.

Python 3 disallows mixing the use of tabs and spaces for indentation.

Python 3에서는 들여 쓰기를 위해 탭과 공백을 혼합하여 사용할 수 없습니다.

Python 2 code indented with a mixture of tabs and spaces should be converted to using spaces exclusively.

탭과 공백이 혼합 된 들여 쓰기 된 Python 2 코드는 공백 만 사용하도록 변환해야합니다.

When invoking the Python 2 command line interpreter with the `-t` option, it issues warnings about code that illegally mixes tabs and spaces. When using `-tt` these warnings become errors. These options are highly recommended!

Python 2 명령 행 인터프리터를 -t 옵션 과 함께 호출하면 탭과 공백을 잘못 혼합하는 코드에 대한 경고가 발행됩니다. -tt를 사용하면 이러한 경고가 오류가됩니다. 이러한 옵션을 적극 권장합니다!

Maximum Line Length

최대 라인 길이

Limit all lines to a maximum of 79 characters.

모든 줄을 최대 79 자로 제한하십시오.

For flowing long blocks of text with fewer structural restrictions (docstrings or comments), the line length should be limited to 72 characters.

구조적 제한이 적은 긴 텍스트 블록 (문서 문자열 또는 주석)을 줄이려면 줄 길이를 72 자로 제한해야합니다.

Limiting the required editor window width makes it possible to have several files open side-by-side, and works well when using code review tools that present the two versions in adjacent columns.

필요한 편집기 창 너비를 제한하면 여러 파일을 나란히 열 수 있으며 인접한 열에 두 버전을 표시하는 코드 검토 도구를 사용할 때 잘 작동합니다.

The default wrapping in most tools disrupts the visual structure of the code, making it more difficult to understand. The limits are chosen to avoid wrapping in editors with the window width set to 80, even if the tool places a marker glyph in the final column when wrapping lines. Some web based tools may not offer dynamic line wrapping at all.

대부분의 도구에서 기본 줄 바꿈은 코드의 시각적 구조를 방해하므로 이해하기가 더 어렵습니다. 줄을 줄 바꿈 할 때 도구가 최종 열에 마커 글리프를 배치하더라도 창 너비가 80으로 설정된 편집기에서 줄 바꿈을 피하기 위해 한계가 선택됩니다. 일부 웹 기반 도구는 동적 줄 바꿈을 전혀 제공하지 않을 수 있습니다.

Some teams strongly prefer a longer line length. For code maintained exclusively or primarily by a team that can reach agreement on this issue, it is okay to increase the line length limit up to 99 characters, provided that

comments and docstrings are still wrapped at 72 characters.

일부 팀은 더 긴 줄 길이를 선호합니다. 이 문제에 대해 합의에 도달 할 수있는 팀이 독점적으로 또는 주로 유지 관리하는 코드의 경우 주석 및 문서 문자열이 여전히 72 자로 줄 바꿈되어 있으면 줄 길이 제한을 99 자까지 늘릴 수 있습니다.

The Python standard library is conservative and requires limiting lines to 79 characters (and docstrings/comments to 72).

파이썬 표준 라이브러리는 보수적이며 줄을 79 자로 제한해야합니다 (docstrings / comments는 72로 제한).

The preferred way of wrapping long lines is by using Python's implied line continuation inside parentheses, brackets and braces. Long lines can be broken over multiple lines by wrapping expressions in parentheses. These should be used in preference to using a backslash for line continuation.

긴 줄을 줄 바꿈하는 가장 좋은 방법은 괄호, 괄호 및 중괄호 안에 파이썬의 묵시적 줄 연속을 사용하는 것입니다. 긴 줄은 식을 괄호로 묶어 여러 줄로 나눌 수 있습니다. 라인 연속에 백 슬래시를 사용하는 것보다 우선적으로 사용해야합니다.

Backslashes may still be appropriate at times. For example, long, multiple `with`-statements cannot use implicit continuation, so backslashes are acceptable::

백 슬래시는 여전히 적절한 경우가 있습니다. 예를 들어, 긴 -statements를 가진 다중 은 암시 적 연속을 사용할 수 없으므로 백 슬래시가 허용됩니다.

```
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())
```

(See the previous discussion on `multiline if-statements`_ for further thoughts on the indentation of such multiline `with`-statements.)

(에 이전 논의를 참조 하면 명령문 `multiline if-statements`_ 같은 여러 줄의 들여 쓰기에 대한 자세한 생각 `with`-statements.)

Another such case is with `assert` statements.

또 다른 경우는 `assert` 진술입니다.

Make sure to indent the continued line appropriately.

연속 줄을 적절히 들여 쓰십시오.

Should a Line Break Before or After a Binary Operator?

이진 연산자 전후에 줄 바꿈이 필요한가요?

For decades the recommended style was to break after binary operators. But this can hurt readability in two ways: the operators tend to get scattered across different columns on the screen, and each operator is moved

away from its operand and onto the previous line. Here, the eye has to do extra work to tell which items are added and which are subtracted::

수십 년 동안 권장되는 스타일은 이진 연산자를 따르는 것이 었습니다. 그러나 이것은 두 가지 방식으로 가독성을 떨어 뜨릴 수 있습니다. 연산자는 화면의 다른 열에 흩어져 있고 각 연산자는 피연산자에서 이전 줄로 이동합니다. 여기서 눈은 어떤 항목이 추가되고 어떤 항목이 감산되는지 알기 위해 추가 작업을 수행해야합니다.

```
# No: operators sit far away from their operands
# 아니오 : 연산자는 피연산자와 멀리 떨어져 있습니다.
income = (gross_wages +
          taxable_interest +
          (dividends - qualified_dividends) -
          ira_deduction -
          student_loan_interest)
```

To solve this readability problem, mathematicians and their publishers follow the opposite convention. Donald Knuth explains the traditional rule in his *Computers and Typesetting* series: "Although formulas within a paragraph always break after binary operations and relations, displayed formulas always break before binary operations" [3]_.

이 가독성 문제를 해결하기 위해 수학자와 출판사는 반대되는 규칙을 따릅니다. Donald Knuth는 자신의 컴퓨터 및 조판 시리즈 에서 전통적인 규칙을 설명합니다 . "단락 내의 수식은 항상 이진 연산 및 관계 후에 끊어 지지만 표시된 수식은 항상 이진 연산 전에 끊어집니다" [3] .

Following the tradition from mathematics usually results in more readable code::

수학의 전통을 따르면 일반적으로 더 읽기 쉬운 코드가 생성됩니다.

```
# Yes: easy to match operators with operands
# 예 : 피연산자와 연산자를 쉽게 일치
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

In Python code, it is permissible to break before or after a binary operator, as long as the convention is consistent locally. For new code Knuth's style is suggested.

파이썬 코드에서는 규칙이 로컬로 일관성이있는 한 이진 연산자 전후에 중단 할 수 있습니다. 새로운 코드의 경우 크 누스의 스타일을 제안한다.

Blank Lines

빈 줄

Surround top-level function and class definitions with two blank lines.

두 개의 빈 라인 최상위 함수 및 클래스 정의를 둘러싸고 있습니다.

Method definitions inside a class are surrounded by a single blank line.

클래스 내부의 메소드의 정의는 하나의 빈 줄에 둘러싸여 있습니다.

Extra blank lines may be used (sparingly) to separate groups of related functions. Blank lines may be omitted between a bunch of related one-liners (e.g. a set of dummy implementations).

빈 줄은 관련 기능을 별도의 그룹 (아껴서)을 사용할 수있다. 빈 줄은 다수의 관련 단일 라이너들 사이에서 생략 될 수있다 (예를 들어, 더미 구현들의 세트).

Use blank lines in functions, sparingly, to indicate logical sections.

논리적 섹션을 표시하기 위해, 드물게, 함수에 빈 줄을 사용합니다.

Python accepts the control-L (i.e. ^L) form feed character as whitespace; Many tools treat these characters as page separators, so you may use them to separate pages of related sections of your file. Note, some editors and web-based code viewers may not recognize control-L as a form feed and will show another glyph in its place.

파이썬은 control-L (즉 ^ L) 폼 피드 문자를 공백으로 받아들입니다. 많은 도구가 이러한 문자를 페이지 구분 기호로 취급하므로 파일의 관련 섹션 페이지를 구분하는 데 사용할 수 있습니다. 일부 편집기 및 웹 기반 코드 뷰어는 control-L을 양식 피드로 인식하지 못할 수 있으며 대신 다른 글리프를 표시합니다.

Source File Encoding

소스 파일 인코딩

Code in the core Python distribution should always use UTF-8 (or ASCII in Python 2).

핵심 Python 배포판의 코드는 항상 UTF-8 (또는 Python 2의 ASCII)을 사용해야합니다.

Files using ASCII (in Python 2) or UTF-8 (in Python 3) should not have an encoding declaration.

ASCII (Python 2) 또는 UTF-8 (Python 3)을 사용하는 파일에는 인코딩 선언이 없어야합니다.

In the standard library, non-default encodings should be used only for test purposes or when a comment or docstring needs to mention an author name that contains non-ASCII characters; otherwise, using `\x`, `\u`, `\U`, or `\N` escapes is the preferred way to include non-ASCII data in string literals.

표준 라이브러리에서 기본이 아닌 인코딩은 테스트 목적으로 만 또는 주석 또는 문서 문자열에 ASCII가 아닌 문자가 포함 된 작성자 이름을 언급해야하는 경우에만 사용해야합니다. 그렇지 않으면 `\x` , `\u` , `\U` 또는 `\N` 이스케이프를 사용하는 것이 ASCII가 아닌 데이터를 문자열 리터럴에 포함시키는 데 선호되는 방법입니다.

For Python 3.0 and beyond, the following policy is prescribed for the standard library (see PEP 3131): All identifiers in the Python standard library MUST use ASCII-only identifiers, and SHOULD use English words wherever feasible (in many cases, abbreviations and technical terms are used which aren't English). In addition, string literals and comments must also be in ASCII. The only exceptions are (a) test cases testing the non-ASCII features, and (b) names of authors. Authors whose names are not based on the Latin alphabet (latin-1, ISO/IEC 8859-1 character set) MUST provide a transliteration of their names in this character set.

Python 3.0 이상의 경우 표준 라이브러리에 대해 다음 정책이 규정됩니다 (PEP 3131 참조). Python 표준 라이브러리의 모든 식별자는 반드시 ASCII 전용 식별자를 사용해야하며 가능한 경우 영어 단어를 사용해야합니다 (대부분의 경우 약어 및 기술) 영어가 아닌 용어가 사용됨). 또한 문자열 리터럴 및 주석도 ASCII 여야합니다. 단, (a) 비 ASCII 기능을 테스트하는 테스트 사례 및 (b) 저자 이름은 예외입니다. 라틴 알파벳 (latin-1, ISO / IEC 8859-1 문자 세트)을 기반으로하지 않는 저자는 반드시이 문자 세트로 이름을 음역해야합니다.

Open source projects with a global audience are encouraged to adopt a similar policy.

전 세계 잠재 고객이있는 오픈 소스 프로젝트는 유사한 정책을 채택하는 것이 좋습니다.

Imports

- Imports should usually be on separate lines::
  import 대개 별도의 줄에 있어야합니다.

  ```
  Yes: import os
       import sys

  No:  import sys, os
  ```

  It's okay to say this though::
  그래도 이렇게 말할 수 있습니다.

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

- Imports are always put at the top of the file, just after any module comments and docstrings, and before module globals and constants.

  import 는 항상 파일의 맨 위에, 모듈 주석 및 문서 문자열 바로 다음과 모듈 전역 및 상수 바로 앞에 배치됩니다.

  Imports should be grouped in the following order:

  수입품은 다음 순서로 그룹화해야합니다.

  1. Standard library imports.
     표준 라이브러리 가져 오기.

  2. Related third party imports.
     관련 제 3 자 수입품.

  3. Local application/library specific imports.
     로컬 애플리케이션 / 라이브러리 특정 가져 오기

  You should put a blank line between each group of imports.
  각 import 그룹 사이에 빈 줄을 넣어야합니다.

- Absolute imports are recommended, as they are usually more readable and tend to be better behaved (or at least give better error messages) if the import system is incorrectly configured (such as when a directory inside a package ends up on `sys.path`)::

가져 오기 시스템이 잘못 구성된 경우 (예 : 패키지 내의 디렉토리가 `sys.path` 에있는 경우) 일반적으로 더 읽기 쉽고 더 나은 동작 (또는 적어도 더 나은 오류 메시지)을 나타내는 경향이 있으므로 절대 가져 오기가 권장됩니다 .

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

However, explicit relative imports are an acceptable alternative to absolute imports, especially when dealing with complex package layouts where using absolute imports would be unnecessarily verbose::

그러나 절대적 가져 오기를 사용하는 것이 불필요하게 장황한 복잡한 패키지 레이아웃을 처리 할 때 명시 적 상대 가져 오기는 절대 가져 오기에 대한 대체 대안입니다.

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

Standard library code should avoid complex package layouts and always use absolute imports.

표준 라이브러리 코드는 복잡한 패키지 레이아웃을 피하고 항상 절대 가져 오기를 사용해야합니다.

Implicit relative imports should *never* be used and have been removed in Python 3.

암시 적 상대적 가져 오기는 절대 사용 해서는 안되며 Python 3에서 제거되었습니다.

- When importing a class from a class-containing module, it's usually okay to spell this::

  클래스가 포함 된 모듈에서 클래스를 가져올 때 일반적으로 다음과 같이 철자를 입력해도됩니다.

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

  If this spelling causes local name clashes, then spell them explicitly::

  이 철자가 로컬 이름 충돌을 일으키는 경우 명시 적으로 철자를 입력하십시오.

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

  and use "myclass.MyClass" and "foo.bar.yourclass.YourClass".

"myclass.MyClass"및 "foo.bar.yourclass.YourClass"를 사용하십시오.

- Wildcard imports (`from <module> import *`) should be avoided, as they make it unclear which names are present in the namespace, confusing both readers and many automated tools. There is one defensible use case for a wildcard import, which is to republish an internal interface as part of a public API (for example, overwriting a pure Python implementation of an interface with the definitions from an optional accelerator module and exactly which definitions will be overwritten isn't known in advance).

  와일드 카드 가져 오기 (''from import *``)는 네임 스페이스에 어떤 이름이 있는지 명확하지 않으므로 독자와 많은 자동화 도구를 혼동하기 때문에 피해야합니다. 와일드 카드 가져 오기에는 방어적인 사용 사례가 하나 있는데, 공개 API의 일부로 내부 인터페이스를 다시 게시하는 것입니다 (예 : 선택적 가속기 모듈의 정의를 사용하여 인터페이스의 순수 Python 구현을 덮어 쓰고 정확히 어떤 정의를 정의할지) 덮어쓰기는 미리 알려져 있지 않습니다).

  When republishing names this way, the guidelines below regarding public and internal interfaces still apply.

  이런 식으로 이름을 다시 게시 할 때 공개 및 내부 인터페이스에 관한 아래 지침이 여전히 적용됩니다.

Module Level Dunder Names

모듈 레벨 던더 이름

Module level "dunders" (i.e. names with two leading and two trailing underscores) such as `__all__`, `__author__`, `__version__`, etc. should be placed after the module docstring but before any import statements *except* `from __future__` imports. Python mandates that future-imports must appear in the module before any other code except docstrings::

`__all__` , `__author__` , `__version__` 등과 같은 모듈 수준 "dunders"(즉, 두 개의 선행 및 두 개의 밑줄이있는 이름) 는 모듈 docstring 뒤에 있지만 `from__future__` imports를 제외한 모든 import 문 앞에 배치해야합니다 . 파이썬은 미래 수입을 문서화 문자열을 제외한 다른 코드보다 먼저 모듈에 표시해야합니다.

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

This module does stuff.
"""

"""이 예제 모듈입니다.

이 모듈은 작업을 수행합니다.
"""

from __future__ import barry_as_FLUFL

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

import os
import sys
```

String Quotes

문자열 따옴표

In Python, single-quoted strings and double-quoted strings are the same. This PEP does not make a recommendation for this. Pick a rule and stick to it. When a string contains single or double quote characters, however, use the other one to avoid backslashes in the string. It improves readability.

파이썬에서 작은 따옴표로 묶인 문자열과 큰 따옴표로 묶인 문자열은 같습니다. 이 PEP는이를 권장하지 않습니다. 규칙을 고르십시오. 그러나 문자열에 작은 따옴표 나 큰 따옴표가 포함 된 경우 다른 문자열을 사용하여 문자열에서 백 슬래시를 피하십시오. 가독성을 향상시킵니다.

For triple-quoted strings, always use double quote characters to be consistent with the docstring convention in PEP 257.

삼중 따옴표로 묶인 문자열의 경우 항상 PEP 257 의 docstring 규칙과 일치하도록 큰 따옴표 문자를 사용하십시오 .

Whitespace in Expressions and Statements

식과 문장의 공백

Pet Peeves

펫 피브스

Avoid extraneous whitespace in the following situations:

다음과 같은 상황에서는 불필요한 공백을 피하십시오.

- Immediately inside parentheses, brackets or braces. ::

괄호, 괄호 또는 중괄호 안에 즉시 있습니다.

```
Yes: spam(ham[1], {eggs: 2})
No:  spam( ham[ 1 ], { eggs: 2 } )
```

- Between a trailing comma and a following close parenthesis. ::

후행 쉼표와 다음 닫는 괄호 사이.

```
Yes: foo = (0,)
No:  bar = (0, )
```

- Immediately before a comma, semicolon, or colon::

쉼표, 세미콜론 또는 콜론 바로 앞에 :

```
Yes: if x == 4: print x, y; x, y = y, x
No:  if x == 4 : print x , y ; x , y = y , x
```

- However, in a slice the colon acts like a binary operator, and should have equal amounts on either side (treating it as the operator with the lowest priority). In an extended slice, both colons must have the same amount of spacing applied. Exception: when a slice parameter is omitted, the space is omitted.

그러나 슬라이스에서 콜론은 이항 연산자처럼 작동하며 양쪽에 동일한 양이 있어야합니다 (우선 순위가 가장 낮은 연산자로 처리). 확장 슬라이스에서 두 콜론에는 동일한 양의 간격이 적용되어야합니다. 예외 : 슬라이스 매개 변수를 생략하면 공백이 생략됩니다.

Yes::

```
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]
```

No::

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

- Immediately before the open parenthesis that starts the argument list of a function call::

  함수 호출의 인수 목록을 시작하는 열린 괄호 바로 직전에 :

  ```
  Yes: spam(1)
  No:  spam (1)
  ```

- Immediately before the open parenthesis that starts an indexing or slicing::

  인덱싱 또는 슬라이싱을 시작하는 열린 괄호 바로 앞 :

```
Yes: dct['key'] = lst[index]
No:  dct ['key'] = lst [index]
```

- More than one space around an assignment (or other) operator to align it with another.

할당 (또는 다른) 연산자 주위에 둘 이상의 공백을 배치하여 다른 연산자와 정렬합니다.

Yes::

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

No::

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

Other Recommendations

다른 추천

- Avoid trailing whitespace anywhere. Because it's usually invisible, it can be confusing: e.g. a backslash followed by a space and a newline does not count as a line continuation marker. Some editors don't preserve it and many projects (like CPython itself) have pre-commit hooks that reject it.

  어디서나 공백을 피하십시오. 일반적으로 보이지 않기 때문에 혼동 될 수 있습니다. 예를 들어 백 슬래시 뒤에 공백이 있고 줄 바꾸기는 줄 연속 표시 자로 계산되지 않습니다. 일부 편집자는이를 보존하지 않으며 CPython 자체와 같은 많은 프로젝트에는이를 거부하는 사전 커미트 후크가 있습니다.

- Always surround these binary operators with a single space on either side: assignment (`=`), augmented assignment (`+=`, `-=` etc.), comparisons (`==`, `<`, `>`, `!=`, `<>`, `<=`, `>=`, `in`, `not in`, `is`, `is not`), Booleans (`and`, `or`, `not`).

  항상 양쪽에 하나의 공간이 바이너리 연산자를 둘러싸고 : 할당 (`=`), 증강 할당 (`+=` , `-=` 등), 비교 (`==` , `<` , `>` , `!=` , `<>` , `<=` , `>=` , `in` , `not in`, `is`, `is not`), Booleans (`and`, `or`, `not`).

- If operators with different priorities are used, consider adding whitespace around the operators with the lowest priority(ies). Use your own judgment; however, never use more than one space, and always have the same amount of whitespace on both sides of a binary operator.

  우선 순위가 다른 연산자를 사용하는 경우 우선 순위가 가장 낮은 연산자 주위에 공백을 추가하십시오. 자신의 판단을 사용하십시오. 그러나 둘 이상의 공백을 사용하지 말고 항상 이항 연산자의 양쪽에 같은 양의 공백이 있어야합니다.

Yes::

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

No::

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

- Function annotations should use the normal rules for colons and always have spaces around the `->` arrow if present. (See `Function Annotations`_ below for more about function annotations.)

  함수 주석은 콜론에 일반 규칙을 사용해야하며 항상 `->` 화살표 주위에 공백이 있어야합니다 . 함수 주석에 대한 자세한 내용은 아래의 함수 주석을 참조하십시오 .

  Yes::

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

  No::

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

- Don't use spaces around the `=` sign when used to indicate a keyword argument, or when used to indicate a default value for an *unannotated* function parameter.

  키워드 인수를 표시하거나 어노테이션이없는 함수 매개 변수 의 기본값을 표시하는 데 사용될 때는 `=` 부호 주위에 공백을 사용하지 마십시오 .

  Yes::

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

No::

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

When combining an argument annotation with a default value, however, do use spaces around the `=` sign:

그러나 인수 어노테이션을 기본값과 결합 할 때 `=` 기호 주위에 공백을 사용 하십시오.

Yes::

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

No::

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

- Compound statements (multiple statements on the same line) are generally discouraged.

  복합 문장 (같은 줄에 여러 문장)은 일반적으로 사용하지 않는 것이 좋습니다.

  Yes::

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

  Rather not::

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

- While sometimes it's okay to put an if/for/while with a small body on the same line, never do this for multi-clause statements. Also avoid folding such long lines!

작은 줄이있는 if / for / while을 같은 줄에 두는 것이 좋은 경우도 있지만, 다중 절에서이 작업을 수행하지 마십시오. 또한 긴 줄을 접지 마십시오!

Rather not::

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

Definitely not::

```
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()
```

When to Use Trailing Commas

후행 쉼표를 사용하는 경우

Trailing commas are usually optional, except they are mandatory when making a tuple of one element (and in Python 2 they have semantics for the `print` statement). For clarity, it is recommended to surround the latter in (technically redundant) parentheses.

후행 쉼표는 일반적으로 선택 사항입니다. 단 하나의 요소로 튜플을 만들 때 필수적이며 Python 2에서는 `print` 문에 대한 의미가 있습니다 . 명확하게하기 위해 (기술적으로 중복되는) 괄호 안에 후자를 포함하는 것이 좋습니다.

Yes::

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

OK, but confusing::

그래도 혼란 스럽습니다.

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

When trailing commas are redundant, they are often helpful when a version control system is used, when a list of values, arguments or imported items is expected to be extended over time. The pattern is to put each value (etc.) on a line by itself, always adding a trailing comma, and add the close parenthesis/bracket/brace on the next line. However it does not make sense to have a trailing comma on the same line as the closing delimiter (except in the above case of singleton tuples).

후행 쉼표가 중복되면 버전 관리 시스템을 사용할 때, 값 목록, 인수 또는 가져온 항목이 시간이 지남에 따라 확장 될 것으로 예상되는 경우 종종 도움이됩니다. 패턴은 각 값 (예 : 등)을 한 줄에 하나씩 배치하고 항상 뒤에 쉼표를 추가하고 다음 줄에 닫는 괄호 / 괄호 / 괄호를 추가합니다. 그러나 닫는 구분 기호와 동일한 줄에 후행 쉼표를 사용하는 것은 의미가 없습니다 (위의 싱글 톤 튜플은 제외).

Yes::

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

No::

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

Comments

Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up-to-date when the code changes!

코드와 모순되는 주석은 주석이없는 것보다 나쁩니다. 코드가 변경 될 때 항상 주석을 최신 상태로 유지하십시오!

Comments should be complete sentences. The first word should be capitalized, unless it is an identifier that begins with a lower case letter (never alter the case of identifiers!).

주석은 완전한 문장이어야합니다. 첫 단어는 소문자로 시작하는 식별자가 아닌 한 대문자로 표기해야합니다 (식별자의 경우를 절대로 바꾸지 마십시오!).

Block comments generally consist of one or more paragraphs built out of complete sentences, with each sentence ending in a period.

블록 주석은 일반적으로 완전한 문장으로 구성된 하나 이상의 단락으로 구성되며 각 문장은 마침표로 끝납니다.

You should use two spaces after a sentence-ending period in multi- sentence comments, except after the final sentence.

마지막 문장 이후를 제외하고 문장 끝 기간 이후에는 문장 문장에서 두 칸을 사용해야합니다.

When writing English, follow Strunk and White.

영어를 쓸 때 Strunk and White를 따르십시오.

Python coders from non-English speaking countries: please write your comments in English, unless you are 120% sure that the code will never be read by people who don't speak your language.

영어를 사용하지 않는 국가의 Python 코더 : 언어를 모르는 사람이 코드를 읽지 않을 것이라고 120 % 확신하지 않는 한 영어로 의견을 작성하십시오.

Block Comments

커멘트 영역

Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code. Each line of a block comment starts with a # and a single space (unless it is indented text inside the comment).

블록 주석은 일반적으로 그 뒤에 오는 일부 (또는 모든) 코드에 적용되며 해당 코드와 동일한 수준으로 들여 쓰기됩니다. 블록 주석의 각 줄은 # 과 단일 공백으로 시작 합니다 (주석 안에 텍스트가 들여 쓰기되지 않은 경우).

Paragraphs inside a block comment are separated by a line containing a single #.

블록 주석 내의 단락은 단일 #을 포함하는 행으로 구분됩니다 .

Inline Comments

Use inline comments sparingly.

인라인 주석은 드물게 사용하십시오.

An inline comment is a comment on the same line as a statement. Inline comments should be separated by at least two spaces from the statement. They should start with a # and a single space.

인라인 주석은 명령문과 같은 행에 대한 주석입니다. 인라인 주석은 명령문에서 최소한 두 공백으로 분리해야합니다. #과 단일 공백으로 시작해야합니다.

Inline comments are unnecessary and in fact distracting if they state the obvious. Don't do this::

인라인 주석은 불필요하며 실제로 명백한 경우 산만합니다. 이 작업을 수행하지 마십시오 :

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

But sometimes, this is useful::

그러나 때로는 이것이 유용합니다.

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

Documentation Strings

설명서 문자열

Conventions for writing good documentation strings (a.k.a. "docstrings") are immortalized in PEP 257.

좋은 문서화 문자열 (일명 "docstrings")을 작성하기위한 규칙은 PEP 257 에서 불멸화됩니다 .

- Write docstrings for all public modules, functions, classes, and methods. Docstrings are not necessary for non-public methods, but you should have a comment that describes what the method does. This comment should appear after the `def` line.

모든 공개 모듈, 함수, 클래스 및 메소드에 대한 docstring을 작성하십시오. 비 공용 메서드에는 문서 문자열이 필요하지 않지만 메서드의 기능을 설명하는 주석이 있어야합니다. 이 주석은 `def` 줄 다음에 나타납니다 .

- PEP 257 describes good docstring conventions. Note that most importantly, the `"""` that ends a multiline docstring should be on a line by itself::

PEP 257 은 훌륭한 문서 문자열 규칙을 설명합니다. 가장 중요한 것은 여러 줄의 docstring을 끝내는 "" " 는 한 줄에 있어야합니다.

```
    """Return a foobang

    Optional plotz says to frobnicate the bizbaz first.
    """
```

- For one liner docstrings, please keep the closing `"""` on the same line.

하나의 라이너 문서 문자열의 경우, 닫는 "" " 를 같은 줄에 유지하십시오 .

Naming Conventions

명명 규칙

The naming conventions of Python's library are a bit of a mess, so we'll never get this completely consistent -- nevertheless, here are the currently recommended naming standards. New modules and packages (including third party frameworks) should be written to these standards, but where an existing library has a different style, internal consistency is preferred.

파이썬 라이브러리의 명명 규칙은 약간 엉망이므로, 우리는 이것이 완전히 일관성을 얻지 못할 것입니다. 그럼에도 불구하고 현재 권장되는 명명 표준이 있습니다. 새로운 모듈 및 패키지 (타사 프레임 워크 포함)는 이러한 표준에 작성해야하지만 기존 라이브러리의 스타일이 다른 경우 내부 일관성이 선호됩니다.

Overriding Principle

재정의 원칙

Names that are visible to the user as public parts of the API should follow conventions that reflect usage rather than implementation.

API의 공개 부분으로 사용자에게 표시되는 이름은 구현보다는 사용을 반영하는 규칙을 따라야합니다.

Descriptive: Naming Styles

설명 : 이름 지정 스타일

There are a lot of different naming styles. It helps to be able to recognize what naming style is being used, independently from what they are used for.

다양한 이름 지정 스타일이 있습니다. 사용하는 이름과 무관하게 어떤 이름 지정 스타일을 사용하고 있는지 인식 할 수 있습니다.

The following naming styles are commonly distinguished:

다음과 같은 이름 지정 스타일이 일반적으로 구별됩니다.

- `b` (single lowercase letter)
- `B` (single uppercase letter)
- `lowercase`
- `lower_case_with_underscores`
- `UPPERCASE`
- `UPPER_CASE_WITH_UNDERSCORES`
- `CapitalizedWords` (or CapWords, or CamelCase -- so named because of the bumpy look of its letters [4]_). This is also sometimes known as StudlyCaps.

  대문자로 된 단어 (또는 CapWords 또는 CamelCase-문자의 울퉁불퉁 한 모양 때문에 이름이 지정됨 [4]) 이것은 때때로 StudlyCaps라고도합니다.

  Note: When using acronyms in CapWords, capitalize all the letters of the acronym. Thus HTTPServerError is better than HttpServerError.

  참고 : CapWords에서 두문자어를 사용하는 경우 약어의 모든 문자를 대문자로 사용하십시오. 따라서 HTTPServerError가 HttpServerError보다 낫습니다.

- `mixedCase` (differs from CapitalizedWords by initial lowercase character!)(대문자와 대문자가 다릅니다!)
- `Capitalized_Words_With_Underscores` (ugly!)(어글리!)

There's also the style of using a short unique prefix to group related names together. This is not used much in Python, but it is mentioned for completeness. For example, the `os.stat()` function returns a tuple whose

items traditionally have names like `st_mode`, `st_size`, `st_mtime` and so on. (This is done to emphasize the correspondence with the fields of the POSIX system call struct, which helps programmers familiar with that.)

짧은 고유 접두사를 사용하여 관련 이름을 그룹화하는 스타일도 있습니다. 이것은 파이썬에서 많이 사용되지는 않지만 완전성을 위해 언급됩니다. 예를 들어, `os.stat ()` 함수는 전통적으로 항목의 이름이 `st_mode` , `st_size` , `st_mtime` 등과 같은 튜플을 반환합니다 . (이것은 POSIX 시스템 콜 구조체의 필드와의 대응을 강조하기 위해 수행되며, 이는 프로그래머에게 익숙합니다.)

The X11 library uses a leading X for all its public functions. In Python, this style is generally deemed unnecessary because attribute and method names are prefixed with an object, and function names are prefixed with a module name.

X11 라이브러리는 모든 공용 기능에 선행 X를 사용합니다. 파이썬에서이 스타일은 일반적으로 속성과 메소드 이름 앞에 객체가 붙고 함수 이름 앞에 모듈 이름이 있기 때문에 불필요한 것으로 간주됩니다.

In addition, the following special forms using leading or trailing underscores are recognized (these can generally be combined with any case convention):

또한 선행 또는 후행 밑줄을 사용하는 다음과 같은 특수 형식이 인식됩니다 (일반적으로 모든 경우 규칙과 결합될 수 있음).

- `_single_leading_underscore`: weak "internal use" indicator. E.g. `from M import *` does not import objects whose names start with an underscore.

  `_single_leading_underscore` : 약한 "내부 사용"표시기. 예 를 들어 M import * 에서는 이름이 밑줄로 시작하는 객체를 가져 오지 않습니다.

- `single_trailing_underscore_`: used by convention to avoid conflicts with Python keyword, e.g. ::

  `single_trailing_underscore_` : 예를 들어 파이썬 키워드와의 충돌을 피하기 위해 사용

  ```
  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_underscore` : 클래스 속성의 이름을 지정할 때 이름 mangling을 호출합니다 (FooBar 클래스 내부에서 `__boo` 는 `_FooBar__boo`가 되며 아래 참조).

- `__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.

  `__double_leading_and_trailing_underscore__` : 사용자 제어 네임 스페이스에있는 "마법의"개체 또는 특성입니다. 예를 `__init__` , `__import__` 또는 `__file__` . 그런 이름을 발명하지 마십시오. 문서화 된 대로만 사용하십시오.

Prescriptive: Naming Conventions

규범 : 명명 규칙

Names to Avoid

피해야 할 이름

Never use the characters 'l' (lowercase letter el), 'O' (uppercase letter oh), or 'I' (uppercase letter eye) as single character variable names.

문자 'l'(소문자 소문자 el), 'O'(대문자 oh) 또는 'I'(대문자 눈)를 단일 문자 변수 이름으로 사용하지 마십시오.

In some fonts, these characters are indistinguishable from the numerals one and zero. When tempted to use 'l', use 'L' instead.

일부 글꼴에서 이러한 문자는 숫자 1과 0과 구별 할 수 없습니다. 'l'을 사용하고 싶을 때는 대신 'L'을 사용하십시오.

ASCII Compatibility

ASCII 호환성

Identifiers used in the standard library must be ASCII compatible as described in the `policy section <https://www.python.org/dev/peps/pep-3131/#policy-specification>`_ of PEP 3131.

표준 라이브러리에 사용 된 식별자 는 PEP 3131 의 정책 섹션 에 설명 된대로 ASCII 호환 가능해야합니다 .

Package and Module Names

패키지 및 모듈 이름

Modules should have short, all-lowercase names. Underscores can be used in the module name if it improves readability. Python packages should also have short, all-lowercase names, although the use of underscores is discouraged.

모듈은 짧은 소문자 이름을 가져야합니다. 가독성을 높이려면 모듈 이름에 밑줄을 사용할 수 있습니다. 밑줄은 사용하지 않는 것이 좋지만 파이썬 패키지는 소문자로 된 짧은 소문자 이름을 가져야합니다.

When an extension module written in C or C++ has an accompanying Python module that provides a higher level (e.g. more object oriented) interface, the C/C++ module has a leading underscore (e.g. `_socket`).

C 또는 C ++로 작성된 확장 모듈에 상위 레벨 (예 : 더 많은 객체 지향) 인터페이스를 제공하는 동반 Python 모듈이있는 경우 C / C ++ 모듈에는 밑줄 (예 : `_socket`)이 있습니다.

Class Names

Class names should normally use the CapWords convention.

클래스 이름은 일반적으로 CapWords 규칙을 사용해야합니다.

The naming convention for functions may be used instead in cases where the interface is documented and used primarily as a callable.

인터페이스가 문서화되고 주로 호출 가능으로 사용되는 경우 함수의 명명 규칙이 대신 사용될 수 있습니다.

Note that there is a separate convention for builtin names: most builtin names are single words (or two words run together), with the CapWords convention used only for exception names and builtin constants.

내장 이름에 대한 별도의 규칙이 있습니다. 대부분의 내장 이름은 단일 단어 (또는 두 단어가 함께 실행 됨)이며 CapWords 규칙은 예외 이름 및 내장 상수에만 사용됩니다.

Type Variable Names

타입 변수 이름

Names of type variables introduced in PEP 484 should normally use CapWords preferring short names: `T`, `AnyStr`, `Num`. It is recommended to add suffixes `_co` or `_contra` to the variables used to declare covariant or contravariant behavior correspondingly::

PEP 484에 도입 된 유형 변수의 이름 은 일반적으로 짧은 이름을 선호하는 CapWord를 사용해야합니다 : `T` , `AnyStr` , `Num` . 공변량 또는 반 변량 동작 을 적절 하게 선언하는 데 사용되는 변수 에 접미사 `_co` 또는 `_contra` 를 추가하는 것이 좋습니다 .

```
from typing import TypeVar

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

Exception Names

예외 이름

Because exceptions should be classes, the class naming convention applies here. However, you should use the suffix "Error" on your exception names (if the exception actually is an error).

예외는 클래스 여야하므로 클래스 명명 규칙이 여기에 적용됩니다. 그러나 예외 이름에 접미사 "Error"를 사용해야합니다 (예외가 실제로 오류 인 경우).

Global Variable Names

전역 변수 이름

(Let's hope that these variables are meant for use inside one module only.) The conventions are about the same as those for functions.

(이러한 변수가 하나의 모듈에서만 사용되기를 바랍니다.) 규칙은 함수의 규칙과 거의 같습니다.

Modules that are designed for use via `from M import *` should use the `__all__` mechanism to prevent exporting globals, or use the older convention of prefixing such globals with an underscore (which you might want to do to indicate these globals are "module non-public").

`from M import *` 를 통해 사용하도록 설계된 모듈 은 `__all__` 메커니즘을 사용하여 전역 내보내기를 방지하거나 이러한 전역 앞에 밑줄을 붙이는 기존 규칙을 사용해야합니다 (이 전역을 "비공개 모듈"이라고 표시하려

는 경우 ").

Function and Variable Names

함수와 변수 이름

Function names should be lowercase, with words separated by underscores as necessary to improve readability.

가독성을 높이기 위해 필요에 따라 밑줄로 구분 된 단어와 함께 함수 이름은 소문자 여야합니다.

Variable names follow the same convention as function names.

변수 이름은 함수 이름과 동일한 규칙을 따릅니다.

mixedCase is allowed only in contexts where that's already the prevailing style (e.g. threading.py), to retain backwards compatibility.

mixedCase는 이전 스타일과의 호환성을 유지하기 위해 이미 일반적인 스타일 (예 : threading.py) 인 컨텍스트에서만 허용됩니다.

Function and Method Arguments

함수와 메소드 인수

Always use `self` for the first argument to instance methods.

인스턴스 메소드의 첫 번째 인수에는 항상 `self` 를 사용 하십시오.

Always use `cls` for the first argument to class methods.

클래스 메소드의 첫 번째 인수에는 항상 `cls` 를 사용 하십시오.

If a function argument's name clashes with a reserved keyword, it is generally better to append a single trailing underscore rather than use an abbreviation or spelling corruption. Thus `class_` is better than `clss`. (Perhaps better is to avoid such clashes by using a synonym.)

함수 인수의 이름이 예약 키워드와 충돌하는 경우 일반적으로 약어 또는 철자 손상을 사용하는 대신 단일 후행 밑줄을 추가하는 것이 좋습니다. 따라서 `class_` 가 `clss` 보다 낫습니다 . (아마 동의어를 사용하여 이러한 충돌을 피하는 것이 좋습니다.)

Method Names and Instance Variables

메소드 이름 및 인스턴스 변수

Use the function naming rules: lowercase with words separated by underscores as necessary to improve readability.

가독성을 높이기 위해 필요에 따라 밑줄로 구분 된 단어가있는 소문자 기능 명명 규칙을 사용하십시오.

Use one leading underscore only for non-public methods and instance variables.

비공개 메소드 및 인스턴스 변수에만 하나의 선행 밑줄을 사용하십시오.

To avoid name clashes with subclasses, use two leading underscores to invoke Python's name mangling rules.

서브 클래스와의 이름 충돌을 피하려면 두 개의 밑줄을 사용하여 Python의 이름 변경 규칙을 호출하십시오.

Python mangles these names with the class name: if class Foo has an attribute named `__a`, it cannot be accessed by `Foo.__a`. (An insistent user could still gain access by calling `Foo._Foo__a`.) Generally, double leading underscores should be used only to avoid name conflicts with attributes in classes designed to be subclassed.

파이썬은이 이름을 클래스 이름과 맹 글링 합니다. Foo 클래스에 `__a` 라는 속성이 있으면 `Foo.__ a` 로 액세스 할 수 없습니다 . (일관된 사용자는 여전히 `Foo._Foo__a` 를 호출하여 액세스 할 수 있습니다 .) 일반적으로 이중 선행 밑줄은 서브 클래 싱되도록 설계된 클래스의 속성과 이름 충돌을 피하기 위해 사용해야합니다.

Note: there is some controversy about the use of __names (see below).

참고 : __names 사용에 대한 논쟁이 있습니다 (아래 참조).

Constants

상수

Constants are usually defined on a module level and written in all capital letters with underscores separating words. Examples include `MAX_OVERFLOW` and `TOTAL`.

상수는 일반적으로 모듈 수준에서 정의되며 단어를 구분하는 밑줄과 함께 모든 대문자로 작성됩니다. 예는 `MAX_OVERFLOW` 및 `TOTAL` 입니다.

Designing for Inheritance

상속을위한 설계

Always decide whether a class's methods and instance variables (collectively: "attributes") should be public or non-public. If in doubt, choose non-public; it's easier to make it public later than to make a public attribute non-public.

클래스의 메소드와 인스턴스 변수 (총칭하여 "속성")가 퍼블릭인지 퍼블릭이어야하는지 항상 결정하십시오. 확실하지 않은 경우 비공개를 선택하십시오. 공개 속성을 비공개로 만드는 것보다 나중에 공개하는 것이 더 쉽습니다.

Public attributes are those that you expect unrelated clients of your class to use, with your commitment to avoid backwards incompatible changes. Non-public attributes are those that are not intended to be used by third parties; you make no guarantees that non-public attributes won't change or even be removed.

공용 속성은 이전 버전과 호환되지 않는 변경 사항을 피하기 위해 클래스의 관련없는 클라이언트가 사용할 것으로 예상하는 속성입니다. 비공개 속성은 타사에서 사용하지 않는 속성입니다. 비공개 속성이 변경되거나 제거되지 않을 것이라는 보장은 없습니다.

We don't use the term "private" here, since no attribute is really private in Python (without a generally unnecessary amount of work).

여기서는 "비공개"라는 용어를 사용하지 않습니다. 일반적으로 불필요한 작업없이 파이썬에서 실제로는 속성이 없기 때문입니다.

Another category of attributes are those that are part of the "subclass API" (often called "protected" in other languages). Some classes are designed to be inherited from, either to extend or modify aspects of the class's behavior. When designing such a class, take care to make explicit decisions about which attributes are public, which are part of the subclass API, and which are truly only to be used by your base class.

또 다른 속성 범주는 "하위 클래스 API"의 일부인 속성입니다 (다른 언어에서는 "protected"라고도 함). 일부 클래스는 클래스 동작의 측면을 확장하거나 수정하기 위해 상속되도록 설계되었습니다. 이러한 클래스를 설계 할 때는 공개되는 속성, 서브 클래스 API의 일부, 기본 클래스에서만 사용할 속성에 대해 명시 적으로 결정해야합니다.

With this in mind, here are the Pythonic guidelines:

이를 염두에두고 다음은 파이썬 지침입니다.

- Public attributes should have no leading underscores.
  공개 속성에는 밑줄이 없어야합니다.

- If your public attribute name collides with a reserved keyword, append a single trailing underscore to your attribute name. This is preferable to an abbreviation or corrupted spelling. (However, notwithstanding this rule, 'cls' is the preferred spelling for any variable or argument which is known to be a class, especially the first argument to a class method.)

  공개 속성 이름이 예약 키워드와 충돌하는 경우 속성 이름에 단일 밑줄을 추가하십시오. 약어 또는 손상된 철자보다 선호됩니다. 그러나이 규칙에도 불구하고 'cls'는 클래스로 알려진 모든 변수 또는 인수, 특히 클래스 메소드에 대한 첫 번째 인수에 대해 선호되는 철자입니다.

  Note 1: See the argument name recommendation above for class methods.

  참고 1 : 클래스 메소드에 대해서는 위의 인수 이름 권장 사항을 참조하십시오.

- For simple public data attributes, it is best to expose just the attribute name, without complicated accessor/mutator methods. Keep in mind that Python provides an easy path to future enhancement, should you find that a simple data attribute needs to grow functional behavior. In that case, use properties to hide functional implementation behind simple data attribute access syntax.

  단순한 공개 데이터 속성의 경우 복잡한 접근 자 / 돌연변이 방법없이 속성 이름 만 노출하는 것이 가장 좋습니다. 간단한 데이터 속성이 기능적 행동을 키워야한다는 사실을 알게되면 Python은 향후 개선을위한 쉬운 길을 제공한다는 점을 명심하십시오. 이 경우 속성을 사용하여 간단한 데이터 속성 액세스 구문 뒤에 기능 구현을 숨길 수 있습니다.

  Note 1: Properties only work on new-style classes.

  참고 1 : 속성은 새 스타일 클래스에서만 작동합니다.

  Note 2: Try to keep the functional behavior side-effect free, although side-effects such as caching are generally fine.

  참고 2 : 캐싱과 같은 부작용은 일반적으로 좋지만 기능적 동작 부작용을 방지하십시오.

Note 3: Avoid using properties for computationally expensive operations; the attribute notation makes the caller believe that access is (relatively) cheap.

참고 3 : 계산 비용이 많이 드는 작업에는 속성을 사용하지 마십시오. 속성 표기법은 호출자가 액세스가 상대적으로 저렴하다고 믿게합니다.

- 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. This invokes Python's name mangling algorithm, where the name of the class is mangled into the attribute name. This helps avoid attribute name collisions should subclasses inadvertently contain attributes with the same name.

  클래스를 서브 클래스로 만들려고하고 서브 클래스를 사용하지 않으려는 속성이있는 경우 이중 밑줄로 밑줄을 두지 말고 이름을 지정하십시오. 이것은 클래스 이름이 속성 이름으로 맹 글링되는 Python의 이름 맹 글링 알고리즘을 호출합니다. 서브 클래스에 이름이 같은 속성이 실수로 포함되어 있으면 속성 이름 충돌을 피할 수 있습니다.

  Note 1: Note that only the simple class name is used in the mangled name, so if a subclass chooses both the same class name and attribute name, you can still get name collisions.

  참고 1 : 맹 글링 된 이름에는 단순 클래스 이름 만 사용되므로 하위 클래스가 동일한 클래스 이름과 속성 이름을 모두 선택하더라도 이름 충돌이 발생할 수 있습니다.

  Note 2: Name mangling can make certain uses, such as debugging and `__getattr__()`, less convenient. However the name mangling algorithm is well documented and easy to perform manually.

  참고 2 : 이름 맹 글링은 디버깅 및 `__getattr __()` 과 같은 특정 용도를 덜 편리하게 만들 수 있습니다. 그러나 이름 조작 알고리즘은 잘 문서화되어 있으며 수동으로 쉽게 수행 할 수 있습니다.

  Note 3: Not everyone likes name mangling. Try to balance the need to avoid accidental name clashes with potential use by advanced callers.

  참고 3 : 모든 사람이 이름 맹 글링을 좋아하는 것은 아닙니다. 우발적 인 이름 충돌을 피할 필요성과 고급 발신자가 잠재적으로 사용할 수있는 균형을 맞추십시오.

Public and Internal Interfaces

공용 및 내부 인터페이스

Any backwards compatibility guarantees apply only to public interfaces. Accordingly, it is important that users be able to clearly distinguish between public and internal interfaces.

이전 버전과의 호환성 보증은 공용 인터페이스에만 적용됩니다. 따라서 사용자는 공용 인터페이스와 내부 인터페이스를 명확하게 구분할 수 있어야합니다.

Documented interfaces are considered public, unless the documentation explicitly declares them to be provisional or internal interfaces exempt from the usual backwards compatibility guarantees. All undocumented interfaces should be assumed to be internal.

문서화 된 인터페이스가 일반적인 이전 버전과의 호환성 보증에서 제외되는 임시 또는 내부 인터페이스라고 명시 적으로 선언하지 않는 한 문서화 된 인터페이스는 공용으로 간주됩니다. 문서화되지 않은 모든 인터페이스

는 내부 인터페이스로 가정해야합니다.

To better support introspection, modules should explicitly declare the names in their public API using the `__all__` attribute. Setting `__all__` to an empty list indicates that the module has no public API.

내부 검사를보다 잘 지원하려면 모듈이 `__all__` 속성을 사용하여 공개 API에서 이름을 명시 적으로 선언해야합니다. `__all__` 을 빈 목록으로 설정 하면 모듈에 공개 API가 없음을 나타냅니다.

Even with `__all__` set appropriately, internal interfaces (packages, modules, classes, functions, attributes or other names) should still be prefixed with a single leading underscore.

`__all__` 을 적절하게 설정 하더라도 내부 인터페이스 (패키지, 모듈, 클래스, 함수, 특성 또는 기타 이름) 앞에는 단일 밑줄이 붙어야합니다.

An interface is also considered internal if any containing namespace (package, module or class) is considered internal.

포함하는 네임 스페이스 (패키지, 모듈 또는 클래스)가 내부로 간주되면 인터페이스도 내부로 간주됩니다.

Imported names should always be considered an implementation detail. Other modules must not rely on indirect access to such imported names unless they are an explicitly documented part of the containing module's API, such as `os.path` or a package's `__init__` module that exposes functionality from submodules.

가져온 이름은 항상 구현 세부 사항으로 간주되어야합니다. 다른 모듈은 포함 된 모듈의 API에서 명시 적으로 문서화 된 부분 (예 : `os.path` 또는 하위 모듈에서 기능을 노출 하는 패키지의 `__init__` 모듈)이 아닌 한 가져온 이름에 대한 간접 액세스에 의존해서는 안됩니다 .

Programming Recommendations

프로그래밍 추천

- Code should be written in a way that does not disadvantage other implementations of Python (PyPy, Jython, IronPython, Cython, Psyco, and such).

  코드는 다른 Python 구현 (PyPy, Jython, IronPython, Cython, Psyco 등)에 불리하지 않은 방식으로 작성해야합니다.

  For example, do not rely on CPython's efficient implementation of in-place string concatenation for statements in the form `a += b` or `a = a + b`. This optimization is fragile even in CPython (it only works for some types) and isn't present at all in implementations that don't use refcounting. In performance sensitive parts of the library, the `''.join()` form should be used instead. This will ensure that concatenation occurs in linear time across various implementations.

  예를 들어, `a + = b` 또는 `a = a + b` 형식 의 명령문에 대해 CPython의 내부 문자열 연결의 효율적인 구현에 의존하지 마십시오 . 이 최적화는 CPython에서도 취약하며 (일부 유형에서만 작동) 재 계산을 사용하지 않는 구현에는 전혀 존재하지 않습니다. 라이브러리의 성능에 민감한 부분에서는 `''.join()` 형식을 대신 사용해야합니다. 이렇게하면 다양한 구현에서 선형 시간으로 연결이 발생합니다.

- Comparisons to singletons like None should always be done with `is` or `is not`, never the equality operators.

  에 없음 같은 싱글에 대한 비교는 항상 수행해야합니다 `is` 또는 `is not` , 결코 평등 연산자.

  Also, beware of writing `if x` when you really mean `if x is not None` -- e.g. when testing whether a variable or argument that defaults to None was set to some other value. The other value might have a type (such as a container) that could be false in a boolean context!

  또한 `if x is not None` 아닌 경우를 의미 할 `if x` 를 쓰는 경우 에도주의하십시오. 예를 들어, 기본값이 None 인 변수 또는 인수가 다른 값으로 설정되어 있는지 테스트 할 때. 다른 값에는 부울 컨텍스트에서 false 일 수있는 유형 (예 : 컨테이너)이있을 수 있습니다!

- Use `is not` operator rather than `not ... is`. While both expressions are functionally identical, the former is more readable and preferred.

  사용이 `is not` 운영자가 아니라 `not ... is`. 두 표현 모두 기능적으로 동일하지만 전자가 더 읽기 쉽고 선호됩니다.

  Yes::

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

  No::

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

- 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.

  풍부한 비교를 사용하여 순서화 작업을 구현할 때는 특정 비교 만 수행하기 위해 다른 코드에 의존하기보다는 6 가지 작업 (`__eq__` , `__ne__` , `__lt__` , `__le__` , `__gt__` , `__ge__`)을 모두 구현하는 것이 가장 좋습니다 .

  To minimize the effort involved, the `functools.total_ordering()` decorator provides a tool to generate missing comparison methods.

  관련된 노력을 최소화하기 위해 `functools.total_ordering()` 데코레이터는 누락 된 비교 방법을 생성하는 도구를 제공합니다.

  PEP 207 indicates that reflexivity rules *are* assumed by Python. Thus, the interpreter may swap `y > x` with `x < y`, `y >= x` with `x <= y`, and may swap the arguments of `x == y` and `x != y`. The `sort()` and `min()` operations are guaranteed to use the `<` operator and the `max()` function uses the `>` operator. However, it is best to implement all six operations so that confusion doesn't arise in other contexts.

PEP 207 은 반사 규칙 이 Python에 의해 가정 됨을 나타냅니다 . 따라서 인터프리터는 `y > x` 를 `x < y` 로 바꾸고 `y >= x` 를 `x <= y` 로 바꾸고 `x == y` 및 `x != y` 인수를 바꿀 수 있습니다 . `sort()` 와 `min()` 조작은 사용 보장 `<` 연산자와 `max()` 함수를 사용하여 `>` 오퍼레이터. 그러나 다른 상황에서는 혼동이 발생하지 않도록 6 가지 작업을 모두 구현하는 것이 가장 좋습니다.

- Always use a def statement instead of an assignment statement that binds a lambda expression directly to an identifier.

  람다 식을 식별자에 직접 바인딩하는 할당 문 대신 항상 def 문을 사용하십시오.

  Yes::

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

  No::

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

  The first form means that the name of the resulting function object is specifically 'f' instead of the generic ''. This is more useful for tracebacks and string representations in general. The use of the assignment statement eliminates the sole benefit a lambda expression can offer over an explicit def statement (i.e. that it can be embedded inside a larger expression)

  첫 번째 형식은 결과 함수 객체의 이름이 일반 ''대신 'f'임을 의미합니다. 이것은 일반적으로 트레이스 백 및 문자열 표현에 더 유용합니다. 대 입문을 사용하면 람다식이 명시적인 def 문에 제공 할 수있는 유일한 이점을 제거 할 수 있습니다 (즉, 더 큰 식 안에 포함시킬 수 있음)

- Derive exceptions from `Exception` rather than `BaseException`. Direct inheritance from `BaseException` is reserved for exceptions where catching them is almost always the wrong thing to do.

  에서 예외 유도 `Exception` 가 아닌 `BaseException` 을 . `BaseException` 에서 직접 상속은 예외를 잡는 것이 거의 항상 잘못된 경우를 위해 예약되어 있습니다.

  Design exception hierarchies based on the distinctions that code *catching* the exceptions is likely to need, rather than the locations where the exceptions are raised. Aim to answer the question "What went wrong?" programmatically, rather than only stating that "A problem occurred" (see PEP 3151 for an example of this lesson being learned for the builtin exception hierarchy)

  예외가 발생하는 위치가 아니라 예외를 포착 하는 코드 가 필요할 가능성이 높은 차이점을 기반으로 예외 계층을 설계하십시오 . "무엇이 잘못 되었습니까?"라는 질문에 답하십시오. "문제가 발생했다"는 것만이 아니라 프로그래밍 방식으로 (내장 예외 계층에 대해 학습되는이 학습의 예는 PEP 3151 참조)

  Class naming conventions apply here, although you should add the suffix "Error" to your exception classes if the exception is an error. Non-error exceptions that are used for non-local flow control or other forms of signaling need no special suffix.

예외가 오류 인 경우 예외 클래스에 접미사 "Error"를 추가해야하지만 클래스 이름 지정 규칙이 여기에 적용됩니다. 로컬이 아닌 흐름 제어 또는 다른 형태의 신호에 사용되는 비 오류 예외에는 특별한 접미사가 필요하지 않습니다.

- Use exception chaining appropriately. In Python 3, "raise X from Y" should be used to indicate explicit replacement without losing the original traceback.

  예외 체인을 적절하게 사용하십시오. 파이썬 3에서 "X에서 Y 올리기"는 원래 역 추적을 잃지 않으면 서 명시적인 대체를 나타 내기 위해 사용해야합니다.

  When deliberately replacing an inner exception (using "raise X" in Python 2 or "raise X from None" in Python 3.3+), ensure that relevant details are transferred to the new exception (such as preserving the attribute name when converting KeyError to AttributeError, or embedding the text of the original exception in the new exception message).

  내부 예외를 의도적으로 바꾸는 경우 (Python 2에서 "raise X"또는 Python 3.3+에서 "raise X from None"사용), KeyError를 AttributeError로 변환 할 때 속성 이름 유지와 같이 관련 세부 사항이 새 예외로 전송되는지 확인하십시오. 또는 새 예외 메시지에 원래 예외 텍스트를 포함).

- When raising an exception in Python 2, use `raise ValueError('message')` instead of the older form `raise ValueError, 'message'`.

  Python 2에서 예외를 발생 시킬 때는 이전 양식 `raise ValueError, 'message'` 대신 대신 `raise ValueError ( 'message')` 를 사용하십시오 .

  The latter form is not legal Python 3 syntax.

  후자의 형식은 올바른 Python 3 구문이 아닙니다.

  The paren-using form also means that when the exception arguments are long or include string formatting, you don't need to use line continuation characters thanks to the containing parentheses.

  Paren 사용 양식은 또한 예외 인수가 길거나 문자열 형식을 포함 할 때 포함 된 괄호로 인해 줄 연속 문자를 사용할 필요가 없음을 의미합니다.

- When catching exceptions, mention specific exceptions whenever possible instead of using a bare `except:` clause::

  예외를 잡을 때 bare `except:` 절 을 사용하는 대신 가능할 때마다 특정 예외를 언급하십시오 .

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

  A bare `except:` clause will catch SystemExit and KeyboardInterrupt exceptions, making it harder to interrupt a program with Control-C, and can disguise other problems. If you want to catch all exceptions that signal program errors, use `except Exception:` (bare except is equivalent to `except BaseException:`).

bare `except:` 절은 SystemExit 및 KeyboardInterrupt 예외를 포착하여 Control-C로 프로그램을 인터럽트하기 어렵게 만들고 다른 문제를 위장 할 수 있습니다. 프로그램 오류를 알리는 모든 예외를 포착하려면 `except Exception:` 를 제외하고 사용 하십시오 (Bare `except BaseException:` 제외 와 동일 함).

A good rule of thumb is to limit use of bare 'except' clauses to two cases:

일반적으로 'except'절의 사용을 두 가지 경우로 제한하는 것이 좋습니다.

1. If the exception handler will be printing out or logging the traceback; at least the user will be aware that an error has occurred.

예외 처리기가 트레이스 백을 인쇄하거나 기록하는 경우 최소한 사용자는 오류가 발생했음을 알고있을 것입니다.

2. If the code needs to do some cleanup work, but then lets the exception propagate upwards with `raise`. `try...finally` can be a better way to handle this case.

   코드가 정리 작업을 수행해야하지만 `raise` 를 사용하여 예외가 위쪽으로 전파되도록합니다 . `try...finally` 으로이 사건을 처리하는 더 좋은 방법이 될 수 있습니다.

- When binding caught exceptions to a name, prefer the explicit name binding syntax added in Python 2.6::

  발견 된 예외를 이름에 바인딩 할 때 Python 2.6에 추가 된 명시적인 이름 바인딩 구문을 선호하십시오.

```
try:
    process_data()
except Exception as exc:
    raise DataProcessingFailedError(str(exc))
```

  This is the only syntax supported in Python 3, and avoids the ambiguity problems associated with the older comma-based syntax.

  이것은 Python 3에서 지원되는 유일한 구문이며 이전 쉼표 기반 구문과 관련된 모호한 문제를 피합니다.

- When catching operating system errors, prefer the explicit exception hierarchy introduced in Python 3.3 over introspection of `errno` values.

  운영 체제 오류를 잡을 때 errno 값의 검사보다 Python 3.3에 도입 된 명시 적 예외 계층 구조를 선호 합니다.

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

  또한 모든 try / except 절에 대해 `try` 절을 필요한 최소 코드 양으로 제한하십시오 . 다시 말하면 마스킹 버그를 피할 수 있습니다.

  Yes::

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

No::

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

- When a resource is local to a particular section of code, use a `with` statement to ensure it is cleaned up promptly and reliably after use. A try/finally statement is also acceptable.

  자원이 특정 코드 섹션에 국한된 경우 `with` 문을 사용하여 사용 후 신속하고 안정적으로 정리되도록하십시오. try / finally 진술도 허용됩니다.

- Context managers should be invoked through separate functions or methods whenever they do something other than acquire and release resources.

  컨텍스트 관리자는 자원 확보 및 해제 이외의 작업을 수행 할 때마다 별도의 기능 또는 메소드를 통해 호출해야합니다.

  Yes::

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

  No::

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

  The latter example doesn't provide any information to indicate that the `__enter__` and `__exit__` methods are doing something other than closing the connection after a transaction. Being explicit is important in this case.

후자의 예는 `__enter__` 및 `__exit__` 메서드가 트랜잭션 후 연결을 닫는 것 이외의 작업을 수행하고 있음을 나타내는 정보를 제공하지 않습니다 . 이 경우 명시적인 것이 중요합니다.

- Be consistent in return statements. Either all return statements in a function should return an expression, or none of them should. If any return statement returns an expression, any return statements where no value is returned should explicitly state this as `return None`, and an explicit return statement should be present at the end of the function (if reachable).

  반환 진술에서 일관성을 유지하십시오. 함수의 모든 return 문은 식을 반환하거나 그 중 어느 것도 반환하지 않아야합니다. return 문이 식을 반환하는 경우 값이 반환되지 않은 모든 return 문은이를 명시 적으로 return None 으로 명시해야하며 함수의 끝에 명시 적 return 문이 있어야합니다 (연결 가능한 경우).

  Yes::

  ```
  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)
  ```

  No::

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

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

- Use string methods instead of the string module.

  문자열 모듈 대신 문자열 방법을 사용하십시오.

  String methods are always much faster and share the same API with unicode strings. Override this rule if backwards compatibility with Pythons older than 2.0 is required.

  문자열 메서드는 항상 훨씬 빠르며 유니 코드 문자열과 동일한 API를 공유합니다. 2.0 이전의 Python과의 하위 호환성이 필요한 경우이 규칙을 재정의하십시오.

- Use `''.startswith()` and `''.endswith()` instead of string slicing to check for prefixes or suffixes.

사용 `''.startswith()` 와 `''.endswith()` 접두사 나 접미사를 확인하기 위해 대신 문자열 슬라이스를.

startswith() and endswith() are cleaner and less error prone::

startswith () 및 endswith ()는 더 깨끗하고 오류가 덜 발생합니다.

```
Yes: if foo.startswith('bar'):
No:  if foo[:3] == 'bar':
```

- Object type comparisons should always use isinstance() instead of comparing types directly. ::

  객체 유형 비교는 유형을 직접 비교하는 대신 항상 isinstance ()를 사용해야합니다.

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

No:  if type(obj) is type(1):
```

  When checking if an object is a string, keep in mind that it might be a unicode string too! In Python 2, str and unicode have a common base class, basestring, so you can do::

  객체가 문자열인지 확인할 때 유니 코드 문자열 일 수도 있습니다. Python 2에서 str과 unicode에는 공통 기본 클래스 인 basestring이 있으므로 다음을 수행 할 수 있습니다.

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

  Note that in Python 3, `unicode` and `basestring` no longer exist (there is only `str`) and a bytes object is no longer a kind of string (it is a sequence of integers instead).

  파이썬 3에서, 유의하십시오 `unicode` 와 `basestring` 가 더 이상 존재하지 않는 (만이 `str`가)와 바이트 객체가 더 이상 문자열의 종류 (대신 정수의 순서입니다).

- For sequences, (strings, lists, tuples), use the fact that empty sequences are false. ::

  시퀀스 (문자열, 목록, 튜플)의 경우 빈 시퀀스가 거짓이라는 사실을 사용하십시오.

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

No:  if len(seq):
     if not len(seq):
```

- Don't write string literals that rely on significant trailing whitespace. Such trailing whitespace is visually indistinguishable and some editors (or more recently, reindent.py) will trim them.

후행 공백에 의존하는 문자열 리터럴을 작성하지 마십시오. 이러한 후행 공백은 시각적으로 구분할 수 없으며 일부 편집자 (또는 최근에는 reindent.py)가이를 제거합니다.

- Don't compare boolean values to True or False using `==`. ::

  `==` 를 사용하여 부울 값을 True 또는 False와 비교하지 마십시오 .

```
Yes:   if greeting:
No:    if greeting == True:
Worse: if greeting is True:
```

- Use of the flow control statements `return`/`break`/`continue` within the finally suite of a `try...finally`, where the flow control statement would jump outside the finally suite, is discouraged. This is because such statements will implicitly cancel any active exception that is propagating through the finally suite.

  흐름 제어문 이 `try ... finally` 의 최종 스위트 내에서 `return`/`break`/`continue` 을 사용하면 , 플로우 제어 명령문이 최종 스위트 외부로 점프 할 수 없습니다. 이러한 명령문은 finally 제품군을 통해 전파되는 모든 활성 예외를 암시 적으로 취소하기 때문입니다.

  No::

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

Function Annotations

함수 주석

With the acceptance of PEP 484, the style rules for function annotations are changing.

PEP 484를 수용함에 따라 기능 주석에 대한 스타일 규칙이 변경되고 있습니다.

- In order to be forward compatible, function annotations in Python 3 code should preferably use PEP 484 syntax. (There are some formatting recommendations for annotations in the previous section.)

  순방향 호환을 위해 Python 3 코드의 함수 주석은 PEP 484 구문을 사용해야 합니다. 이전 섹션에는 주석에 대한 몇 가지 형식 권장 사항이 있습니다.

- The experimentation with annotation styles that was recommended previously in this PEP is no longer encouraged.

  이 PEP에서 이전에 권장 된 주석 스타일 실험은 더 이상 권장되지 않습니다.

- However, outside the stdlib, experiments within the rules of PEP 484 are now encouraged. For example, marking up a large third party library or application with PEP 484 style type annotations, reviewing how easy it was to add those annotations, and observing whether their presence increases code understandability.

  그러나 stdlib 외부에서는 PEP 484 규칙 내 실험 이 권장됩니다. 예를 들어, PEP 484 스타일의 주석 을 사용하여 큰 타사 라이브러리 또는 응용 프로그램을 마크 업하고 , 주석을 추가하는 것이 얼마나 쉬운 지 검토하고 해당 주석의 존재 여부가 코드 이해성을 높이는지 여부를 관찰합니다.

- The Python standard library should be conservative in adopting such annotations, but their use is allowed for new code and for big refactorings.

  파이썬 표준 라이브러리는 그러한 주석을 채택하는 데 보수적이어야하지만 새로운 코드와 큰 리팩토링에는 사용할 수 있습니다.

- For code that wants to make a different use of function annotations it is recommended to put a comment of the form::

  함수 주석을 다르게 사용하려는 코드의 경우 다음 형식의 주석을 추가하는 것이 좋습니다.

  ```
  # type: ignore
  ```

  near the top of the file; this tells type checker to ignore all annotations. (More fine-grained ways of disabling complaints from type checkers can be found in PEP 484.)

  파일 상단 근처; 이것은 타입 검사기에게 모든 주석을 무시하도록 지시합니다. 유형 검사기의 불만을 비활성화하는보다 세분화 된 방법은 PEP 484 에서 찾을 수 있습니다 .

- Like linters, type checkers are optional, separate tools. Python interpreters by default should not issue any messages due to type checking and should not alter their behavior based on annotations.

  린터와 마찬가지로 타입 체커는 선택적인 별도 도구입니다. 파이썬 인터프리터는 기본적으로 타입 검사로 인해 메시지를 발행해서는 안되며 주석에 따라 동작을 변경해서는 안됩니다.

- Users who don't want to use type checkers are free to ignore them. However, it is expected that users of third party library packages may want to run type checkers over those packages. For this purpose PEP 484 recommends the use of stub files: .pyi files that are read by the type checker in preference of the corresponding .py files. Stub files can be distributed with a library, or separately (with the library author's permission) through the typeshed repo [5]_.

  형식 검사기를 사용하지 않으려는 사용자는 무시해도됩니다. 그러나 타사 라이브러리 패키지 사용자는 해당 패키지에 대해 유형 검사기를 실행할 수 있습니다. 이를 위해 PEP 484 는 스텁 파일을 사용하는 것이 좋습니다. 스텁 파일은 typeshed의 repo를 통해 (도서관 저자의 허가) 별도로 라이브러리와 함께 배포하거나 할 수있다 [5] .

- For code that needs to be backwards compatible, type annotations can be added in the form of comments. See the relevant section of PEP 484 [6]_.

이전 버전과 호환되어야하는 코드의 경우 주석 형식으로 주석을 추가 할 수 있습니다. PEP 484 의 관련 섹션을 참조하십시오 [6] .

Variable Annotations

변수 주석

PEP 526 introduced variable annotations. The style recommendations for them are similar to those on function annotations described above:

PEP 526 에는 변수 주석이 도입되었습니다. 이들에 대한 스타일 권장 사항은 위에서 설명한 함수 주석에 대한 권장 사항과 비슷합니다.

- Annotations for module level variables, class and instance variables, and local variables should have a single space after the colon.

  모듈 수준 변수, 클래스 및 인스턴스 변수 및 로컬 변수에 대한 주석은 콜론 뒤에 단일 공백이 있어야합니다.

- There should be no space before the colon.

  콜론 앞에 공백이 없어야합니다.

- If an assignment has a right hand side, then the equality sign should have exactly one space on both sides.

  과제에 오른쪽이 있으면 등호의 양쪽에 정확히 하나의 공백이 있어야합니다.

- Yes::

```
code: int

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

- No::

```
code:int  # No space after colon
code : int  # Space before colon

class Test:
    result: int=0  # No spaces around equality sign
```

- Although the PEP 526 is accepted for Python 3.6, the variable annotation syntax is the preferred syntax for stub files on all versions of Python (see PEP 484 for details).

PEP 526 은 Python 3.6에서 허용 되지만 변수 주석 구문은 모든 버전의 Python에서 스텁 파일에 대한 기본 구문입니다 (자세한 내용은 PEP 484 참조).

.. rubric:: Footnotes

.. [#fn-hi] *Hanging indentation* is a type-setting style where all the lines in a paragraph are indented except the first line. In the context of Python, the term is used to describe a style where the opening parenthesis of a parenthesized statement is the last non-whitespace character of the line, with subsequent lines being indented until the closing parenthesis.

들여 쓰기 매달린 단락의 모든 선이 첫 줄을 제외한 들여 유형 - 설정 스타일입니다. 파이썬의 맥락에서,이 용어는 괄호 안의 문장의 여는 괄호가 줄의 마지막 비 공백 문자 인 스타일을 설명하는 데 사용되며, 마지막 줄은 닫는 괄호까지 들여 쓰기됩니다.

References

참고 문헌

.. [1] PEP 7, Style Guide for C Code, van Rossum

.. [2] Barry's GNU Mailman style guide
http://barry.warsaw.us/software/STYLEGUIDE.txt

.. [3] Donald Knuth's *The TeXBook*, pages 195 and 196.

.. [4] http://www.wikipedia.com/wiki/CamelCase

.. [5] Typeshed repo
https://github.com/python/typeshed

.. [6] Suggested syntax for Python 2.7 and straddling code
https://www.python.org/dev/peps/pep-0484/ #suggested-syntax-for-python-2-7-and-straddling-code

Copyright

저작권

..
Local Variables:
mode: indented-text
indent-tabs-mode: nil
sentence-end-double-space: t
fill-column: 70
coding: utf-8
End: