PESSOAS 2030 PORTUGAL 2030 Cofinanciado pela União Europeia
Os Fundos Europeus mais próximos de si.

GuiaDeEstilo

PEP 8 - Guia de Estilo Para Python

Este artigo é uma tradução livre da PEP 8 - Style Guide for Python Code, de autoria de GuidoVanRossum (<guido at python.org>, referenciado também como GvR) e Barry Warsaw (<barry at python.org>), disponível em http://www.python.org/peps/pep-0008.html.

Introdução

Este documento oferece convenções para o código Python compreendido pela biblioteca padrão que acompanha a distribuição. Há outro documento semelhante [1] que trata do estilo para o código em C usado na implementação do interpretador e nas extensões que compõe a biblioteca padrão.

O conteúdo deste documento é uma adaptação do artigo original *Python Style Guide, de GuidoVanRossum*, com alguns acréscimos retirados do guia de estilo de Barry Warsaw. Onde houve conflitos, as regras de GvR foram mantidas.

Uma Consistência Tosca é o Bicho-Papão das Pequenas Mentes

O propósito de um guia de estilo é manter a consistência. Consistência em relação a esse guia é importante. Consistência em relação a outros detalhes de um projeto é mais importante. Consistência em relação a um módulo ou função é ainda mais importante.

Mais importante ainda: saiba quando ser inconsistente - algumas vezes, as regras deste guia simplesmente não se aplicam. Em caso de dúvida, use seu melhor julgamento. veja outros exemplos e decida o que fica melhor. E não hesite em perguntar!

Duas boas razões para quebrar uma regra:

- Quando a adoção de uma regra irá tornar o código menos legível, mesmo para alguém acostumado com essas regras.
- Quando deseja-se ser consistente com outro código que acompanhe aquele em desenvolvimento que também viola as regras - apesar dessa ser uma boa oportunidade para consertar a bagunça de alguém.

Formatação do Código

- Indentação
    - Use o padrão usado pelo "Python-mode" do Emacs: 4 espaços por nível de indentação. Para código realmente antigo que você não quer bagunçar, você pode continuar a usar 8 espaços. O Python-mode automaticamente detecta o nível de indentação predominante em um arquivo e segue este padrão.
- Tabulações ou espaços

    - Nunca misture tabulações e espaços. A forma mais popular de indentar código em Python é somente com espaços. A segunda forma mais popular é somente com tabulações. Código com uma mistura dos dois deve ser convertido para usar somente espaços. (No Emacs, selecione o buffer inteiro e digite ESC-x untabify). Passando a opção -t para o interpretador Python, faz com que ele emita avisos sobre código que misture ilegalmente tabulações e espaços. Com -tt esses avisos se tornam erros. Essas opções são altamente recomendadas! Para projetos novos, recomenda-se usar somente espaços. Muitos editores têm opções para tornar isso mais fácil.
- Comprimento máximo de linhas
    - Ainda há por aí muitos monitores limitados a linhas de 80 colunas (além disso, limitando as janelas a 80 caracteres, permite ter várias janelas abertas, lado a lado). As quebras de linha padrão nesses monitores é horrível, portante, limite todas as linhas a um máximo de 79 caracteres. (o Emacs quebra as linhas que têm exatamente 80 caracteres). Para longos blocos de texto (*docstrings* ou comentários), limitar o comprimento a 72 colunas é recomendado.

      A melhor maneira de continuar linhas longas é usando a continuação implícita, entre parênteses, colchetes e chaves. Se necessário, você pode adicionar um par extra de parênteses em volta de uma expressão, mas, às vezes, uma barra invertida fica melhor. Tome o cuidado de indentar a linha adequadamente. O *Python-mode* do Emacs faz isso automaticamente. Exemplo:

Esconder número das linhas

```
  1     class Rectangle(Blob):
  2
  3         def __init__(self, width, height,
  4                      color='black', emphasis=None, highlight=0):
  5             if width == 0 and height == 0 and \
  6                color == 'red' and emphasis == 'strong' or \
  7                highlight > 100:
  8                 raise ValueError, "sorry, you lose"
  9             if width == 0 and height == 0 and (color == 'red' or
 10                                                emphasis is None):
 11                 raise ValueError, "I don't think so"
 12             Blob.__init__(self, width, height,
 13                           color, emphasis, highlight)
```

- Linhas em branco
    - Separe funções e definições de classe com duas linhas em branco. Métodos dentro de uma classe devem ser separados com uma única linha em branco. Linhas extras podem ser usadas (esporadicamente) para separar grupos de funções relacionadas e podem ser omitidas entre grupos relacionados de linhas, como por exemplo, métodos que sejam sobrescritas em subclasses. Quando linhas em branco são usadas para separar métodos, deve haver também uma linha em branco entre a linha 'class' e o primeiro método da classe. Use linhas em branco para separar blocos lógicos dentro de métodos e funções. Python aceita o caractere control-L (^L) como espaço em branco; o Emacs (e algumas ferramentas de impressão) tratam esses caracteres como quebra de página, então você pode usá-los para separar páginas de seções

relacionadas no seu arquivo.

- Import
    - Imports devem sempre ser feitos em linhas separadas, por exemplo:

incorreto:

Esconder número das linhas

```
1 import sys, os
```

correto:

Esconder número das linhas

```
1 import sys
2 import os
```

Mas não há problema algum em usar:

Esconder número das linhas

```
1 from types import StringType, ListType
```

- Imports devem ser sempre colocados no topo do arquivo, logo depois de quaisquer comentários ou *docstrings*, e antes de constantes ou globais. Eles devem ser agrupados seguindo a ordem:
    - módulos da biblioteca padrão
    - módulos grandes relacionados entre si (por exemplo, todos os módulos de *e-mail* usados pela aplicação)
    - módulos específicos da aplicação

  Você deve colocar uma linha em branco separando cada grupo de módulos.
- Quando importar uma classe de um módulo de mesmo nome, não há problemas em usar:

Esconder número das linhas

```
1 from MyClass import MyClass
2 from foo.bar.YourClass import YourClass
```

- No entanto, se isso causar conflitos com nomes, use:

Esconder número das linhas

```
1 import MyClass
2 import foo.bar.YourClass
```

- e depois "`MyClass.MyClass`" e "`foo.bar.YourClass.YourClass`".
- Espaços em expressões e instruções
    - Guido odeia espaços nos seguintes lugares:
    - Imediatamente antes e após parêntese, colchete ou chave, como em:

Esconder número das linhas

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

Sempre escreva assim:

Esconder número das linhas

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

- Logo antes de uma vírgula, ponto-e-vírgula ou dois-pontos:

Esconder número das linhas

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

Sempre escreva assim:

Esconder número das linhas

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

- Logo antes do parêntese que abre a lista de argumentos de uma função, como em:

Esconder número das linhas

```
spam (1)
```

Sempre escreva:

Esconder número das linhas

```
spam(1)
```

- Imediatamente antes da chave que abre um índice, como em:

Esconder número das linhas

```
dict ['key'] = list [index]
```

Sempre escreva:

Esconder número das linhas

```
dict['key'] = list[index]
```

- Mais do que um espaço em volta de algum operador, para alinhar os operandos:

Esconder número das linhas

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

Escreva:

Esconder número das linhas

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

- Outras recomendações
    - Sempre circunde os seguintes operadores binários com um único espaço de cada lado: =, ==, <, >, !=, <>, <=, >=, in, not in, is, and, or, not
    - Use o seu julgamento na hora de inserir espaços entre operadores aritméticos. Exemplos:

Esconder número das linhas

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

- Não use espaços ao redor do sinal de igual (=) quando usado para indicar um valor padrão de um argumento. Faça assim, por exemplo:

Esconder número das linhas

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

- Múltiplos comandos na mesma linha são desencorajados:

Não faça:

Esconder número das linhas

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

E sim:

Esconder número das linhas

```
1 if foo == 'blah':
2     do_blah_thing()
3
4 do_one()
5 do_two()
6 do_three()
```

Comentários

Comentários que contradizem o código são piores do que nenhum comentário. Sempre tenha como prioridade manter os comentários atualizados com as mudanças no código! Comentários devem sempre ser frases completas e sua primeira letra deve ser maiúscula, a menos que ele comece com um identificador que começa com uma letra minúcula.

Se um comentário for curto, o ponto final deve ser omitido. Comentários grandes normalmente consistem de um ou mais parágrafos e sentenças completas, estas sim devem terminar com ponto.

Você deve usar dois espaços depois do ponto final de uma frase, permitindo que o Emacs ajuste a linha de maneira consistente.

Programadores de países que não têm o inglês como língua nativa: escrevam seus comentários em inglês, a menos que você tenha 120% de certeza de que o código jamais será lido por pessoas que não falam sua língua.

Comentários em bloco devem ser indentados no mesmo nível do código a que se referem. Cada linha deve começar com # e um espaço (a menos que o texto dentro do comentário seja indentado). Parágrafos dentro de um bloco devem ser separados por uma linha contendo uma única tralha #. O bloco inteiro deve ser separado por uma linha em branco no topo e embaixo.

Comentários na mesma linha devem ser usados esporadicamente. Devem ser separados do comando por pelo menos dois espaços. Como outros comentários, devem começar com uma tralha e um espaço. Não faça comentários em coisas óbvias. Eles distraem mais do que ajudam.

Docstrings

Escreva *docstrings* para todo módulo, função, classe e método público. Elas não são necessárias para métodos "privados", mas é recomendável ter um comentário que explique o que ele faz. Este comentário deve estar logo após a declaração.

A PEP 257 descreve as convenções usadas para *docstrings*. As mais importantes a lembrar são que deve sempre usar aspas triplas (string multiline) mesmo que a *dosctring* ocupe apenas uma linha (facilita uma possível expansão posterior) e que as aspas triplas que finalizam uma dosctring em múltiplas linhas deve estar em uma linha separada.

Esconder número das linhas

```
1 """Return a foobang
2
3 Optional plotz says to frobnicate the bizbaz first.
4 """
```

Controle de Versão

Se você utiliza um cabeçalho para RCS ou CVS nos seus arquivos de código, faça como da seguinte forma:

Esconder número das linhas

```
1 __version__ = "$Revision: 1.20 $"
2 # $Source: /cvsroot/python/python/nondist/peps/pep-0008.txt,v $
```

Estas linhas devem ser incluídas logo após as *docstrings* do módulo, antes de qualquer código, separadas por uma linha em branco acima e abaixo.

Nomes e Identificadores

As convenções usadas em nomes na biblioteca padrão são um pouco bagunçadas e díficilmente vamos conseguir torná-las consistentes. Mesmo assim, vamos a algumas regras.

- Estilos de nomes
    - Há uma série de diferentes estilos usados para identificadores. É bom saber

reconhecer qual estilo está sendo usado, independentemente do que está sendo feito. Os estilos mais comuns são:

- "minúsculas_separadas_com_underscore"
- "MAIÚSCULAS_SEPARADAS_COM_UNDERSCORE"
- "PalavrasComeçandoPorMaíusculas"
- "nome!ComeçandoPorMinúscula"
- "Palavras_Começando_Por_Maiúsculas_E_Underscore" (horrível!)

Há ainda o costume de usar um prefixo curto para agrupar nomes relacionados. Por exemplo, a função `os.stat()` retorna uma tupla cujos ítens têm nomes como st_mode, st_size, st_mtime e assim port diante. A biblioteca X11 usa um X como prefixo para todas suas funções públicas. Este estilo não é muito comum em Python, porque, geralmente, atributos e nomes de métodos já são prefixados por um objeto, e funções, por um módulo.

Adicionalmente, as seguintes formas de usar *underscores* antes ou depois do identificador são reconhecidas.

- `_underscore_no_inicio`: costuma indicar que o atributo é de uso interno. ("from M import *" não importa objetos cujos nomes comecem com _)
- `underscore_no_fim_`: usado para evitar conflitos com palavras-chave.por exemplo: "Tkinter.Toplevel(master, class_='ClassName')".
- `__dois_underscores_no_início`: atributo privado da classe (`classe.__atributo` é convertido para `classe.__classe__atributo`).
- `__dois_underscores_no_início_e_no_fim__`: atributos ou objetos especiais, como `__init__` , `__import__` ou `__file__` . As vezes estes podem ser definidos pelo usuário para disparar alguma ação especial (sobrecarga de operadores, por exemplo).

- Convenções para Nomes:
  - Nomes a evitar Nunca use os caracteres 'l' (L minúsculo), 'O' (o maiúsculo) ou 'I' (i maiúsculo) sozinhos como nomes de variáveis. Em algumas fontes, esses caracteres são indistinguíveis dos números um e zero. Quando tentado a usar somente 'l', use 'L'.
  - Nomes de Módulos

    Módulos devem ter nomes ou em PalavrasComeçandoPorMaíusculas ou totalmente_em_minúsculas. Módulos que contenham uma única classe podem ter o mesmo nome da classe (como no módulo StringIO, por exemplo). Módulos que exportam apenas funções são normalmente nomeados em minúsculas. Como nomes de módulos são mapeados para nomes de arquivos, e alguns sistemas de arquivo não apenas desprezam maiúsculas e minúsculas como também reduzem o comprimento do nome, é importante que eles sejam escolhidos de forma a serem curtos e não entrar em conflito com outros módulos. Isso não é um problema em sistemas Unix ou

Linux, mas pode ser um problema se o código for usado em Mac ou Windows.

Há uma convenção surgindo de que quando uma extensão escrita em C ou C++ tem um módulo em Python que ofereça uma interface de alto nível, este módulo deve ter o nome em PalavrasComeçandoPorMaiúsculas, enquanto o módulo em C/C++ deve ter o nome todo em minúsculas, começando por um underscore (_socket, por exemplo).

- Nomes de Classes

  Quase sem exceção, nomes de classe devem usar o padrão de PalavrasComeçandoPorMaiúscula, exceto no caso de classes para uso interno, que devem começar com um *underscore*.

- Nomes de Exceptions

  Se um módulo define uma única *exception* usada para todos os tipos de erro, ela é geralmente chamada "error" ou "Error".

- Nomes de Funções

  Funções globais, exportadas por um módulo podem usar tanto o padrão PalavrasComeçandoPorMaiúscula, quanto totalmente em minúsculas ou minúsculas_separadas_por_underscore). Não há nenhuma preferência clara, mas o primeiro estilo costuma ser mais usado para funções que provêm mais funcionalidade, enquanto o segundo é usado por funções mais simples.

- Variáveis Globais Variáveis globais devem ser usadas somente dentro do módulo. As convenções são as mesmas para funções. Módulos que são projetados para ser usados com 'from M import *' devem ter suas globais com um underscore como prefixo, para evitar que sejam exportadas.
- Nomes de Métodos

  Vale o mesmo para as funções. Use dois *underscores* quando for importante que apenas a classe atual acesse um atributo. (mas tenha em mente que isso não torna o método realmente privado. Um usuário insistente ainda pode acessá-lo de diversas formas, através do atributo `__dict__` por exemplo).

- Herança
  - Decida sempre se os métodos de uma classe e as variáveis de uma instância serão públicos ou não. Em geral, nunca torne variáveis públicas, a menos que você esteja implementando algum tipo de registro. Decida ainda se os atributos serão privados ou não. A diferença entre eles é que atributos privados são aqueles que jamais terão utilidade para uma subclasse, enquanto os públicos podem ter. É prudente projetar suas classes tendo a possibilidade de herança em mente.

Atributos privados devem ter dois *underscores* no começo e nenhum no fim. Atributos não-públicos devem ter um *underscore* no começo e nenhum no fim.

Atributos públicos não devem ter *underscores* nem no começo, nem no fim, a menos que eles

entrem em conflito com palavras reservadas, caso em que um único *underscore* no fim é preferível a um no começo, ou a uma pronúncia diferente, como class_ ao invés de klass.

Recomendações ao Programar

Comparações com *singletons*, como `None`, `False` e `True` devem sempre ser feitas com "is" ou "is not". Além disso, cuidado para não escrever "if x" quando o que você deseja é "if x is not None", como ao testar se uma variável ou argumento que tem um valor padrão de `None`, teve outro valor atribuído.

Classes são sempre preferidas à *strings*, como em *exceptions*. Módulos ou pacotes devem definir sua própria classe-exception base, que deve ser uma subclasse da classe Exception. Sempre inclua uma *docstring*. Exemplo:

Esconder número das linhas

```
1 class MessageError(Exception):
2     """Base class for errors in the email package."""
```

Use métodos do objeto *string* ao invés do módulo *string*, a menos que seja exigida compatibilidade com versões de Python anteriores à 2.0. Os métodos são muito mais rápidos e têm a mesma API de strings *Unicode*.

Evite fatiar *strings* quando verificando prefixos ou sufixos. Use os métodos `startswith()` e `endswith()`, que são mais eficientes e menos sujeitos a erro. Por exemplo:

Não use:

Esconder número das linhas

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

E sim:

Esconder número das linhas

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

A exceção é se existir a necessidade do seu código de funcionar com versões de Python anteriores à 1.5.2.

Comparações de tipo de objetos devem sempre usar `isinstance()` ao invés de comparar tipos diretamente. Exemplo:

Não use:

Esconder número das linhas

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

E sim:

Esconder número das linhas

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

Quando estiver verificando se o objeto é uma string, lembre-se que ele também pode ser uma *string unicode*! Em Python 2.3, *str* e *unicode* têm uma classe base em comum, *basestring*, então você pode simplesmente escrever:

Esconder número das linhas

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

Em Python 2.2 o módulo *types* tem o tipo StringTypes para o mesmo propósito:

Esconder número das linhas

```
1 from types import StringTypes
2 if isinstance(obj, StringTypes):
```

Com seqüências (strings, listas, tuples), tenha em mente o fato de que, quando vazias, elas são falsas em um contexto booleano, portanto, "if not seq" ou "if seq" são preferíveis a "if len(seq)" ou "if not len(seq)".

Não use *strings* que dependam de uma quantia significativa de espaços no começo ou no fim. A quantidade desses espaços é visualmente indistinguível e alguns editores até mesmo a ajustam.

Não compare valores booleanos com `True` e `False` usando == (o tipo Bool é novo em Python 2.3)

Não use

Esconder número das linhas

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

E sim

Esconder número das linhas

```
1 if greeting:
```

Referências

1. PEP 7, Style Guide for C Code, van Rossum
2. http://www.python.org/doc/essays/styleguide.html
3. PEP 257, Docstring Conventions, Goodger, van Rossum
4. http://www.wikipedia.com/wiki/CamelCase
5. Barry's GNU Mailman/mimelib style guide

Copyright

Traduzido por PedroWerneck

1. PEP 7, Style Guide for C Code, van Rossum (1)