はじめに

この文書は Python の標準ライブラリに含まれているPythonコードのコーディング規約です。CPython に含まれるC言語のコードについては、[**対応するC言語のスタイルガイドを記した PEP を参照してください。**](https://peps.python.org/pep-0007/)

この文書と [**PEP 257**](https://peps.python.org/pep-0257/) (Docstring 規約) は、Guido が書いたオリジナルのPythonスタイルガイドのエッセイと、 Barry のスタイルガイドに少し追記したものをまとめたものです。 [[2]](https://pep8-ja.readthedocs.io/ja/latest/#id46)

このスタイルガイドは、追加の規約が必要だとわかったり、Pythonの言語自体が変更されることで過去の規約が時代遅れになった時に徐々に改訂されてゆきます。

多くのプロジェクトには、自分たちのコーディングスタイルに関するガイドラインがあります。それとこの文書の規約の内容が矛盾した場合は、そのプロジェクトのガイドラインが優先します。

一貫性にこだわりすぎるのは、狭い心の現れである

Guido の重要な洞察のひとつに、コードは書くよりも読まれることの方が多い、というものがあります。この文書で示すガイドラインの目的は、コードを読みやすくするとともに、Pythonで書かれた幅広いコードのスタイルを一貫させることです。[**PEP 20**](https://peps.python.org/pep-0020/) にもあるように "可読性重要" です。

スタイルガイドは一貫性に関するものです。このスタイルガイドに合わせることは重要ですが、プロジェクトの中で一貫性を保つことはもっと重要です。一番重要なのは、特定のモジュールや関数の中で一貫性を保つことです。

しかし、一貫性を崩すべき場合があることも知っておいてください - つまり、このスタイルガイドが適用されない場合があります。疑問に思ったときは、あなたの判断を優先してください。他の例を調べ、一番良さそうなものを決めて下さい。そして、躊躇せずに質問して下さい！

特に、このPEPに準拠するためにコードの後方互換性を壊すようなことは絶対にしないで下さい！

この他に、特定のガイドラインを無視する正当な理由がいくつか考えられます:

1. ガイドラインに従うとコードが読みにくくなること。このPEPに準拠したコードを読んでいた人にとっても読みにくくなったのならなおさらです。
2. (多分歴史的な理由で) ガイドラインに従っていない周囲のコードと一貫性を保つため -- しかし、これは誰かの汚いコードを綺麗にするチャンスでもあります。
3. 問題になっているコードが、ガイドラインが出てくるより前に書かれたもので、 それに準拠させること以外にコードを変更する理由がないとき。
4. スタイルガイドで推奨されている機能をサポートしていない古いバージョンの Python と互換性を保つ必要がある場合。

コードのレイアウト

インデント

1レベルインデントするごとに、スペースを4つ使いましょう。

行を継続する場合は、折り返された要素を縦に揃えるようにすべきです。括弧やブラケットおよび波括弧で囲まれた要素については、Python が暗黙のうちに行を結合することを利用して揃えます。そうでない場合は、手でインデントさせることで揃えます。*突き出しインデント* [[1]](https://pep8-ja.readthedocs.io/ja/latest/#fn-hi) を使う場合は、次のことを考慮すべきです： はじめの行には引数を付けずに次の行以降をインデントし、継続行だとはっきりわかるようにしましょう。:

*# 正しい:*

*# 開き括弧に揃える*

foo = long\_function\_name(var\_one, var\_two,

var\_three, var\_four)

*# 引数とそれ以外を区別するため、スペースを4つ(インデントをさらに)加える*

**def** long\_function\_name(

var\_one, var\_two, var\_three,

var\_four):

print(var\_one)

*# 突き出しインデントはインデントのレベルを深くする*

foo = long\_function\_name(

var\_one, var\_two,

var\_three, var\_four)

*# 間違い:*

*# 折り返された要素を縦に揃えない場合、1行目の引数は禁止*

foo = long\_function\_name(var\_one, var\_two,

var\_three, var\_four)

*# インデントが区別できないので、2行目以降でさらにインデントが必要*

**def** long\_function\_name(

var\_one, var\_two, var\_three,

var\_four):

print(var\_one)

複数行を継続したときにインデントする場合は、4つスペースを使うルールを守らなくても構いません。

任意:

*# 突き出しインデントの場合は、インデントに \*スペースを4つ使わなくてもよい\**

foo = long\_function\_name(

var\_one, var\_two,

var\_three, var\_four)

if 文の条件部分が、複数行にまたがって書かなければならないくらい十分に長い場合があります。この場合、2文字のキーワード(つまり、 if) の後にスペースを一つ置き、開き括弧を置くと、2行目以降の条件部分は通常スペース4つ分インデントされることになります。 if 文の中でネストされるインデントされたコードも通常スペース4つ分インデントされるので、ネストされたコードの固まりと条件部分が見た目上区別がつかなくなってしまう可能性があります。 この PEP は、 if 文に含まれるネストされたコードの部分と、継続された条件部分を区別するかどうか(またはどうやって区別するか)については立場を示しませんが、許容できるやり方はいくつかあります:

*# 追加のインデントをしない*

**if** (this\_is\_one\_thing **and**

that\_is\_another\_thing):

do\_something()

*# シンタックスのハイライトをサポートするエディタで区別するため*

*# コメントを追加する*

**if** (this\_is\_one\_thing **and**

that\_is\_another\_thing):

*# 両方の条件がtrueなので、処理を調整可能*

do\_something()

*# 継続された行の条件をインデントする*

**if** (this\_is\_one\_thing

**and** that\_is\_another\_thing):

do\_something()

(後で述べる、2項演算子の前で改行すべきか、後で改行すべきかについての議論も参照してください)

行を継続して 波括弧/ブラケット/括弧 を閉じる時は、「リストの最後の要素が置かれた行の、はじめのホワイトスペースでない文字の直下」に閉じる記号を置いても構いません。次のようにします:

my\_list = [

1, 2, 3,

4, 5, 6,

]

result = some\_function\_that\_takes\_arguments(

'a', 'b', 'c',

'd', 'e', 'f',

)

もしくは、閉じる記号を「継続された行のはじめの文字」に合わせて置いても構いません。次のようにします:

my\_list = [

1, 2, 3,

4, 5, 6,

]

result = some\_function\_that\_takes\_arguments(

'a', 'b', 'c',

'd', 'e', 'f',

)

タブか、スペースか?

スペースが好ましいインデントの方法です。

タブを使うのは、既にタブでインデントされているコードと一貫性を保つためだけです。

Python では、インデントにタブとスペースを混ぜることを禁止しています。

1行の長さ

すべての行の長さを、最大79文字までに制限しましょう。

(docstring やコメントのように) 構造に関する制約が少ないテキストのブロックについては、1行72文字までに制限すべきです。

エディタのウィンドウの幅を制限すると、複数のファイルを並べて開くことができ、二つのバージョンを隣り合ったカラムに表示するコードレビューツールを使うときにもうまくいきます。

ほとんどのツールのデフォルトの折り返し動作は、コードの見た目の構造を壊し、理解するのを難しくします。79文字という制限は、ウィンドウの幅を80に制限し、行を折り返すときにツールが行末にマーカーを置いたとしても、エディタに折り返す動作をさせない目的で選ばれています。

1行を79文字より長くするのを好むチームもあります。この問題について合意できるチームが独占的に、もしくは重点的にメンテナンスするコードについては、1行99文字まで制限を緩めてもOKです。ただし、コメントや docstring については72文字で折り返すようにすることが条件です。

Python の標準ライブラリは保守的なので、1行の文字数は79文字に制限することが必須です(docstring やコメントは72文字)。

長い行を折り返す好ましい方法は、Python が括弧やブラケット、波括弧の中では暗黙のうちに行を継続させることを利用することです。長い行は文を括弧で囲むことで、複数行に分割できます。行を継続させるには、バックスラッシュよりもこれらを使用すべきです。

バックスラッシュを使うのが適切な場合もあります。たとえば、Python 3.10 より前のバージョン限定ですが、長い複数の with 文は暗黙のうちに行を継続してくれません。この場合は、バックスラッシュが望ましいです:

**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())

(このように with 文が複数行にまたがって使われる場合の詳しい考え方については、 [条件が複数行にまたがるif文](https://pep8-ja.readthedocs.io/ja/latest/#if) の議論も参照してください)

バックスラッシュを使うのが好ましい別のケースとして assert 文を使う場合が挙げられます。

必ず、行を継続させた場合はインデントを適切に行うようにしてください。

2項演算子の前で改行すべきか、後で改行すべきか?

数十年の間、2項演算子の後で改行するスタイルが推奨されていました。しかし、このやり方は2つの理由で読みにくくなってしまいます。画面のいろんな場所に演算子が散らばってしまいがちです。つまり、それぞれの演算子が前の行の演算子から離れてしまいます。以下の例では、どのオペランドが加算されているのか、減算されているのかを知るために、眼を余計に動かさなければなりません:

*# 間違い:*

*# 演算子がオペランドと離れてしまっている*

income = (gross\_wages +

taxable\_interest +

(dividends - qualified\_dividends) -

ira\_deduction -

student\_loan\_interest)

この読みやすさの問題を解決するため、数学者や、数学の本の出版社は全く逆の規約に従っています。Donald Knuth は自身の *Computers and Typesetting* に関する一連の論文の中で、この伝統的なルールについて以下のように説明しています: "段落の中にある式は、常に2項演算子や関係演算子の後で改行しますが、構造化された式は常に2項演算子の前で改行します" [[3]](https://pep8-ja.readthedocs.io/ja/latest/#id47)

数学の伝統に従うと、通常はもっと読みやすいコードになります:

*# 正しい:*

*# 演算子とオペランドを一致させやすい*

income = (gross\_wages

+ taxable\_interest

+ (dividends - qualified\_dividends)

- ira\_deduction

- student\_loan\_interest)

*# 正しい:*

*# 演算子とオペランドを一致させやすい*

income = (gross\_wages

+ taxable\_interest

+ (dividends - qualified\_dividends)

- ira\_deduction

- student\_loan\_interest)

Python のコードでは、プロジェクトの内部で統一されているのであれば、2項演算子の前後どちらででも改行して構いません。新しいコードでは、Knuth のスタイルをお勧めします。

空行

トップレベルの関数やクラスは、2行ずつ空けて定義するようにしてください。

クラス内部では、1行ずつ空けてメソッドを定義してください。

関連する関数のグループを分けるために、2行以上空けても構いません(ただし控えめに)。 関連するワンライナーの場合は、空行を省略しても問題ありません。(例: ダミー実装)

関数の中では、ロジックの境目を示すために、空行を控えめに使うようにします。

Python は 用紙送りをあらわす Control-L (^L) 文字を空白文字として認めています。多くのツールはこの文字をページの区切りとして扱います。よって、ファイルの関連する部分を複数のページに分割する用途で、こうした文字を使っても構いません。ただし、Webベースのコードビューアやエディタの中には、Control-L を用紙送り文字として認識せず、違うグリフを表示するものもあるので注意してください。

ソースファイルのエンコーディング

Python のコアディストリビューションに含まれるコードは常に UTF-8 を使用すべきで、エンコーディング宣言を入れるべきではありません。

標準ライブラリでは、UTF-8 以外のエンコーディングを使うのはテスト目的に限るべきです。ASCII でない文字は控えめに使い、場所や人の名前にのみ使うのが望ましいです。ASCII でない文字をデータとして使う場合、z̯̯͡a̧͎̺l̡͓̫g̹̲o̡̼̘ のようなノイズが多い Unicode 文字や、BOM を入れるのを避けましょう。

Python の標準ライブラリの識別子には ASCII のみを使わなければなりませんし、適切な場合 (英語でない技術的な用語や略語が使われる場合が多くあります) はいつでも英単語を使うべきです。

世界中の人が利用するオープンソースプロジェクトは、これと似たポリシーを採用することを推奨します。

import

* import文は、通常は行を分けるべきです:
* *# 正しい:*
* **import** **os**
* **import** **sys**
* *# 悪い:*
* **import** **sys**, **os**

しかし、次のやり方はOKです:

*# 正しい:*

**from** **subprocess** **import** Popen, PIPE

* import文 は常にファイルの先頭、つまり モジュールコメントや docstring の直後、そしてモジュールのグローバル変数や定数定義の前に置くようにします。

import文 は次の順番でグループ化すべきです:

* 1. 標準ライブラリ
  2. サードパーティに関連するもの
  3. ローカルな アプリケーション/ライブラリ に特有のもの

上のグループそれぞれの間には、1行空白を置くべきです。

* 絶対import を推奨します。なぜなら、絶対import の方が通常は読みやすく、importシステムが正しく設定されなかった(たとえばパッケージ内部のディレクトリが sys.path で終わっていた) 場合でも、より良い振る舞いをする(または少なくともより良いエラーメッセージを出す)からです:
* **import** **mypkg.sibling**
* **from** **mypkg** **import** sibling
* **from** **mypkg.sibling** **import** example

しかしながら、明示的に相対importを使うことが許される場合があります。特に絶対importを使うと不必要に冗長になる複雑なパッケージレイアウトを扱う場合です。:

**from** **.** **import** sibling

**from** **.sibling** **import** example

標準ライブラリのコードは複雑なパッケージレイアウトを避け、常に絶対importを使うようにすべきです。

* クラスを含んだモジュールからクラスをimportする場合は、次のようにしても通常はOKです:
* **from** **myclass** **import** MyClass
* **from** **foo.bar.yourclass** **import** YourClass

このやり方で名前の衝突が起きたら、次のように明示的にimportします:

**import** **myclass**

**import** **foo.bar.yourclass**

そして "myclass.MyClass" や "foo.bar.yourclass.YourClass" を使います。

* ワイルドカードを使った import (from <module> import \*) は避けるべきです。なぜなら、どの名前が名前空間に存在しているかをわかりにくくし、コードの読み手や多くのツールを混乱させるからです。ワイルドカードを使った import を正当化できるユースケースがひとつあります。内部インターフェイスを公開APIとして再公開する場合 (たとえば、Pure Python の実装をオプションの高速化モジュールの内容で上書きし、どの定義が上書きされるかがあらかじめわからない場合) です。

名前をこのやり方で再公開する場合でも、公開インターフェイスと内部インターフェイスに関するガイドラインは有効です。

モジュールレベルの二重アンダースコア変数名

\_\_all\_\_, \_\_author\_\_, \_\_version\_\_ のような、モジュールレベルの "二重アンダースコア変数" (変数名の前後にアンダースコアが2つ付いている変数) は、モジュールに関する docstring の後、そして from \_\_future\_\_ *以外の* あらゆるimport文の前に置くべきです。Python はfuture importを、docstring 以外のあらゆるコードの前に置くように強制します:

*"""This is the example module.*

*This module does stuff.*

*"""*

**from** **\_\_future\_\_** **import** barry\_as\_FLUFL

\_\_all\_\_ = ['a', 'b', 'c']

\_\_version\_\_ = '0.1'

\_\_author\_\_ = 'Cardinal Biggles'

**import** **os**

**import** **sys**

文字列に含まれる引用符

Python では、単一引用符 ' で囲まれた文字列と、二重引用符 " で囲まれた文字列は同じです。この PEP では、どちらを推奨するかの立場は示しません。どちらを使うかのルールを決めて、守るようにして下さい。単一引用符 や 二重引用符 が文字列に含まれていた場合は、文字列中でバックスラッシュを使うことを避けるため、もう一方の引用符を使うようにしましょう。可読性が向上します。

三重引用符 で文字列を囲むときは、[**PEP 257**](https://peps.python.org/pep-0257/) での docstring に関するルールと一貫させるため、常に二重引用符 """ を使うようにします。

式や文中の空白文字

イライラの元

次の場合に、余計な空白文字を使うのはやめましょう:

* 括弧やブラケット、波括弧 のはじめの直後と、終わりの直前:
* *# 正しい:*
* spam(ham[1], {eggs: 2})
* *# 間違い:*
* spam( ham[ 1 ], { eggs: 2 } )
* 末尾のカンマと、その後に続く閉じカッコの間:
* *# 正しい:*
* foo = (0,)
* *# 間違い:*
* bar = (0, )
* カンマやセミコロン、コロンの直前:
* *# 正しい:*
* **if** x == 4: print(x, y); x, y = y, x
* *# 間違い:*
* **if** x == 4 : print(x , y) ; x , y = y , x
* しかし、スライスではコロンは二項演算子のように振る舞います。よって、(コロンは優先度が最も低い演算子として扱われるので)両側に同じ数(訳注: 無しでも可だと思われる)のスペースを置くべきです。拡張スライスでは、両側に同じ数のスペースを置かなければなりません。例外: スライスのパラメータが省略された場合は、スペースも省略されます。:
* *# 正しい:*
* 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]
* *# 間違い:*
* ham[lower + offset:upper + offset]
* ham[1: 9], ham[1 :9], ham[1:9 :3]
* ham[lower : : step]
* ham[ : upper]
* 関数呼び出しの引数リストをはじめる開き括弧の直前:
* *# 正しい:*
* spam(1)
* *# 間違い:*
* spam (1)
* インデックスやスライスの開き括弧の直前:
* *# 正しい:*
* dct['key'] = lst[index]
* *# 間違い:*
* dct ['key'] = lst [index]
* 代入(や他の)演算子を揃えるために、演算子の周囲に1つ以上のスペースを入れる:
* *# 正しい:*
* x = 1
* y = 2
* long\_variable = 3
* *# 間違い:*
* x = 1
* y = 2
* long\_variable = 3

その他の推奨事項

* 行末に余計な空白文字を残さないようにしましょう。通常それは目に見えないため、混乱のもとになるかもしれません。たとえば、バックスラッシュの後にスペースをひとつ入れて改行してしまうと、行を継続すると見なされません。エディタによっては行末の余計な空白文字を保存しないものもありますし、多くのプロジェクト (CPythonもそうです) ではコミット前のフックでそれを拒否するように設定しています。
* 次の2項演算子は、両側に常にひとつだけスペースを入れましょう: 代入演算子 (=), 拡張代入演算子 (+=, -= など.), 比較演算子 (==, <, >, !=, <>, <=, >=, in, not in, is, is not), ブール演算子 (and, or, not).
* 優先順位が違う演算子を扱う場合、優先順位が一番低い演算子の両側にスペースを入れることを考えてみましょう。入れるかどうかはあなたの判断にお任せしますが、二つ以上のスペースを絶対に使わないでください。そして、2項演算子の両側には、常に同じ数の空白文字を入れてください。:
* *# 正しい:*
* i = i + 1
* submitted += 1
* x = x\*2 - 1
* hypot2 = x\*x + y\*y
* c = (a+b) \* (a-b)
* *# 間違い:*
* i=i+1
* submitted +=1
* x = x \* 2 - 1
* hypot2 = x \* x + y \* y
* c = (a + b) \* (a - b)
* 関数アノテーションは、 コロンに関する通常のルール(訳注:コロンの前には余計なスペースを入れない)を守るようにしつつ、 -> 演算子がある場合、その両側には常にスペースを入れるようにしましょう。(関数アノテーションについて詳しくは、 [関数アノテーション](https://pep8-ja.readthedocs.io/ja/latest/#id42) も参照してください):
* *# 正しい:*
* **def** munge(input: AnyStr): ...
* **def** munge() -> PosInt: ...
* *# 間違い:*
* **def** munge(input:AnyStr): ...
* **def** munge()->PosInt: ...
* *アノテーションされていない* 関数の引数におけるキーワード引数や、デフォルトパラメータを示す = の両側にスペースを入れてはいけません:
* *# 正しい:*
* **def** complex(real, imag=0.0):
* **return** magic(r=real, i=imag)
* *# 間違い:*
* **def** complex(real, imag = 0.0):
* **return** magic(r = real, i = imag)

しかし、デフォルト値をもった引数アノテーションと組み合わせる場合、 = の前後にはスペースを入れるようにしてください:

*# 正しい:*

**def** munge(sep: AnyStr = **None**): ...

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

*# 間違い:*

**def** munge(input: AnyStr=**None**): ...

**def** munge(input: AnyStr, limit = 1000): ...

* 複合文 (一行に複数の文を入れること) は一般的に推奨されません。:
* *# 正しい:*
* **if** foo == 'blah':
* do\_blah\_thing()
* do\_one()
* do\_two()
* do\_three()

やらない方が良い:

*# 間違い:*

**if** foo == 'blah': do\_blah\_thing()

do\_one(); do\_two(); do\_three()

* if/for/while と 短い文を同じ行に置くことがOKな場合もありますが、複合文を置くのはやめてください。また、複合文でできた長い行を折り返すのもやめましょう！

やらない方が良い:

*# Wrong:*

**if** foo == 'blah': do\_blah\_thing()

**for** x **in** lst: total += x

**while** t < 10: t = delay()

絶対やってはいけない:

*# 間違い:*

**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()

末尾にカンマを付けるべき場合

末尾にカンマを付けるかどうかは、通常は任意です。但し、要素数がひとつのタプルを作るときは例外的に必須です。混乱を防ぐため、要素数がひとつのタプルを作るときは、(技術的には冗長ですが) 括弧で囲むことを推奨します。:

*# 正しい:*

FILES = ('setup.cfg',)

*# 間違い:*

FILES = 'setup.cfg',

末尾のカンマは冗長ですが、便利な場合もあります。たとえば値や引数、もしくはimportされた値のリストが繰り返し展開されることが期待される場合や、バージョン管理システムを使っている場合です。それぞれの値(など)を一行にひとつずつ置き、末尾にカンマをひとつ追加し、その次の行を 閉じ括弧/角括弧/中括弧 で閉じるというのがひとつのパターンです。しかし、区切りの終わりを示す目的で、同じ行の末尾にカンマを付けることは意味がありません(但し、既に示した、要素数がひとつのタプルの場合は除きます)。:

*# 正しい:*

FILES = [

'setup.cfg',

'tox.ini',

]

initialize(FILES,

error=**True**,

)

*# 間違い:*

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

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

コメント

コードと矛盾するコメントは、コメントしないことよりタチが悪いです。コードを変更した時は、コメントを最新にすることをいつも優先させてください！

コメントは複数の完全な文で書くべきです。はじめの単語はそれが小文字で始まる識別子でない限り、大文字にすべきです(間違ってもその識別子の大文字小文字を変更しないでね！)。

ブロックコメントは一般的にひとつかそれ以上の段落からなり、段落は複数の完全な文からできています。そしてそれぞれの文はピリオドで終わります。

コメントが2つ以上の文からなる場合、文の終わりのピリオドの後は、ひとつか2つスペースを入れるべきです。ただし、最後の文を除きます。

コメントを書くときは、あなたが今書いている言葉の他の話者から見ても、明快かつわかりやすいコメントにしましょう。

英語を話さない国出身の Python プログラマの方々へ：あなたのコードが、自分の言葉を話さない人に 120% 読まれないと確信していなければ、コメントを英語で書くようにお願いします。

ブロックコメント

ブロックコメントは、一般的にその後に続くいくつか（またはすべて）のコードに適用され、そのコードと同じレベルにインデントされます。ブロックコメントの各行は (コメント内でインデントされたテキストでない限り) # とスペースひとつではじまります。

ブロックコメント内の段落は、# だけを含んだ1行で区切るようにします。

インラインコメント

インラインコメントは控えめに使いましょう。

インラインコメントは、文と同じ行に書くコメントです。文とインラインコメントの間は、少なくとも二つのスペースを置くべきです。インラインコメントは # とスペースひとつから始めるべきです。

自明なことを述べている場合、インラインコメントは不要ですし、邪魔です。次のようなことはしないでください:

x = x + 1 *# Increment x*

しかし次のように、役に立つ場合もあります:

x = x + 1 *# 境目を補う*

ドキュメンテーション文字列

良いドキュメンテーション文字列(別名 "docstrings")を書くための規約は、[**PEP 257**](https://peps.python.org/pep-0257/) にまとめられています。

* すべての公開されているモジュールや関数、クラス、メソッドの docstring を書いてください。docstring は公開されていないメソッドには不要ですが、そのメソッドが何をしているのかは説明すべきです。このコメントは def の行のあとに置くべきです。
* [**PEP 257**](https://peps.python.org/pep-0257/) は良い docstring の規約です。もっとも重要なのは、複数行の docstring は """ だけからなる行で閉じることです:
* *"""Return a foobang*
* *Optional plotz says to frobnicate the bizbaz first.*
* *"""*
* docstring が1行で終わる場合は、同じ行を """ で閉じるようにしてください:
* *"""Return an ex-parrot."""*

命名規約

Python のライブラリで採用されている命名規約はちょっと面倒です。よって、この命名規約を完全に一貫したものにするつもりはありません - とはいえ、現在推奨している命名規約をここで説明します。新しいモジュールとパッケージ (サードパーティのフレームワークを含む) はこの規約に従って書くべきです。しかし、既にあるライブラリが異なるスタイルを採用している場合は、内部を一貫させることが望ましいです。

一番重要な原則

公開されている API の一部としてユーザーに見える名前は、実装よりも使い方を反映した名前にすべきです。

実践されている命名方法

命名のやり方には多くのバリエーションがあります。どういう目的で使われているのかは別として、どんなやり方が使われているのかがわかります。

よく知られたやり方として、次のものが挙げられます:

* b (小文字1文字)
* B (大文字1文字)
* lowercase
* lower\_case\_with\_underscores
* UPPERCASE
* UPPER\_CASE\_WITH\_UNDERSCORES
* CapitalizedWords (CapWords, または CamelCase - 文字がデコボコに見えることからこう呼ばれます [[4]](https://pep8-ja.readthedocs.io/ja/latest/#id48))。StudlyCaps という呼び名でも知られています。

注意: CapWords の中で頭字語 [[6]](https://pep8-ja.readthedocs.io/ja/latest/#fn-hu) を含める場合、頭字語の全ての文字を大文字にします。つまりこのやり方だと、HttpServerError より HTTPServerError の方が良いということになります。

* mixedCase (はじめの文字が小文字である点が、CapitalizedWords と違います！)
* Capitalized\_Words\_With\_Underscores (醜い！)

関連する名前の集まりに、短い一意なプレフィックスを付けるやり方もあります。Python ではこのやり方を多く使っているわけではありませんが、完全を期すために紹介しておきます。たとえば、 os.stat() 関数は伝統的に st\_mode, st\_size, st\_mtime などの名前からなるタプルを返します。 (これは、 POSIX システムコールが返す構造体のフィールドとの関連を強調するために使われており、POSIXシステムコール に馴染んだプログラマを助けてくれます)

X11ライブラリは、公開されている関数全てに「X」を付けています。Python では一般的にこのやり方は不要だと考えられています。なぜなら、Python の属性やメソッドの名前の前にはクラス名が付きますし、関数名の前にはモジュール名が付くからです。

それに加えて、次のようにアンダースコアを名前の前後に付ける特別なやり方が知られています(これらに大文字小文字に関する規約を組み合わせるのが一般的です):

* \_single\_leading\_underscore: "内部でだけ使う" ことを示します。 たとえば from M import \* は、アンダースコアで始まる名前のオブジェクトをimportしません。
* single\_trailing\_underscore\_: Python のキーワードと衝突するのを避けるために使われる規約です。例を以下に挙げます:
* tkinter.Toplevel(master, class\_='ClassName')
* \_\_double\_leading\_underscore: クラスの属性に名前を付けるときに、名前のマングリング機構を呼び出します (クラス Foobar の \_\_boo という名前は \_FooBar\_\_boo になります。以下も参照してください)
* \_\_double\_leading\_and\_trailing\_underscore\_\_: ユーザーが制御する名前空間に存在する "マジック"オブジェクト または "マジック"属性です。 たとえば \_\_init\_\_, \_\_import\_\_, \_\_file\_\_ が挙げられます。この手の名前を再発明するのはやめましょう。ドキュメントに書かれているものだけを使ってください。

守るべき命名規約

こんな名前は嫌だ

単一の文字 'l' (小文字のエル)、'O' (大文字のオー)、'I'(大文字のアイ) を決して変数に使わないでください。

フォントによっては、これらの文字は数字の1や0と区別が付かない場合があります。'l'(小文字のエル) を使いたくなったら、'L' を代わりに使いましょう。

ASCII との互換性

標準ライブラリで使われる識別子は、[**PEP 3131**](https://peps.python.org/pep-3131/) の [**policy section**](https://peps.python.org/pep-3131/#policy-specification) にあるとおり、 ASCII と互換性がなければなりません。

パッケージとモジュールの名前

モジュールの名前は、全て小文字の短い名前にすべきです。読みやすくなるなら、アンダースコアをモジュール名に使っても構いません。Python のパッケージ名は、全て小文字の短い名前を使うべきですが、アンダースコアを使うのは推奨されません。

C や C++ で書かれた Python の拡張モジュールに、高レベルの (例：オブジェクト指向的な) インターフェイスを提供する Python モジュールが付いている場合は C/C++ のモジュールはアンダースコアで始まります (例： \_socket)

クラスの名前

クラスの名前には通常 CapWords 方式を使うべきです。

主に callable として使われる、ドキュメント化されたインターフェイスの場合は、クラスではなく関数向けの命名規約を使っても構いません。

Python にビルドインされている名前には別の規約があることに注意してください： ビルトインされている名前のほとんどは、単一の単語(または、二つの単語が混ざったもの) ですが、例外的に CapWords 方式が使われている名前や定数も存在しています。

型変数の名前

[**PEP 484**](https://peps.python.org/pep-0484/) で導入された型変数の名前には、通常 CapWords 方式を使うべきです。また、 T や AnyStr や Num のような短い名前が好ましいです。 共変や反変の振る舞いをする変数を宣言するために \_co や \_contra のような名前を変数の末尾に付け加えることを推奨します:

**from** **typing** **import** TypeVar

VT\_co = TypeVar('VT\_co', covariant=**True**)

KT\_contra = TypeVar('KT\_contra', contravariant=**True**)

例外の名前

例外はクラスであるべきです。よって、クラスの命名規約がここにも適用されます。しかし、(その例外が実際にエラーである場合は) 例外の名前の最後に "Error" をつけるべきです 。

グローバル変数の名前

(ここで言う「グローバル変数」はモジュールレベルでグローバルという意味だと思いたいですが) ここで示す規約は、関数レベルのものについても同じです。

from M import \* 方式でimportされるように設計されているモジュールは、 グローバル変数をエクスポートするのを防ぐため \_\_all\_\_ の仕組みを使うか、エクスポートしたくないグローバル変数の頭にアンダースコアをつける以前からある規約を使うべきです (こうすることで、これらのグローバル変数は「モジュールレベルで公開されていない」ことを開発者が示したいかもしれません)。

関数や変数の名前

関数の名前は小文字のみにすべきです。また、読みやすくするために、必要に応じて単語をアンダースコアで区切るべきです。

変数の名前についても、関数と同じ規約に従います。

mixedCase が既に使われている (例: threading.py) 場合にのみ、互換性を保つために mixedCase を許可します。

関数やメソッドに渡す引数

インスタンスメソッドのはじめの引数の名前は常に self を使ってください。

クラスメソッドのはじめの引数の名前は常に cls を使ってください。

関数の引数名が予約語と衝突していた場合、アンダースコアを引数名の後ろに追加するのが一般的には望ましいです。衝突した名前を変更しようとして、略語を使ったりスペルミスをするよりマシです。よって、 class\_ は clss より好ましいです。 (多分、同義語を使って衝突を避けるのがよいのでしょうけど)

メソッド名とインスタンス変数

関数の命名規約を使ってください。つまり、名前は小文字のみにして、読みやすくするために必要に応じて単語をアンダースコアで区切ります。

公開されていないメソッドやインスタンス変数にだけ、アンダースコアを先頭に付けてください。

サブクラスと名前が衝突した場合は、Python のマングリング機構を呼び出すためにアンダースコアを先頭に二つ付けてください。

Python はアンダースコアが先頭に二つ付いた名前にクラス名を追加します。つまり、クラス Foo に \_\_a という名前の属性があった場合、この名前は Foo.\_\_a ではアクセスできません (どうしてもアクセスしたいユーザーは Foo.\_Foo\_\_a とすればアクセスできます)。一般的には、アンダースコアを名前の先頭に二つ付けるやり方は、サブクラス化されるように設計されたクラスの属性が衝突したときに、それを避けるためだけに使うべきです。

注意: アンダースコアを名前の先頭に二つ付けるやり方については、別の議論があります。 (下を参照)

定数

定数は通常モジュールレベルで定義します。全ての定数は大文字で書き、単語をアンダースコアで区切ります。例として MAX\_OVERFLOW や TOTAL があります。

継承の設計

クラスのメソッドやインスタンス変数 (まとめて "属性" といいます) を公開するかどうかをいつも決めるようにしましょう。よくわからないなら、公開しないでおきます。なぜなら、公開されている属性を非公開にすることよりも、非公開の属性を公開することの方がずっと簡単だからです。

クラスのユーザーは、公開されている(public)属性に対して、開発者が後方互換性を壊す変更をしないことを期待します。公開されていない(non-public)属性は、サードパーティに使われてることを意図していないものです。つまり、非公開の属性に変更されない保証はありませんし、削除されない保証すらありません。

ここでは "private" という用語を使っていません。なぜなら、Python の世界で本当の意味で private なものは存在しない (実現するには通常は不要なほどの多くの作業が必要です) からです。

別の属性のカテゴリとして "サブクラスで実装されるAPI" (Python以外の言語では "protected" と呼ばれます) があります。クラスによっては、継承されることでクラスを拡張したり、クラスの振る舞いの一部を変えられるように設計されているものがあります。このようなクラスを設計する場合、どの属性が公開されるか、どの属性が "サブクラスで実装されるAPI" なのか、そしてどれが基底クラスでだけ本当に使われるのかを明示的に決めるようにしましょう。

これらのことを念頭に置くと、Pythonic なガイドラインは以下のようになります:

* 公開されている(public)属性の先頭にはアンダースコアを付けない
* もしあなたが公開している属性の名前が予約語と衝突する場合は、属性の名前の直後にアンダースコアを追加します。省略語を使ったり、スペルミスをするよりはマシです。 (しかし、このルールに関わらず、'cls' という名前はクラスを示す変数や引数、特にクラスメソッドのはじめの引数として望ましいものです)

注意1: 上のクラスメソッドの引数に関する推奨事項も参照してください。

* 公開する属性をシンプルにするには、複雑なアクセサやミューテータ(訳注:内部状態を変更するメソッド) を公開せず、属性の名前だけを公開するのがベストです。そういった属性に関数的な振る舞いが必要になった場合でも、Python はそういった拡張を将来簡単に行える手段を提供していることを覚えておきましょう。この場合は、関数呼び出しの実装をシンプルなデータアクセスの文法で隠すために、プロパティを使います。

注意1: 関数的な振る舞いは、副作用がない状態を保つようにしましょう。しかし、キャッシュのような副作用は一般的に問題ありません。

注意2: 計算コストが高く付く処理でプロパティを使うのはやめましょう。この機能を使うと、属性を呼び出す側は、この演算のコストが（比較的）安いものだと思ってしまいます。

* サブクラス化して使うクラスがあるとします。サブクラスで使って欲しくない属性があった場合、その名前の最後ではなく、先頭にアンダースコアを二つ付けることを検討してみましょう。これによって Python のマングリングアルゴリズムが呼び出され、その属性にはクラス名が付加されます。これはサブクラスにうっかり同名の属性が入ってしまうことによる属性の衝突を避けるのに役立ちます。

注意1: マングリングされる名前は、単なるクラス名であることに注意して下さい。よって、サブクラスで親クラスと同じ名前のクラス名と属性名を選んだ場合、衝突は回避できません。

注意2: 名前のマングリングは、ユースケースによっては不便な場合もあります。たとえば \_\_getattr\_\_() を使ったり、デバッグを行う場合です。しかし、名前のマングリングアルゴリズムはきちんとドキュメント化されているので、簡単に手動で実行できます。

注意3: 皆がマングリングを好きだとは限りません。うっかり名前を衝突させてしまうリスクを避けることと、Pythonの上級者が使う可能性があることとのバランスを考えましょう。

公開インターフェイスと内部インターフェイス

後方互換性は公開されているインターフェイスにのみ保証されます。よって、公開インターフェイスと内部インターフェイスをユーザーが明確に区別できることが重要になります。

ドキュメントが明示的に一時的な、もしくは互換性を保つ義務が免除された内部インターフェイスだと宣言していない限り、ドキュメント化されたインターフェイスは公開インターフェイスと見なされます。ドキュメント化されていないインターフェイスはすべて内部的なものと見なすべきです。

イントロスペクションがうまく機能するようにするため、モジュールは公開しているAPIを \_\_all\_\_ 属性を使ってすべて宣言すべきです。 \_\_all\_\_ 属性を空にすると、そのモジュールには公開しているAPIはないということになります。

\_\_all\_\_ 属性を適切に設定したとしても、内部インターフェイス (パッケージ、モジュール、クラス、関数、属性、その他の名前) は名前の前にアンダースコアをひとつ付けるべきです。

あるインターフェイスが含まれている名前空間（パッケージ、モジュール、クラス）が内部的なものだと見なされる場合は、そのインターフェイスも内部インターフェイスと見なされます。

importされた名前は、常に実装の詳細を表現していると見なすべきです。他のモジュールは、importされた名前に間接的にアクセスされることによって生じる動作に依存してはいけません。但し、それが明示的にドキュメント化されたモジュールAPIの一部の場合、 たとえば os.path や パッケージのサブモジュールの機能を公開している \_\_init\_\_ モジュール を除きます。

プログラミングに関する推奨事項

* 他のPython実装 (PyPy, Jython, IronPython, Cython, Psyco など) で不利にならないようなコードを書くべきです。

たとえば a += b や a = a + b のように、データを直接置き換える CPython の効率的な実装に依存しないでください。この最適化は CPython の場合でも弱い(いくつかの型でしか機能しません)ですし、リファレンスカウントが入っていないPython実装には存在しません。ライブラリの中でパフォーマンスに敏感な部分には、 ''.join() を代わりに使うべきです。このやり方であれば、様々なPython実装で、文字列の連結が線形時間で終わることを保証してくれます。

* None のようなシングルトンと比較をする場合は、常に is か is not を使うべきです。絶対に等値演算子を使わないでください。

また、 本当は if x is not None と書いているつもりで、 if x と書いている場合は注意してください - たとえば、デフォルトの値がNoneになる変数や引数に、何かしら別の値が設定されているかどうかをテストする場合です。この「別の値」は、ブール型のコンテクストでは False と評価される(コンテナのような)型かもしれませんよ！

* not ... is ... ではなく、 is not 演算子を使いましょう。これらは機能的に同じですが、後者の方が読みやすく、好ましいです。:
* *# 正しい:*
* **if** foo **is** **not** **None**:
* *# 間違い:*
* **if** **not** foo **is** **None**:
* 拡張比較(rich comparion)を使って並び替えを実装する場合、特定の比較を実行するだけの他のコードに依存するよりはむしろ、全ての演算 (\_\_eq\_\_, \_\_ne\_\_, \_\_lt\_\_, \_\_le\_\_, \_\_gt\_\_, \_\_ge\_\_) を実装するのがベストです。

必要な作業を最小の労力で行えるように、 functools.total\_ordering() デコレータが存在しない比較メソッドを自動生成するツールを提供しています。

[**PEP 207**](https://peps.python.org/pep-0207/) は、Python では 反射律 *が* 想定されていると述べています。つまり、インタプリタは y > x と x < y, y >= x と x <= y がそれぞれ交換可能であり、x == y と x != y の引数が交換可能だということです。 sort() と min() 演算は < を確実に使いますし、 max() 関数は確実に > 演算子を使います。しかし、他のコンテクストで混乱が起きないように6つの演算を全て実装するのがベストです。

* ラムダ式を直接識別子に結びつける代入文を書くのではなくて、常に def 文を使いましょう。:
* *# 正しい:*
* **def** f(x): **return** 2\*x
* *# 間違い:*
* f = **lambda** x: 2\*x

はじめの書き方は、結果として生成される関数オブジェクトの名前が、ラムダではなくて f であると明示的に述べています。これは traceback や文字列表現を使うときに役立ちます。代入文を使うと、ラムダ式が提供できる唯一の利点(つまり、大きな式に埋め込めること)を消してしまいます。

* BaseException ではなくて、 Exception から例外を派生させるようにしましょう。 BaseException を直接継承する方法は、例外をキャッチするのが殆どの場合不適切な場合向けに予約されています。

例外の階層は、例外が投げられる場所ではなく、*キャッチする* コードが必要そうなコードの特徴に基づいて設計すべきです。"問題が起きました" と言うだけではなく、プログラム的に "何が起こった？" のか、という質問に答えるようにしましょう (ビルトイン例外の階層から学んだこの教訓の例が [**PEP 3151**](https://peps.python.org/pep-3151/) にあります。参照してください)。

クラスの命名規約がここにあてはまります。しかし、例外がエラーである場合は、例外クラスの名前の最後に "Error" を付けるべきです。ローカルに閉じていないフローの制御や、他のシグナルを送信する用途に使う例外については、特別なサフィックスは不要です。

* 例外チェインを適切に使いましょう。オリジナルの traceback を失わず明示的に例外を入れ替えるために raise X from Y を使うべきです。

内部の例外をわざと入れ替える (raise X from None を使います) ときは、例外の詳細を新しい例外にも伝えるようにしましょう（たとえば KeyError を AttributeError に変換するときに、KeyError が持っていた属性の名前を保護したりとか、オリジナルの例外が持っていたエラーメッセージを新しい例外にも埋め込む、みたいなことです）。

* 例外をキャッチする時は、可能なときはいつでも、例外を指定しない生の except: ではなく、特定の例外を指定するようにしましょう:
* **try**:
* **import** **platform\_specific\_module**
* **except** ImportError:
* platform\_specific\_module = **None**

生の except: は SystemExit や KeyboardInterrupt 例外もキャッチしてしまうため、プログラムを Control-C で中断することが難しくなりますし、他の問題をもみ消してしまうかもしれません。シグナルのエラーもすべて例外でキャッチしたい場合は、 except Exception: を使ってください (生の except は except BaseException: と同義です)。

よく使うやり方は、生の 'except' を使う場合を次の二つに限ることです:

* 1. 例外ハンドラが traceback を出力するかロギングする場合。ユーザーは少なくともエラーが起きたことがわかります。
  2. リソースの後始末が必要な場合、後始末をしたのちに raise を使って上流に例外を伝播させるとき。 この手の問題は、 try...finally の方が適切かもしれません。
* オペレーティングシステムのエラーをキャッチするときは、Python 3.3 以降では errno の値を調べるのではなく、新しいオペレーティングシステム関連のエラー階層を明示的に使うのが望ましいです。
* それに加えて、 すべての try/except について、try で囲む範囲を必要最小限のコードに限るようにしましょう。繰り返しますが、これはバグのもみ消しを防いでくれます。:
* *# 正しい:*
* **try**:
* value = collection[key]
* **except** KeyError:
* **return** key\_not\_found(key)
* **else**:
* **return** handle\_value(value)
* *# 間違い:*
* **try**:
* *# try で囲む処理が大きすぎる！*
* **return** handle\_value(collection[key])
* **except** KeyError:
* *# handle\_value() が発生させる KeyError もキャッチする*
* **return** key\_not\_found(key)
* リソースがコードの特定の部分だけで使われる場合、 使った後すぐ信頼できるやり方で後始末ができるように with 文を使いましょう。 try/finally 文でも問題ありません。
* コンテキストマネージャーは、リソースの取得や解放以外のことをするときは常に、別の関数やメソッドを通じて呼び出すべきです。 例を挙げます:
* *# 正しい:*
* **with** conn.begin\_transaction():
* do\_stuff\_in\_transaction(conn)
* *# 間違い:*
* **with** conn:
* do\_stuff\_in\_transaction(conn)

後者の例は、 \_\_enter\_\_ と \_\_exit\_\_ メソッドがトランザクションの後に接続を閉じる以外に何をするかがまったく分かりません。明示的にそれを示すのがこの場合は重要です。

* return文は一貫した書き方をしましょう。関数の中の全てのreturn文は式を返すか、全く何も返さないかのどちらかにすべきです。式を返しているreturn文が関数の中にある場合、値を何も返さないreturn文は 明示的に return None と書くべきですし、(到達可能であれば)return文を関数の最後に明示的に置くべきです。:
* *# 正しい:*
* **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)
* *# 間違い:*
* **def** foo(x):
* **if** x >= 0:
* **return** math.sqrt(x)
* **def** bar(x):
* **if** x < 0:
* **return**
* **return** math.sqrt(x)
* 文字列に特定のプレフィックスやサフィックスがついているかをチェックするには、文字列のスライシングではなく ''.startswith() と ''.endswith() を使いましょう。

startswith() と endswith() を使うと、綺麗で間違いが起こりにくいコードになります:

*# 正しい:*

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

*# 間違い:*

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

* オブジェクトの型の比較は、型を直接比較するかわりに、常に isinstance() を使うようにすべきです。:
* *# 正しい:*
* **if** isinstance(obj, int):
* *# 間違い:*
* **if** type(obj) **is** type(1):
* シーケンス (文字列, リスト, タプル) については、 空のシーケンスが False であることを利用しましょう。:
* *# 正しい:*
* **if** **not** seq:
* **if** seq:
* *# 間違い:*
* **if** len(seq):
* **if** **not** len(seq):
* 行末の空白文字に依存した文字列リテラルを書かないでください。そういった空白文字は視覚的に判別することができず、エディタによっては (つい最近では reindent.py も) 自動で削除するものもあります。
* ブール型の値と True や False を比較するのに == を使うのはやめましょう。:
* *# 正しい:*
* **if** greeting:
* *# 間違い:*
* **if** greeting == **True**:

もっと悪い:

*# 間違い:*

**if** greeting **is** **True**:

* try...finally の組み合わせの中で、finally の外に脱出する制御構文 return/break/continue を使うのは推奨されません。なぜなら、このような構文は finally の中から伝播する例外を暗黙のうちにキャンセルしてしまうからです。:
* *# 間違い:*
* **def** foo():
* **try**:
* 1 / 0
* **finally**:
* **return** 42

関数アノテーション

[**PEP 484**](https://peps.python.org/pep-0484/) が採用されたので、関数アノテーションに関するスタイルのルールも変更されました。

* 関数アノテーションを使うコードは、[**PEP 484**](https://peps.python.org/pep-0484/) に記された文法に従うべきです。(ただし、アノテーションの書式については、いくつかの推奨事項を [その他の推奨事項](https://pep8-ja.readthedocs.io/ja/latest/#id17) で説明しています)
* この PEP では、以前アノテーションの使い方について実験することを勧めていましたが、もはや推奨されません。
* しかし、標準ライブラリ以外では、[**PEP 484**](https://peps.python.org/pep-0484/) のルールの範囲内で実験することが推奨されています。たとえば、大規模なサードパーティのライブラリやアプリケーションを [**PEP 484**](https://peps.python.org/pep-0484/) の型アノテーションの書式を使ってマークアップし、これらのアノテーションを追加するのがどれだけ簡単だったかをレビューしたり、アノテーションがある場合にコードの理解度があがるかどうかを観察すること、などです。
* Python の標準ライブラリでは、こうしたアノテーションの採用は控えめにすべきですが、新しいコードや大規模なリファクタリングの際には使っても構いません。
* 関数アノテーションを [**PEP 484**](https://peps.python.org/pep-0484/) とは違うスタイルで使いたいコードについては、次のようなコメントを付けておくことを推奨します:
* *# type: ignore*

これをファイルの先頭あたりに書いておきます。こうすることで、型チェックのプログラムにすべてのアノテーションを無視するように伝えます。([**PEP 484**](https://peps.python.org/pep-0484/) では、型チェックプログラムを黙らせるきめの細かい方法が説明されています)

* linter や 型チェックプログラム のようなツールは Pythonインタプリタ とは別のツールですし、使うかどうかも任意です。Pythonインタプリタは、デフォルトでは型チェックによるメッセージを表示しませんし、アノテーションによって振る舞いを変えることもありません。
* 型チェックを望まない人は、無視することも自由です。しかし、サードパーティーライブラリパッケージのユーザーは、パッケージに対して型チェックを実行したいと思うかもしれません。こうした目的で、[**PEP 484**](https://peps.python.org/pep-0484/) はスタブファイル、つまり、対応する .py ファイルの設定に応じて型チェッカーが読み取る .pyi ファイルを使うことを推奨しています。 スタブファイルはライブラリとともに配布することもできますし、(ライブラリの作者の許可を得て) typeshed repo [[5]](https://pep8-ja.readthedocs.io/ja/latest/#id49) でパッケージとは別に配布することもできます。

変数アノテーション

[**PEP 526**](https://peps.python.org/pep-0526/) で、変数アノテーションが導入されました。変数アノテーションで推奨されるスタイルは、上で説明している関数アノテーションのそれに似ています:

モジュールレベルの変数や、クラス変数、インスタンス変数、そしてローカル変数に対するアノテーションは、コロンの後にひとつスペースを入れるべきです。

コロンの前には、スペースを入れてはいけません。

変数の右辺で代入を行う場合は、等号の前後にひとつだけスペースを入れるべきです。:

*# 正しい:*

code: int

**class** **Point**:

coords: Tuple[int, int]

label: str = '<unknown>'

*# 間違い:*

code:int *# コロンの後にスペースがない*

code : int *# コロンの前にスペース*

**class** **Test**:

result: int=0 *# 等号の前後にスペースがない*

[**PEP 526**](https://peps.python.org/pep-0526/) は Python 3.6 で採用されましたが、変数アノテーションの文法は、全てのバージョンの Python のスタブファイルで使われるのが望ましいです (詳細は [**PEP 484**](https://peps.python.org/pep-0484/) を参照して下さい)

**脚注**

[[1](https://pep8-ja.readthedocs.io/ja/latest/#id6)]

*突き出しインデント* は、はじめの行以外の全ての行をインデントするDTPのやり方です。Python の文脈では、括弧で囲む文の開き括弧を行の終わりに置いて、残りの行を閉じ括弧までインデントするスタイルのことを言います。

[[6](https://pep8-ja.readthedocs.io/ja/latest/#id27)]

頭字語は、NATO (=North Atlantic Treaty Organization 北大西洋条約機構) のように、頭文字を並べたものがひとつの単語として扱われているものです。個々の頭文字をアルファベットとして発音するもの (例: U.S = United States) は該当しないとされています。

参考文献

[[2](https://pep8-ja.readthedocs.io/ja/latest/#id2)]

Barry's GNU Mailman style guide <http://barry.warsaw.us/software/STYLEGUIDE.txt>

[[3](https://pep8-ja.readthedocs.io/ja/latest/#id10)]

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

[[4](https://pep8-ja.readthedocs.io/ja/latest/#id26)]

<http://www.wikipedia.com/wiki/CamelCase>

[[5](https://pep8-ja.readthedocs.io/ja/latest/#id43)]

Typeshed repo <https://github.com/python/typeshed>

著作権

この文書は パブリックドメイン に置かれています。