PEP 8 – Style Guide for Python Code

Author:
Guido van Rossum <guido at python.org>, Barry Warsaw <barry at python.org>, Alyssa Coghlan <ncoghlan at gmail.com>
Created:
05-Jul-2001

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

PEP 8 – Guia de estilo para o código Python

Introdução

Este documento fornece as diretrizes para a codificação da linguagem Python e que compõe a biblioteca padrão dos principais códigos para linguagem Python. Este documento e o PEP257 foram adaptados do manuscrito original do Guia de Estilo Python do Guido e com algumas inclusões do Guia de Estilo do Barry.

Esse guia de estilo de linguagem muda com o tempo à medida que novos códigos são inseridos e os códigos anteriores se tornam obsoletos com a mudança na linguagem.

Muitos projetos têm o seu próprio estilo de linguagem e para evitar conflitos, este guia específico tem prioridade para o presente projeto.

A Inconsistência 'preenche' as pequenas mentes

Um dos entendimentos chave de Guido, é que um código é lido muito mais vezes, do que escrito. O guia apresentado aqui tem a intenção de melhorar a legibilidade do código e torná-lo consistente através do amplo espectro do código Python. Como diz o PEP20, "legibilidade conta!".

Um guia de estilo para linguagem trata de consistência. Consistência no código de linguagem, consistência no projeto, e consistência em um módulo ou função é mais importante ainda!

Entretanto é importante saber quando ser inconsistente. Às vezes, algumas regras deste guia de linguagem não são aplicáveis. Quando estiver em dúvida, use o seu julgamento. Veja outros exemplos e decida qual parece ser o melhor. E não hesite em perguntar!

Em particular não descarte a compatibilidade aprendida anteriormente, só para cumprir este PEP.

Algumas outras razões para ignorar um guia em particular:

1. Quando o guia pode fazer o código menos legível, mesmo que alguém seja acostumado a ler este código, seguindo este PEP;
2. Ser consistente com um código comum, mas que também quebra, apesar disso ser uma oportunidade de limpar a bagunça do 'último programador';
3. Porque o código em questão é anterior a última versão e não há razão para modificá-lo;
4. Quando o código precisa se manter compatível com versões anteriores do Python que não suportam o recurso recomendado pelo guia.

O layout do código

Indentation (recuo)

Usar 4 espaços para cada nível de recuo.

As linhas devem alinhar os elementos agrupados verticalmente usando a linha implícita do Python, usando parênteses, colchetes ou chaves ou usando um recuo deslocado [1]. Com o uso do recuo deslocado, não deve haver um argumento na primeira linha e um recuo adicional deve ser usado para mostrar que é uma outra linha:

```
# Correto:

# Alinhado com um delimitador.
foo = oresumo_pep8_python(var_p, var_y,
                          var_t, var_h)

# Adicione 4 espaços (um nível extra de recuo) para diferenciar os argumentos
do resto.
def resumo_pep8_python(
          var_p, var_y, var_t,
          var_h):
    print(var_p)

# Os recuos deslocados devem adicionar um nível.
foo = resumo_pep8_python(
    var_p, var_y,
    var_t, var_h)
```

```
# Errado:

# Argumentos na primeira linha são proibidos quando não estão alinhados
verticalmente.
foo = resumo_pep8_python(var_p, var_y,
    var_t, var_h)

# Recuo adicional é preciso quando o recuo não é distinguível.
```

```
def resumo_pep8_python(
    var_p, var_y, var_t,
    var_h):
    print(var_p)
```

A regra dos 4 espaços é opcional, pois os recuos podem ter outros formatos assim:

```
foo = resumo_pep8_python(
  var_p, var_y,
  var_t, var_h)
```

Quando uma condição para um enunciado é longa o bastante para ser escrita em várias linhas, é importante notar que usar dois caracteres (por exemplo, if), mais 1 espaço, mais 1 parêntese, cria um recuo de 4 espaços para as linhas subsequentes desta condição e isso pode confundir com a instrução dentro da condição (if). Mas há diferentes formas de usar o recuo e assim, distinguir as linhas de condição das linhas da função dentro da condição (if):

```
# Sem recuo extra.
if (this_is_python and
    that_is_pep8):
    do_something()
```

```
# Adicionando recuo extra na linha de condição.
if (this_is_python
        and that_is_pep8):
    do_something()
```

Os parênteses, chaves e colchetes de fechamento da construção do enunciado podem estar alinhados com o primeiro caractere da linha acima, assim:

```
my_list = [
    1, 2, 3,
    4, 5, 6,
    ]
result = mostrar números pares(
    'números', 'pares'
    )
```

Ou podem estar alinhados com o primeiro caractere da linha de construção do enunciado, assim:

```
my_list = [
    1, 2, 3,
    4, 5, 6,
]
result = mostrar números pares(
    'números', 'pares'
)
```

Tabular ou dar Espaço?

Os espaços são os preferidos para a metodologia do recuo. E a tabulação deve ser usada para manter a consistência dos códigos que já tem tabulação. O Python não permite misturar tabulação e espaço para a metodologia do recuo.

O comprimento máximo da linha

Deve-se limitar todas as linhas ao máximo de 79 caracteres.

Para textos grandes como diretrizes ou comentários, deve-se limitar a 72 caracteres e estas limitações tornam a abertura da janela apta a abrir vários arquivos lado a lado e também quando se usa ferramentas de revisão de códigos que têm diferentes versões em colunas lado a lado.

Os limites são definidos para evitar a quebra automática da linha em alguns editores, até porque nem todas as ferramentas da web fornecem a quebra automática de linha (marcador na coluna final ao quebrar a linha) e isso torna mais difícil a leitura do código.

A linguagem do Python requer linhas limitadas a 79 caracteres e diretrizes e comentários com máximo de 72 caracteres.

No Python a melhor maneira de quebrar linhas é usando parênteses, chaves ou colchetes, quebrando a linha longa em várias linhas ao colocar as expressões entre parênteses, assim:

```
file_2.write(file_1.read())
```

Mas sempre certificar que o recuo da linha está correto!

Devo quebrar a linha antes ou depois de um operador binário?

Quebrar a linha após o operador binário pode dificultar a leitura do código, pois os operadores ficam espalhados nas diferentes colunas e o operador também fica distante do 'operando', assim:

```
# Wrong:
# operators sit far away from their operands
income = (gross_wages +
          taxable_interest +
          (dividends - qualified_dividends) -
          ira_deduction -
          student_loan_interest)
```

Então o matemático Donald Knuth criou a regra oposta, quebrando a linha antes do operador, assim:

```
# Correct:
# easy to match operators with operands
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
```

```
          - ira_deduction
          - student_loan_interest)
```

Na línguagem Python pode-se quebrar a linha antes ou depois do operador, mas na atual regra apresentada no PEP8, sugere-se o uso da regra de Knuth.

Linhas em Branco

As linhas em branco podem ser utilizadas, com moderação, para separar grupos de funções relacionadas.

O Python aceita o caracter usando as teclas Control e L, ou seja, ^L, como espaço em branco. Porém alguns editores e visualizadores de códigos na Web podem não reconhecer o Control-L e então observar se terá outro glifo no lugar.

Imports

Devem estar separados por linhas e são sempre colocados no topo do arquivo como um enunciado, logo depois do módulo com diretrizes ou comentários, mas antes do módulo com as funções.

```
# Correct:
import os
import sys

# Correct:
from subprocess import Popen, PIPE
```

Os arquivos devem ser importados, primeiro da biblioteca padrão global, depois da literatura relacionada e depois do próprio dispositivo local (por exemplo, computador).

E deve-se pôr uma linha em branco entre cada grupo de arquivo importado.

Para importar arquivos complexos, deve-se inserir o passo a passo, assim:

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

Aspas em uma String

Usar aspas simples ou aspas duplas em expressões no Python, é a mesma coisa. Melhor escolher as aspas simples, pois evita o uso de barra invertida e assim, melhora a legibilidade do código.

Espaço em branco em expressões e enunciados

Atritos

Evitar espaços em branco estranhos nas seguintes situações:

Em expressões logo após parênteses, chaves ou colchetes;

Entre um vírgula final e um parêntese de fechamento;

Imediatamente antes de vírgula, ponto e vírgula e dois pontos.

Entretanto dentro de um intervalo de dados, os dois pontos atuam como um operador binário e então deve ter quantidades iguais de dados em ambos os lados. E quando há um grande intervalo de dados, os dois pontos devem ter o mesmo número de espaços aplicados. Mas quando um intervalo de dados é omitido, o espaço também deve ser omitido. Exemplos:

```
# Correct:
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]
```

Ainda deve-se omitir o espaço em branco antes de:

Parêntese que começa uma lista de argumentos para uma função;

Parêntese que começa um intervalo de dados;

Mais de um espaço antes de um atributo para alinhar com outro atributo, assim:

```
# Correct:
x = 1
y = 2
long_variable = 3
```

Outras recomendações

Evite muitos espaços em branco, pois podem confundir e dificultar a legibilidade do código.

Sempre coloque o operador binário entre 1 único espaço de cada lado como em atributo (=), atributo mais outra função (+=, -=, etc), comparações (==, <, >, !=, <>, <=, >=, in, not in, is, is not), Booleanos (and, or, not).

Se são usados operadores com diferentes prioridades, sempre adicione espaço em volta do operador com menor prioridade. Nunca use mais de 1 espaço e certifique-se de ter a mesma quantidade de espaço dos dois lados do operador binário, assim:

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

Para as anotações de função deve-se considerar a regra padrão para dois pontos e sempre ter espaço entre a seta, quando houver ->, assim:

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

Não usar espaços ao redor de = quando este for um indicador de argumento ou palavra-chave ou quando for indicador de uma função não anunciada:

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

Ao combinar uma anotação de argumento deve-se usar o espaço nos dois lados de =, assim:

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

Deve-se evitar vários argumentos numa mesma linha:

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

E melhor não:

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

Pode-se usar if/for/while na mesma linha, mas nunca fazer isso quando tiver muitas funções com atributos múltiplos.

Melhor não:

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

E definitivamente não 'dobrar' linhas longas:

```
# Wrong:
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()
```

Quando usar vírgulas de fechamento

A vírgula de fechamento deve ser usada quando há uma tupla (sequência ordenada de elementos), então se recomenda usar um parêntese de fechamento após a vírgula de fechamento, mesmo sendo redundante, assim:

```
# Correct:
FILES = ('setup.cfg',)
```

Quando a vírgula de fechamento é redundante, ela ainda se torna útil no caso de uma versão do sistema de controle ou quando uma lista de valores ou argumentos importados podem mudar ao longo do tempo. O padrão é colocar cada valor em cada linha e fechar cada linha com uma vírgula de fechamento e ainda, adicionar um parêntese, chave ou colchete na próxima linha, assim:

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

Comentários

Sempre é melhor ter um comentário, mesmo que contradiga o código, do que não ter comentário. Mas sempre atualize seu comentário, quando o código for alterado!

Os comentários devem ser compostos por frases completas e estas, finalizadas com um ponto final. A primeira palavra deve ser em letra MAIÚSCULA, caso ela não seja um indicador, pois então, este deve iniciar com letra minúscula. E isso é padrão para os indicadores, sempre iniciar com letra minúscula!

Após o ponto final de cada frase nos comentários deve ter dois espaços. E somente um espaço quando for o ponto final da última frase do comentário com várias frases.

Os comentários devem ser claros e de fácil leitura para os usuários que falam outros idiomas, diferentes do seu.

Para programadores que não têm o inglês como língua materna, devem sempre escrever seus comentários em inglês!!!

Comentários em bloco

Os comentários em bloco são aplicados a algum código e eles têm o seu recuo alinhado com o recuo deste código.

Cada linha do bloco do comentário deve iniciar com um # e 1 único espaço após o #, assim:

```
# força
# eu estou acabando o resumo
```

Comentários embutidos

Estes devem ser usados com moderação, pois podem distrair do enunciado, quando dizem o óbvio, assim:

```
x = comer + salada                # comer mais salada
```

Comentários embutidos são aqueles que estão na mesma linha de um enunciado, ou seja, quando estão dentro de um enunciado. Neste caso, devem ser separados por ao menos, 2 espaços após o enunciado e devem começar com um # e 1 único espaço após o #, assim:

```
x = comer + salada                # comer mais salada verde
```

Escrevendo expressões

Escrever expressões para módulos, funções, classes e métodos públicos é importante. Mas essas expressões são desnecessárias quando não são públicas e mesmo assim, deve-se ter um comentário que descreve o que sua expressão para o enunciado faz. E este comentário deve aparecer depois da *def line.*

O PEP257 descreve diretrizes para boas expressões. Note que o mais importante, é que “”” que finaliza uma expressão em várias linhas deve estar sozinho, em uma linha assim:

```
"""Return a foobang

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

Mas para uma expressão em uma única linha, é só manter o “”” de fechamento, na mesma linha, assim:

```
"""Return an ex-parrot."""
```

Convenções da nomenclatura

Neste documento estão os padrões de nomenclatura atualmente recomendados. Novos pacotes devem seguir tais padrões e manter a consistência.

Princípio Primordial

É que os nomes de pacotes públicos devem refletir o seu uso e não a sua implementação, ou seja, que os códigos destes pacotes mostrem para que servem, e não porque devem ser implementados.

Descritivos: estilos de nomenclatura

Existem muitos estilos diferentes de nomenclatura. E isso ajuda para reconhecer qual estilo de nomenclatura está sendo usado, independentemente da sua função.

Os seguintes estilos de nomenclatura são diferenciados assim:

- b = letra minúscula;
- B = letra maiúscula;
- lowercase;
- lower_case_with_underscores;
- UPPERCASE;
- UPPER_CASE_WITH_UNDERSCORES;
- CapitalizedWords (CapWords ou o Caso do Camelo em função do formato acidentado das letras);

Porém aqui, ao usar siglas, todas as letras devem ser maiúsculas. Logo, SIGLAS (por exemplo, HTTPServerError e não, 'http').

- casoMisto que difere das CapWords em função da 1ª letra ser minúscula.

Em Python os nomes de atributos e de métodos são prefixados como um objeto, enquanto os nomes de funções são prefixados como um nome de módulo!!!!

Atenção para os formatos especiais que usam sublinhados iniciais ou finais, assim:

- `_single_leading_underscore`: Este é considerado fraco, pois somente diz assim: from M import *, ou seja, não importe objetos que tem seus nomes com a letra inicial minúscula;

- `single_trailing_underscore_`: Este é usado por convenção para evitar conflitos com as palavras-chave do Python, assim:

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

- `__double_leading_underscore`: Ao nomear uma classe de atributos, os nomes dos atributos ficam assim: na classe de atributos LudMilla, __mud será assim: _LudMilla__mud, representando a regra de manipulação dos nomes no Python.
- `__double_leading_and_trailing_underscore__`: usado quando os atributos são controlados pelo usuário, por exemplo: __init__, __import__ ou __file__. Não invente nomes, use os que já estão documentados na língua Python!

Prescritivos: convenções de nomenclatura

Nomes para evitar

Nunca use caracteres minúsculos para 'i' ou 'o' e nem caractere maiúsculo para 'I' como nomes de variáveis com caractere único. Em alguns casos, estes caracteres podem ser confundidos com o número 1 e o número 0. Assim, ao invés de usar 'l', use 'L'.

Nomes de pacotes e módulos

Os nomes devem ser curtos e todos com letras minúsculas. E se for para melhorar a legibilidade do código, sublinhados podem ser usados nestes nomes.

Nomes de classes

Estes nomes normalmente devem usar a norma da CapWords.

Há uma convenção específica para nomes de classes que já existem no Python. Assim, estes nomes são representados por 1 ou 2 palavras que são executadas juntas. Assim, a norma da CapWords é usada somente em algumas exceções e para constantes que já existem.

Formatos de nomes de variáveis

Os nomes de variáveis, de acordo com o PEP484 devem ser curtos e usar o CapWords e ainda usar sufixos, como _co ou _contra para as variáveis usadas para apontar tal comportamento correspondente, assim:

```
from typing import TypeVar

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

Nomes de variáveis globais

Suas convenções são, praticamente, as mesmas das funções.

Mas convém usar a versão mais antiga de prefixar tais globais com um sublinhado e assim, indicar que essas globais são de um 'módulo não público'.

Nomes de funções e nomes de variáveis

Os nomes das funções devem estar com letras minúsculas e as palavras separadas por sublinhados para facilitar a leitura do código. Os nomes das variáveis seguem este mesmo padrão.

A norma de mixedCase é permitida apenas quando já é o estilo predominante do código para manter a compatibilidade com as versões anteriores.

Argumentos de função e argumentos de método

Sempre use self como primeiro argumento dos métodos de função. E sempre use cls para argumento dos métodos de classe.

Se o nome de um argumento de função entrar em conflito com uma palavra-chave reservada no Python, é melhor adicionar um único sublinhado final ao invés de abreviar. Assim, class_ é melhor que clss. Melhor ainda, usar um sinônimo para evitar tal conflito, como por exemplo, grup_.

Nomes de métodos e variáveis de atributos

Usar a norma da nomenclatura de função com letras minúsculas e palavras separadas por sublinhados para melhorar a legibilidade.

Use um sublinhado inicial somente para métodos não públicos e atributos.

Mas para evitar conflitos de nomes com subclasses, é necessário usar dois sublinhados iniciais e o Python fará a regra de manipulação dos nomes. Mas geralmente, o suo de dois sublinhados iniciais devem ser usados somente para conflitos com atributos em classes designadas para ser subclasse.

Constantes

São geralmente definidas em um nível de módulo e escritas com letras maiúsculas e sublinhados separando as palavras, assim: MAX_OVERFLOW e TOTAL.

Projetando para o futuro

Sempre decida se os métodos e atributos de uma classe serão públicos ou não. Em dúvida, faça 'não-público', pois é mais fácil torná-lo público do que o oposto, mais tarde, se necessário.

Atributos públicos são aqueles que serão usados por qualquer pessoa, enquanto que os 'não-públicos' são aqueles que não serão usados por grupos diferentes do seu campo de atuação. Mas não há garantias que os atributos não-públicos não serão alterados ou até removidos.

Existem os atributos 'protegidos'. Neste caso, algumas classes foram projetadas para serem herdadas, seja para estender ou para modificar o comportamento da classe. Neste caso, é importante deixar claro quais os atributos são públicos, não-públicos e protegidos.

Assim, seguem as Pythonic guidelines:

- Atributos públicos não devem ter sublinhados iniciais no código;
- Se o nome do seu atributo público conflita com uma palavra-chave reservada, adicione um único sublinhado ao seu atributo público, o que é melhor que abreviar (observe a norma em nomear argumento para método de classes, explicada acima);
- Para atributos simples de dados públicos, é melhor ter somente o nome do atributo. O Python é simples e fornece um caminho fácil para melhorias futuras, caso seja necessário aumentar o comportamento da função e para isso, é necessário usar expressões simples de acesso a atributos de dados (evite usar códigos para computação complexa, pois a notação do atributo faz o usuário achar que o código demanda operações simples);
- Se a sua classe for designada para ser uma subclasse, mas seus atributos são de classe e não de subclasse, nomeie seus atributos com duplo sublinhado no início e nenhum sublinhado no final. Isso solicita a manipulação de nomes pelo Python e aí, o nome da classe é transformado em nome de atributo. Assim, evitando o conflito de nomes de atributo, caso as subclasses tenham atributos com mesmo nome (ainda, a

manipulação de nomes pode tornar o uso menos conveniente, mas o algoritmo de manipulação de nomes está bem documentado e é fácil de executar manualmente).

Interface Pública e Interface Interna

Somente a interface pública tem garantia de compatibilidade com versões anteriores.

A interface documentada é considerada pública, a menos que o código diga explicitamente, que é uma interface provisória ou interna e está isenta de garantir compatibilidade com versões anteriores. Todas as interfaces não documentadas devem ser consideradas internas.

Os módulos devem declarar claramente que seus nomes são públicos, usando o atributo __all__. Se este atributo for definido como uma lista vazia, o módulo não é considerado público.

Uma interface é considerada interna se qualquer pacote, módulo ou classe for considerado interno.

Recomendações de programação

O código deve ser escrito de forma que não prejudique outras implementações de Python (PyPy, Jython, IronPython, Cython, Psyco, etc.) Para concatenação de strings na forma a += b ou a = a + b, é melhor usar o .join(), pois a concatenação poderá ocorrer em tempo linear em várias implementações, como em casos de recontagem.

Comparações com expressões únicas, como None devem ser feitas com is ou is not e nunca com operadores de igualdade.

Tome cuidado ao escrever if x quando na verdade quer dizer if x is not None e este None é definido com algum outro valor. Neste caso, este valor pode ter um tipo que pode ser falso em um contexto booleano.

Usar is not é melhor que not... is, pois a leitura é mais fácil, assim:

```
# Correct:
if foo is not None:
```

O PEP207 indica que as regras de reflexivity são assumidas pelo Python. Assim, é melhor implementar as seis operações <; >; <=; >=; ==; != , mesmo que as funções sort() e min() usem o operador < e a função max() usa o operador >, pois tais funções podem gerar confusão em um contexto diferente.

Procure responder à pergunta “O que deu errado?” de forma programática, ao invés de apenar declarar que “Ocorreu um erro” (consulte PEP3151).

As convenções de nomenclatura de classe se aplicam, mas é necessário adicionar o sufixo “Erro” às classes de exceção se a exceção for um erro.

Ao capturar exceções, mencione exceções específicas sempre que possível, ao invés de usar o termo except, assim:

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

Usar somente o termo except irá capturar exceções de SystemExit e KeyboardInterrupt, o que pode tornar difícil interromper o programa com Control-C e ainda pode disfarçar outros problemas. Se você deseja capturar todas as exceções que indicam erros de programa, use o código except Exception:

Uma boa regra é limitar o uso do termo except para dois casos:

1. Se o operador de exceção estiver imprimindo ou rastreando o trajeto, o usuário saberá que houve um erro;
2. Se o código precisa fazer algum trabalho de limpeza, deixe a exceção se propagar com o termo raise. try...finally, pois pode ser a melhor maneira de lidar com este caso.

Ao capturar erros do sistema operacional, prefira a hierarquia de exceções explícita introduzida no Python 3.3 ao invés de introspecção ‘errno values’.

Ainda, para todos os termos try/except, limite o termo try a um mínimo de quantidade de código necessário e isso também evitará mascaras erros, assim:

```
# Correct:
try:
    value = collection[dogs]
except DogsError:
    return Dogs_not_found(dogs)
else:
    return handle_value(value)
```

Quando um recurso é local para uma seção específica do código, use a instrução with paraq garantir que ele seja limpo de forma rápida e confiável após o uso. Usar a instrução try/finally também é aceitável.

Os operadores de contexto devem ser solicitados por meio de funções ou métodos separados sempre que fizerem algo diferente de adquirir ou liberar recursos:

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

```
# Wrong:
with conn:
    do_stuff_in_transaction(conn)
```

O último exemplo está errado, pois não fornece nenhuma informação que os métodos de entrada __enter__ e de saída __exit__ estão fazendo algo diferente de fechar a conexão após uma transação. Ser explícito neste caso é importante!

Seja consistente nas instruções de Retorno. Todas as instruções de retorno em uma função devem retornar uma expressão ou nenhuma delas deveria. Assim, se uma instrução de Retorno, retorna uma expressão, toda instrução de Retorno onde nenhum valor é retornado, isso deve retornar explicitamente, como return None e uma explícita instrução de retorno deve estar presente no final da função assim:

```
# Correct:

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)
```

Use ‘.starwith()’ e ‘.endwith()’ ao invés de fatiar expressões para verificar prefixos ou sufixos. Estes termos são mais limpos e menos propensos a erros:

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

As comparações de tipos de objetos devem sempre usar o código isinstance() ao invés de comparar os tipos diretamente, assim:

```
# Correct:
if isinstance(obj, int):
# Wrong:
if type(obj) is type(1):
```

Para sequências (expressões, listas e tuplas), considere o fato de que sequências vazias são falsas:

```
# Correct:
if not seq:
if seq:
```

Não escreva expressões que dependam de espaços em branco significativos. Estes espaços em branco finais são indistinguíveis e alguns editores podem cortá-los.

Não compare valores booleanos com True ou False, usando ==

```
# Correct:
if greeting:
# Wrong:
if greeting == True:
```

E pior ainda:

```
# Wrong:
if greeting is True:
```

O uso das instruções de controle de fluxo como return/break/continue dentro de um pacote final de um código try...finally, onde o controle de fluxo sairia deste pacote final, é desencorajado. Isso ocorre porque tais instruções iriam implicitamente, cancelar qualquer exceção ativa que esteja se propagando através do pacote final:

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

Anotações de função

Com o PEP484, as regras do estilo para anotações de função têm mudado.

- Anotações de função devem usar a combinação do PEP484;
- A experimentação com estilos de anotação recomendada anteriormente neste PEP não é mais incentivada;
- Mas são incentivados experimentos dentro das regras do PEP484. Por exemplo, marcar uma grande biblioteca ou aplicativo de terceiros com anotações do tipo PEP484, revisar como foi fácil adicionar estas anotações e observar se elas ajudam na compreensão do código;
- A biblioteca padrão do Python deve ser conservadora na adoção de tais anotações, mas seu uso é permitido para novos códigos e para grandes repetições;
- Para código que deseja fazer um uso diferente das anotações de função é recomendado colocar um comentário no formato:

```
# type: ignore
```

No topo do arquivo, o que diz aos verificadores de tipo para ignorar todas as anotações.

- Os verificadores de formato são ferramentas opcionais e separadas. Os usuários Python por padrão, não devem omitir nenhuma mensagem devido à verificação de formato e não devem alterar seu comportamento, baseados em anotações;
- Aqueles que não desejam usar verificadores de formato podem ignorá-los. Porém, espera-se que os usuários de pacotes de bibliotecas de terceiros queiram executar os verificadores de formato nesses pacotes. Assim, o PEP484 recomenda o uso de arquivos específicos (arquivos stub), como arquivos .py. Estes arquivos podem ser distribuídos com uma biblioteca ou separadamente, com a permissão do autor, por meio de repositório digitado.

Anotações de variáveis

O PEP526 introduziu anotações de variáveis. E as recomendações de estilo são semelhantes às anotações de função descritas acima.

- As anotações para variáveis de nível de módulo, variáveis de classe e de instrução, e variáveis locais devem ter um único espaço após os dois pontos;
- Não deve haver espaço antes dos dois pontos;
- Se uma atribuição tiver o lado direito, o sinal de igualdade deverá ter exatamente um espaço entre ambos os lados:

```
# Correct:

code: int

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

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

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

Embora o PEP526 seja aceito para Python 3.6, o arranjo de anotação de variável é o arranjo preferido para arquivos stub em todas as versões do Python (consulte PEP484 para mais detalhes).

Notas de rodapé

[1]

O recuo deslocado é um estilo de configuração do tipo em que todas as linhas de um parágrafo são recuadas, exceto a primeira linha. No contexto do Python, o termo é usado para descrever um estilo em que o parêntese de abertura de uma instrução entre parênteses é o último caractere sem espaço em branco da linha, com as linhas subsequentes sendo recuadas até o parêntese de fechamento.

Referências

[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

Source: https://github.com/python/peps/blob/main/peps/pep-0008.rst

Last modified: 2023-12-09 16:19:37 GMT