Guía de estilo para el código Python – PEP 8 en Español

www.recursospython.com

Introducción

Este documento brinda las convenciones de escritura de código Python abarcando la librería estándar en la principal distribución de Python. Vea el documento PEP que describe las pautas para el código C en la implementación escrita en C de Python.

Este documento y el PEP 257 (convenciones para la documentación del código) fueron adaptados del originario texto de convenciones para el código Python, por Guido van Rossum, con algunas adiciones de la guía de Barry Warsaw.

Una consistencia estúpida es el "duende" de las mentes pequeñas

Una de las ideas claves de Guido es que el código es leído muchas más veces de lo que es escrito. Las pautas que se proveen en este documento tienen como objetivo mejorar la legibilidad del código y hacerlo consistente a través de su amplio espectro en la comunidad Python. Tal como el PEP 20 dice, "La legibilidad cuenta".

Una guía de estilo se trata de consistencia. La consistencia con esta guía es importante. La consistencia dentro de un proyecto es más importante. La consistencia dentro de un módulo o función es aún más importante.

Pero lo más importante: saber cuando ser inconsistente – simplemente en ciertas ocasiones la guía de estilo no se aplica. Cuando estés en duda, utiliza tu mejor criterio. Observa otros ejemplos y decide cual se ve mejor. ¡Y no dudes en preguntar!

Dos buenas razones para romper una regla en particular:

1. Cuando el aplicar la regla haga el código menos legible o confuso, incluso para alguien que está acostumbrado a leer códigos que se rigen bajo las indicaciones de este documento.
2. Para ser consistente en código que también la rompe (tal vez

por razones históricas) – aunque esto podría ser una oportunidad para limpiar el “desastre” de otra persona.

Diseño del código

Usa 4 (cuatro) espacios por indentación.

Para código realmente antiguo que no quieras estropear, puedes continuar usando indentaciones de 8 (ocho) espacios.

Las líneas de continuación deben alinearse verticalmente con el carácter que se ha utilizado (paréntesis, llaves, corchetes) o haciendo uso de la “hanging indent” (*aplicar tabulaciones en todas las líneas con excepción de la primera*). Al utilizar este último método, no debe haber argumentos en la primera línea, y más tabulación debe utilizarse para que la actual se entienda como una (línea) de continuación.

Sí:

```
# Alineado con el paréntesis que abre la función
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

# Más indentación para distinguirla del resto de las líneas
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)
```

No:

```
# Argumentos en la primera línea cuando no se está haciendo
uso de la alineación vertical
foo = long_function_name(var_one, var_two,
    var_three, var_four)

# La línea de continuación no se distingue del contenido de
la función
def long_function_name(
    var_one, var_two, var_three,
```

```
    var_four):
    print(var_one)
```

Opcional:

```
# No es necesaria la indentación extra
foo = long_function_name(
  var_one, var_two,
  var_three, var_four)
```

El paréntesis / corchete / llave que cierre una asignación debe estar alineado con el primer carácter que no sea un espacio en blanco:

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

O puede ser alineado con el carácter inicial de la primera línea:

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

¿Tabulaciones o espacios?

Nunca mezcles tabulaciones y espacios.

El método de indentación más popular en Python es con espacios. El segundo más popular es con tabulaciones, sin mezclar unos con otros. Cualquier código indentado con una mezcla de espacios y tabulaciones debe ser convertido a espacios exclusivamente. Al iniciar

la línea de comandos del intérprete con la opción “`-t`”, informa en modo de advertencias si se está utilizando un código que mezcla tabulaciones y espacios. Al utilizar la opción “`-tt`”, estas advertencias se vuelven errores. ¡Estas opciones son altamente recomendadas!

Para proyectos nuevos, únicamente espacios es preferible y recomendado antes que tabulaciones. La mayoría de los editores presentan características para realizar esta tarea de manera sencilla.

Máxima longitud de las líneas

Limita todas las líneas a un máximo de 79 caracteres.

Todavía hay varios dispositivos que limitan las líneas a 80 caracteres; más, limitando las ventanas a 80 caracteres hace posible tener varias ventanas de lado a lado. El estilo en estos dispositivos corrompe la estructura o aspecto visual del código, haciéndolo más dificultoso para comprenderlo. Por lo tanto, limita todas las líneas a 79 caracteres. En el caso de largos bloques de texto (“docstrings” o comentarios), limitarlos a 72 caracteres es recomendado.

El preferido método para “cortar” líneas largas es utilizando la continuación implícita dentro de paréntesis, corchetes o llaves. Además, éstas pueden ser divididas en múltiples líneas envolviéndolas en paréntesis. Esto debe ser aplicado en preferencia a usar la barra invertida (“\”).

La barra invertida aún puede ser apropiada en diversas ocasiones. Por ejemplo, largas, múltiples sentencias with no pueden utilizar continuación implícita, por lo que dicho carácter es aceptable:

```
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())
```

Otro caso de esta índole es la sentencia `assert`.

Asegúrate de indentar la línea continuada apropiadamente. El lugar preferido para “cortar” alrededor de un operador binario es después del operador, no antes.

Algunos ejemplos:

```
class Rectangle(Blob):

    def __init__(self, width, height,
                 color='black', emphasis=None, highlight=0):
        if (width == 0 and height == 0 and
            color == 'red' and emphasis == 'strong' or
            highlight > 100):
            raise ValueError("sorry, you lose")
        if width == 0 and height == 0 and (color == 'red' or
                                           emphasis is
None):
            raise ValueError("I don't think so -- values are
%s, %s" %
                             (width, height))
        Blob.__init__(self, width, height,
                      color, emphasis, highlight)
```

Líneas en blanco

Separa funciones de alto nivel y definiciones de clase con dos líneas en blanco.

Definiciones de métodos dentro de una clase son separadas por una línea en blanco.

Líneas en blanco adicionales pueden ser utilizadas (escasamente) para separar grupos de funciones relacionadas. Se pueden omitir entre un grupo (de funciones) de una línea relacionadas (por ejemplo, un conjunto de implementaciones ficticias).

Usa líneas en blanco en funciones, escasamente, para indicar secciones lógicas.

Python acepta el carácter control-L (^L) como un espacio en blanco; muchas herramientas tratan a estos caracteres como separadores de página, por lo que puedes utilizarlos para separar páginas de secciones relacionadas en un archivo. Nota: algunos editores y visores de código basados en la web pueden no reconocer control-L como dicho carácter y mostrarán otro glifo en su lugar.

Codificaciones (PEP 263)

El código en el núcleo de la distribución de Python siempre debería utilizar la codificación ASCII o Latin-1 (alias ISO-8859-1). Para Python 3.0 y en adelante, UTF-8 es preferible ante Latin-1, véase PEP 3120.

Archivos usando ASCII no deberían tener una “coding cookie” (*especificación de la codificación al comienzo del archivo*); de lo contrario, usar `\x`, `\u` o `\U` es la manera preferida para incluir caracteres que no correspondan a dicha codificación en cadenas (*strings*).

Para Python 3.0 y en adelante, la siguiente política es prescrita para la librería estándar (véase PEP 3131): todos los identificadores en la librería estándar de Python DEBEN usar caracteres correspondientes a la coficiación ASCII, y DEBERÍAN usar palabras en Inglés siempre que sea posible (en muchos casos, abreviaciones y términos técnicos son usados que no corresponden al idioma). Además, las cadenas literales (*string literals)* y los comentarios deben ser también ASCII. Las únicas excepciones son (a) pruebas para características no ASCII, y (b) nombres de autores. Autores cuyos nombres no están basados en el alfabeto latín DEBEN proveer una transcripción.
Se les recomienda a los proyectos de código abierto con gran audiencia adoptar una política similar.

Importaciones

- Las importaciones deben estar en líneas separadas, por ejemplo:

```
Sí:  import os
     import sys

No:  import sys, os
```

  Sin embargo, es correcto decir:

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

- Las importaciones siempre se colocan al comienzo del archivo, simplemente luego de cualquier comentario o documentación del módulo, y antes de globales y constantes.

Las importaciones deben estar agrupadas en el siguiente orden:

1. importaciones de la librería estándar
2. importaciones terceras relacionadas
3. importaciones locales de la aplicación / librería

Deberías poner una línea en blanco entre cada grupo.

Coloca cualquier especificación `__all__` luego de las importaciones.

- Las importaciones relativas (*relative imports*) para intra-paquete (*intra-package*) son altamente desalentadas (*poco recomendadas*) . Siempre usa la ruta absoluta del paquete para todas las importaciones. Incluso ahora que el PEP 328 está completamente implementado en Python 2.5, su estilo de importaciones relativas explícitas es activamente desalentado; las importaciones absolutas (*absolute imports)* son más portables y legibles.

- Al importar una clase desde un módulo que contiene una clase, generalmente está bien realizar esto:

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

Si esto causa coincidencias con nombres locales, realiza:

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

y usa “myclass.MyClass” y “foo.bar.yourclass.YourClass”.

Espacios en blanco en Expresiones y Sentencias

Manías

Evita usar espacios en blanco extraños en las siguientes situaciones:

- Inmediatamente dentro de paréntesis, corchetes o llaves:

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

- Inmediatamente antes de una coma, un punto y coma o dos puntos:

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

- Inmediatamente antes del paréntesis que comienza la lista de argumentos en la llamada a una función:

```
Sí: spam(1)
No: spam (1)
```

- Inmediatamente antes de un corchete que empieza una indexación o "slicing" (*término utilizado tanto en el ámbito de habla inglesa como española*):

```
Sí: dict['key'] = list[index]
No: dict ['key'] = list [index]
```

- Más de un espacio alrededor de un operador de asignación (u otro) para alinearlo con otro:

  Sí:

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

  No:

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

Otras recomendaciones

- Siempre rodea estos operadores binarios con un espacio en cada lado: asignación (`=`), asignación de aumentación (`+=`, `-=`, etc.), comparaciones (`==`, `<`, `>`, `!=`, **`<>`**, `<=`, `>=`, `in`, `not in`, `is`, `is not`), "Booleans" (`and`, `or`, `not`).

- Si se utilizan operadores con prioridad diferente, considera agregar espacios alrededor del operador con la menor prioridad. Usa tu propio criterio, de todas menaras, nunca uses más de un espacio, y siempre mantén la misma cantidad en ambos lados.

  Sí:

```
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)
```

- No uses espacios alrededor del = (igual) cuando es utilizado para indicar un argumento en una función (“keyword argument”) o un parámetro con un valor por defecto.

  Sí:

```
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)
```

- Las sentencias compuestas (múltiples sentencias en la misma línea) son generalmente desalentadas (*poco recomendadas*).

  Sí:

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

```
do_two()
do_three()
```

Más bien no:

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

- Mientras que en algunas ocasiones es aceptado el colocar un if/for/while con un cuerpo pequeño en la misma línea, nunca lo implementes para sentencias de múltiples cláusulas.

  Más bien no:

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

  Definitivamente no:

  ```
  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()
  ```

Comentarios

Comentarios que contradigan el código es peor que no colocar comentarios. ¡Siempre realiza una prioridad de mantener los comentarios al día cuando el código cambie!

Los comentarios deben ser oraciones completas. Si un comentario es una frase u oración, su primera palabra debe comenzar con mayúscula, a menos que sea un identificador que comienza con minúscula. ¡Nunca cambies las mayúsculas/minúsculas de los identificadores! (*Nombres de clases, objetos, funciones, etc.*).

Si un comentario es corto, el punto al final puede omitirse.

Comentarios en bloque generalmente consisten en uno o más párrafos compuestos por oraciones completas, por lo que cada una de ellas debe finalizar en un punto.

Deberías usar dos espacios luego de una oración que termine con un punto.

Al escribir en Inglés, se aplica “Strunk and White”.

Para los programadores de Python de países de habla no Inglesa: por favor escribe tus comentarios en Inglés, a menos que estés 120% seguro que tu código jamás será leído por gente que no hable tu idioma.

Comentarios en bloque

Los comentarios en bloque generalmente se aplican a algunos (o todos) códigos que los siguen, y están indentados al mismo nivel que ese código. Cada línea de un comentario en bloque comienza con un `#` (*numeral)* y un espacio (a menos que esté indentado dentro del mismo comentario).

Los párrafos dentro de un comentario en bloque están separados por una línea que contiene únicamente un `#` (*numeral*).

Comentarios en línea (*comentarios en la misma línea*)

Usa comentarios en línea escasamente.

Un comentario en línea es aquel que se encuentra en la misma línea que una sentencia. Los comentarios en línea deberían estar separados por al menos dos espacios de la sentencia. Deberían empezar con un # (*numeral*) seguido de un espacio.

Los comentarios en línea son innecesarios y de hecho molestos si su acción es obvia. No hagas esto:

```
x = x + 1                 # Incrementar x
```

Pero algunas veces es útil:

```
x = x + 1                 # Compensar el borde
```

Cadenas de documentación

Las convenciones para escribir buenas cadenas de documentación (alias “docstrings”) están inmortalizadas en el PEP 257.

- Escribe “docstrings” para todos los módulos, funciones, clases y métodos públicos. Los “docstrings” no son necesarios para los métodos no públicos, pero deberían tener un comentario que describa su función. Este comentario debe aparecer luego de la línea “def”.

- El PEP 257 describe las convenciones para los “docstrings”. Nótese lo más importante, el """ que finaliza un “docstring” multilínea debe estar solo en una línea, y preferiblemente precedido por una línea en blanco. Ejemplo:

```
"""Return a foobang

Optional plotz says to frobnicate the bizbaz first.

"""
```

- Para “docstrings” de una línea, está bien terminar el """ en la misma línea.

Contabilidad de Versiones

Si tienes que tener Subversion, CVS, o RCS en tu archivo de fuente, hazlo así:

```
__version__ = "$Revision: 70b79ccd671a $"
# $Source$
```

Estas líneas deben ser incluidas luego del “docstring” del módulo, antes de cualquier código, separado por una línea en blanco arriba y abajo.

Convenciones de nombramiento

Las convenciones de nombramiento (*o nomenclatura*) de la librería

Python son un poco desastrosas, por lo que nunca vamos a obtener esto completamente consistente – sin embargo, aquí hay algunos de los actualmente recomendados estándares de nombramiento. Los nuevos módulos o paquetes (incluyendo frameworks de terceros) deberían estar escritos en base a estos estándares, pero donde una librería existente tenga un estilo diferente, la consistencia interna es preferible.

Descriptivo: Estilos de nombramiento

No hay gran cantidad de diversos estilos de nombramiento. Te ayuda a ser capaz de reconocer qué estilo de nombramiento está siendo utilizado, independientemente de para qué están usados. *Las tildes no deben incluirse, tómese como una regla para mantener la traducción con la correcta ortografía. En cualquiera de los casos, no deben utilizarse caracteres con tildes para el nombramiento.*

Se distinguen los siguientes estilos:

- `b` (una letra en minúscula)
- `B` (una letra en mayúscula)
- `minúscula` (*lowercase*)
- `minúscula_con_guiones_bajos` (*lower_case_with_underscores*)
- `MAYÚSCULA` (*UPPERCASE*)
- `MAYÚSCULA_CON_GUIONES_BAJOS` (*UPPER_CASE_WITH_UNDERSCORES*)
- `PalabrasConMayúscula` (*CapitalizedWords*) (“CapWords” o “CamelCase”).

  Nota: Al momento de usar abreviaciones en “CapWords”, pon en mayúscula las letras de la abreviación. “HTTPServerError” es mejor que “HttpServerError”.

- `minúsculaMayúscula` (*mixedCase*) (difiere con “CapWords” por la primer letra en minúscula).

- `Palabras Con Mayúsculas Con Guiones Bajos` (*Capitalizad_Words_With_Underscores*) (¡horrible!).

Existe también el estilo de usar un prefijo único para identificar a un grupo de funciones relacionadas. Esto no es muy utilizado en Python, pero se menciona para cubrir los estilos en su totalidad. Por ejemplo, la función `os.stat()` retorna una tupla cuyos ítems tradicionalmente tienen nombres como `st_mode`, `st_size`, `st_mtime` y así. (Se realiza esto para enfatizar la correspondencia con los campos del "POSIX system call struct", que ayuda a los programadores a estar familiarizados.)

La libraría X11 usa un prefijo "X" para todas sus funciones públicas. En Python, este estilo es generalmente innecesario ya que tanto los métodos como los atributos están precedidos con un objeto, al igual que los nombres de las funciones lo están con el nombre de un módulo.
Además, la siguiente forma precedida por un guión bajo es reconocida (esta puede ser combinada con cualquiera de las convenciones nombradas anteriormente):

- `_simple_guion_bajo_como_prefijo` (*_single_leading_underscore*): débil indicador de "uso interno". Por ejemplo, `from M import *` no importa los objetos cuyos nombres comienzan con un guión bajo.

- `simple guion bajo como sufijo` (*single_trailing_underscore_*): utilizado como convención para evitar conflictos con un nombre ya existente, ejemplo:

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

- `__doble guion bajo como prefijo` (*__double_leading_underscore*): al nombrar un atributo en una clase, se invoca el "name mangling" (dentro de la clase FooBar, `__boo` se convierte en `_FooBar__boo`; véase abajo).

- `__doble_guion_bajo_como_prefijo_y_sufijo__` (*__double_leading_and_trailing_underscore__*): los objetos y atributos que viven en "namespaces" controlados por el usuario. Ejemplo, `__init__`, `__import__` o `__file__`. Nunca inventes nombres como esos; únicamente utiliza los

documentados.

Prescriptivo: Convenciones de nombramiento

Nombres para evitar

Nunca uses los caracteres ‘l’ (letra ele en minúscula), ‘O’ (letra o mayúscula), o ‘I’ (letra i mayúscula) como simples caracteres para nombres de variables.

En algunas fuentes, estos caracteres son indistinguibles de los números uno y cero. Cuando se quiera usar ‘l’, en lugar usa ‘L’.

Nombres de paquetes y módulos

Los módulos deben tener un nombre corto y en minúscula. Guiones bajos pueden utilizarse si mejora la legiblidad.
Los paquetes en Python también deberían tener un nombre corto y en minúscula, aunque el uso de guiones bajos es desalentado (*poco recomendado*).

Ya que los nombres de módulos están ligados a los de los archivos, y algunos sistemas operativos distinguen caracteres entre minúsculas y mayúsculas y truncan nombres largos, es importante que sean bastante cortos – esto no será un problema en Unix, pero podrá ser un problema cuando el código es portado a una antigua versión de Mac, Windows o DOS.

Cuando un módulo escrito en C o C++ provee una interfaz de alto nivel escrita en Python, debe llevar un guión bajo como prefijo (por ejemplo, `_socket`).

Nombres de clases

Casi sin excepción, los nombres de clases deben utilizar la convención “CapWords” (*palabras que comienzan con mayúsculas*). Clases para uso interno tienen un guión bajo como prefijo.

Nombres de excepciones

Debido a que las excepciones deben ser clases, se aplica la convención anterior. De todas maneras, deberías usar un sufijo

“Error” en los nombres de excepciones (en caso que corresponda a un error).

Nombres de variables globales

(Esperemos que esas variables sean únicamente para uso dentro del módulo). Las convenciones son las mismas que se aplican para las funciones.

Los módulos que estén diseñados para usarse vía `from M import *` deben usar el mecanismo `__all__` para prevenir la exportación de las variables globales, o usar la antigua convención especificando un guión bajo como prefijo (lo que tratarás de hacer es indicar esas globales como “no públicas”).

Nombres de funciones

Las funciones deben ser en minúscula, con las palabras separadas por un guión bajo, aplicándose éstos tanto como sea necesario para mejorar la legibilidad.

“mixedCase” (*primera palabra en minúscula*) es aceptado únicamente en contextos en donde éste es el estilo predominante (por ejemplo, threading.py) con el objetivo de mantener la compatibilidad con versiones anteriores.

Argumentos de funciones y métodos

Siempre usa `self` para el primer argumento de los métodos de instancia.

Siempre usa `cls` para el primer argumento de los métodos de clase.

Si los argumentos de una función coincide con una palabra reservada del lenguaje, generalmente es mejor agregar un guión bajo como sufijo antes de usar una abreviación u ortografía incorrecta. `class_` es mejor que `clss`. (Tal vez es mejor evitar las coincidencias usando un sinónimo.)

Nombres de métodos y variables de instancia

Usa las mismas reglas que para el nombramiento de funciones: en

minúscula con palabras separadas con guiones bajos, tantos como sea necesario para mejorar la legibilidad.

Usa un solo guión bajo como prefijo para métodos no públicos y variables de instancia.

Para evitar coincidencias con subclases, usa dos guiones bajos como prefijo para invocar las reglas del “name mangling” de Python.

Python cambia (*“mangles”, en el texto original*) estos nombres con el nombre de la clase: si la clase Foo tiene un atributo llamado `__a`, no puede ser accedido por `Foo.__a`. (Un usuario insistente aún puede acceder llamando a `Foo._Foo_a`.) Generalmente, el doble guión bajo como prefijo debería ser únicamente utilizado para evitar conflicto con los nombres en atributos y clases diseñados para ser parte de una subclase.

Nota: hay alguna controversia acerca del uso de __nombres (véase abajo).

Constantes

Las constantes son generalmente definidas a nivel módulo, escritas con todas las letras en mayúscula y con guiones bajos separando palabras. Por ejemplo, `MAX_OVERFLOW` y `TOTAL`.

Diseñando para herencia

Siempre decide si los métodos de clase o variables de instancia (atributos) deben ser públicos o no públicos. En caso de estar en una duda, elige no público; es más fácil de hacerlo público posteriormente que hacer un atributo público no público.

Los atributos públicos son aquellos los cuales esperas que los clientes no relacionados con tu clase usen, con tu compromiso de evitar cambios incompatibles con versiones anteriores. Los atributos no públicos son aquellos que no tienen la intención de ser utilizados por terceras personas; por lo que no das la garantía que éstos no sufrirán cambios o serán removidos.

No hablamos del término “privado”, ya que ningún atributo es realmente privado es Python (sin un innecesario monto de trabajo).

Otra categoría de atributos son aquellos que son parte del “API de subclase” (a menudo llamado “protegido” en otros lenguajes). Algunas clases están diseñadas para que sean heredadas, o para extender o modificar aspectos del comportamiento de la clase. Al momento de diseñar una clase de dicha índole, ten cuidado al hacer decisiones explícitas acerca de cuales atributos son públicos, cuales son parte del API de subclase, y cuales son de uso únicamente dentro de la clase.

Con esta mentalidad, a continuación hay algunas guías *Pythonicas*:

- Los atributos públicos no deben tener un guión bajo como prefijo.

- Si tu atributo público provoca problemas con una palabra reservada, agrega un guión bajo como sufijo al nombre. Esto es preferible ante una abreviación o mala ortografía. (De todas maneras, a pesar de esta regla, ‘cls’ es preferido ante cualquier variable o argumento que se conoce como una clase, especialmente el primer argumento de un método de clase.)

  Nota 1: Véase la recomendación de nombres de argumentos arriba para métodos de clase.

- Para atributos públicos de datos simples, es mejor exponer únicamente el nombre del atributo, sin complicados métodos de acceso. Ten en cuenta que Python provee una ruta fácil para futuras mejoras, encontrarás que un atributo de simples datos necesita aumentar su comportamiento funcional. En ese caso, utiliza propiedades para ocultar la implementación funcional debajo de un acceso por un atributo de simples datos.

  Nota 1: Las propiedades funcionan únicamente en las clases del nuevo estilo (*new-style classes*).

  Nota 2: Intenta mantener el comportamiento funcional libre de efecto secundario, a pesar de efectos secundarios como el almacenamiento en caché que generalmente están bien.

  Nota 3: Evita usar propiedades para operaciones muy pesadas; la notación de atributo hace creer que el acceso es relativamente liviano.

- Si tu clase está intencionada a ser utilizada como una subclase, y tienes atributos que no quieres que dicha subclase use, considera nombrarlos con dos guiones bajos como prefijo y sin sufijos. Esto invoca el algoritmo de “name mangling” de Python, donde el nombre de la clase es insertado dentro del nombre del atributo. Esto ayuda a evitar colisiones de nombres de atributos entre clase y subclase.

  Nota 1: Nótese que únicamente el nombre de la clase es insertado en el atributo en caso de invocar el “name mangling”, por lo que si una subclase escoge el mismo nombre tanto para la clase como el atributo, aún puedes obtener colisiones.

  Nota 2: El “name mangling” puede hacer ciertos usos, como depuración y `__getattr__()`, menos conveniente. De todas maneras, el algoritmo “name mangling” está bien documentado y es fácil de ejecutar manualmente.

  Nota 3: No a todos les agrada el “name mangling”. Intenta hacer un balance entre la necesidad de evitar colisiones de nombres accidentalmente.

Interfaces públicas e internas

Toda compatibilidad con antiguas versiones garantiza la aplicación únicamente en interfaces públicas. Por lo tanto, es importante que los usuarios sean capaces de distinguir claramente entre interfaces públicas e interfaces internas.

Las interfaces documentadas se consideran públicas, a menos que dicha documentación especifique que se trata de interfaces internas. Todas las interfaces indocumentadas deben ser asumidas como internas.

Para un mejor soporte de introspección, los módulos deberían declarar explícitamente los nombres en su API pública usando el atributo `__all__`. Aplicando una lista vacía a `__all__` indica que el módulo no tiene un API pública.

Incluso una vez especificado correctamente el atributo `__all__`, las interfaces internas (paquetes, módulos, clases, funciones, atributos u

otros nombres) deben estar prefijadas con un guión bajo.

Una interfaz es también considerada interna si cualquiera de los "namespace" que contiene (paquete, módulo o clase) es considerado interno.

Los nombres importados siempre deberían considerar un detalle de implementación. Otros módulos no deben confiar en el acceso indirecto a dichos nombres importados a menos que sean parte del API del módulo y estén explícitamente documentados, como `os.path` o el módulo `__init__` de un paquete que expone funcionalidad para submódulos.

Recomendaciones

- El código debe estar escrito de tal manera que no provoque inconvenientes con otras implementaciones de Python (PyPy, Jython, IronPython, Cython, Psyco, etc.).

  Por ejemplo, no te confíes en la eficiencia para con la concatenación de cadenas de CPython para sentencias del tipo `a += b` o `a = a + b`. Esta optimización es frágil incluso en CPython (funciona únicamente con algunos tipos) y no se encuentra presente en todas las implementaciones que no usan "refcounting". En esos casos, la forma `"".join()` debe ser utilizada en lugar de la anterior. Esto asegurará que la concatenación ocurra en tiempo lineal en las diversas implementaciones.

- Comparaciones con None deben siempre realizarse con `is` o `is not`, nunca con los operadores de igualdad.

  También, ten cuidado al escribir `if x` cuando lo que pretendes es `if x is not None` – por ejemplo, al momento de probar (*testear*) si a una variable o argumento que se inicializa por defecto a None se le aplicó algún otro valor. El otro valor podría tener un tipo (como un contenedor) que puede ser falso en un contexto "boolean".

- Al momento de implementar operaciones de ordenamiento con ricas comparaciones, es mejor implementar las seis operaciones (`__eq__`, `__ne__`, `__lt__`, `__le__`, `__gt__`, `__ge__`)

antes que confiarse en otro código que sólo ejerce una comparación particular.

Para minimizar el esfuerzo involucrado, el decorador `functools.total ordering()` provee una herramienta para generar métodos de comparación faltantes.

El PEP 207 indica que las reglas de reflexión son asumidas por Python. Así, el intérprete puede cambiar `y > x` con `x < y`, `y >= x` con `x <= y`, y también los argumentos de `x == y` y `x != y`. Las operaciones `sort()` y `min()` garantizan usar el operador < como así la función `max()` utiliza el operador >. De todas maneras, es mejor implementar las seis operaciones por lo que esa confusión no se manifiesta en otros contextos.

- Siempre usa una sentencia `def` en lugar de una sentencia de asignación que liga a la expresión lambda directamente a un nombre.

  Sí:

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

  No:

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

  La primera manera indica que el nombre del objeto resultante de la función es 'f' en lugar del genérico '<lambda>'. Esto es más útil para "tracebacks" y representaciones en cadena en general. El uso de la asignación por sentencia elimina el único beneficio que la expresión lambda puede ofrecer ante una sentencia def (por ejemplo, puede ser añadido dentro de una expresión más larga).

- Deriva excepciones desde `Exception` y no desde `BaseException`. La herencia directa desde `BaseException` es reservada para excepciones en donde capturarlas es casi siempre el método incorrecto de hacerlo.

  Diseña las jerarquías de excepciones basadas en la distinción en donde la excepción sea necesaria, y no en el lugar desde donde se lanzan (*"lanzar" de la palabra reservada "raise"*).

Apunta a responder la pregunta “¿Qué salió mal?” antes que únicamente afirmar que “Ha ocurrido un error” (véase PEP 3151 para un ejemplo).

Las convenciones de nombramiento de clases se aplican aquí, aunque debes añadir el sufijo “Error” a tu excepción si la misma es un error. Las excepciones que no correspondan a un error, sean utilizadas para flujo de control no local u otro tipo de señalización no necesitan un sufijo en especial.

- Utiliza el encadenamiento de excepciones apropiadamente. En Python 3, “raise X from Y” debe ser utilizado para indicar el reemplazo explícito sin perder el “traceback” original.

  Al momento de reemplazar deliberadamente una excepción interior (usando “raise X” en Python 2 o “raise X from None” en Python 3.3+), asegúrate que los detalles relevantes sean transferidos a la nueva excepción (tal como preservar el nombre del atributo al convertir KeyError a AttributeError, o añadiendo el texto original al mensaje de la nueva).

- Al momento de lanzar (*de la palabra reservada “raise”*) una excepción en Python 2, usa `raise ValueError('message')` en lugar del viejo método `raise ValueError, 'message'`.

  La última forma no está permitida en la sintaxis de Python 3.

  La forma de los paréntesis también indica que en el momento que los argumentos son demasiado largos o incluyen el formateo de cadenas, no necesitas usar los caracteres de continuación de la línea gracias a los paréntesis.

- Al momento de capturar excepciones, menciona sus nombres específicamente siempre que sea posible, en lugar de utilizar la simple claúsula `except:`.

  Por ejemplo, utiliza:

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

Una simple cláusula `except:` capturará las excepciones SystemExit y KeyboardInterrupt, haciendo más difícil de interrumpir un programa con Control-C, y puede causar otros problemas. Si quieres capturar todas las excepciones que señaliza errores del programa, utiliza `except Exception:` (el simple `except:` es equivalente a `except BaseException:`).

Una buena regla es limitar el uso de la simple cláusula “except:” a dos casos:

1. Si el manejador de la excepción va a estar imprimiendo o registrando el “traceback”; al menos el usuario va a estar al tanto que ocurrió un error.
2. Si el código necesita hacer un trabajo de limpieza, pero luego deja que la excepción se propague hacia arriba con raise. `try…finally` puede ser una mejor manera para manejar este caso.

- Al momento de asignar una excepción a un nombre, prefiere la siguiente sintaxis añadida en Python 2.6:

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

Esta es la única sintaxis soportada en Python 3, y evita tener problemas de ambigüedad asociados con la vieja sintaxis de la coma.

- Al momento de capturar errores del sistema operativo, prefiere la jerarquía de excepción explícita introducida en Python 3.3 en lugar de la introspección de los valores de `errno`.

- Adicionalmente, para todas las cláusulas try/except, limita la `try` al mínimo absoluto de líneas de código necesarias.
  De nuevo, esto evita errores:

  Sí:

```
try:
    value = collection[key]
except KeyError:
```

```
    return key_not_found(key)
else:
    return handle_value(value)
```

No:

```
try:
    # ¡Muy extenso!
    return handle_value(collection[key])
except KeyError:
    # También capturará la excepción KeyError lanzada
por handle_value()
    return key_not_found(key)
```

- Cuando un recurso es local para una sección de código particular, utiliza la sentencia `with` para asegurarte que es limpiada inmediatamente luego del uso. Una sentencia try/finally también es aceptada.

- Los manejadores de contexto (*context managers*) deben ser invocados desde funciones o métodos separadas siempre y cuando hacen algo diferente que adquirir y liberar recursos. Por ejemplo:

  Sí:

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

  No:

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

  Este último ejemplo no provee ninguna información para indicar que los métodos __enter__ y __exit__ están haciendo algo más que cerrar la conexión luego de la transacción. Ser explícito es importante en este caso.

- Utiliza los métodos de cadenas en lugar del módulo string.

  Los métodos de cadenas son siempre más rápidos y comparten el mismo API con las cadenas unicode. Pasa por alto esta regla si la compatibilidad con versiones anteriores a

Python 2.0 es requerida.

- Usa `''.startswith()` y `''.endswith()` en lugar de aplicar el string slicing para chequear prefijos o sufijos.

  startswith() y endswith() son más limpias y causan menos errores. Por ejemplo:

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

- Al realizar comparaciones de objetos siempre se debe usar isinstance() en lugar de comparar los tipos directamente.

```
Sí: if isinstance(obj, int):

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

- Al momento de chequear si un objeto es una cadena, ten en cuenta que puede ser unicode también. En Python 2, str y unicode tiene una clase base en común, basestring, por lo que puedes hacer:

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

  Nótese que en Python 3, `unicode` y `basestring` ya no existen (únicamente `str`) y un objeto de bytes no es más un tipo de cadena (en lugar de eso, es una secuencia de enteros).

- Para secuencias, (strings, listas, tuplas), haz uso del hecho que las que se encuentran vacías son falsas.

```
Sí: if not seq:
    if seq:

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

- No escribas cadenas literales que se basen en un importante espacio al final. Dichos espacios son visualmente indistinguibles y algunos editores (o los más nuevos, reindent.py) los removerán.

- No compares valores del tipo boolean con True o False usando ==.

```
Sí:   if greeting:
No:   if greeting == True:
Peor: if greeting is True:
```

- La librería estándar de Python no utilizará anotaciones en las funciones que resultaran comprometedoras ante un estilo particular. En lugar de eso, las anotaciones se dejan a los usuarios para descubrir y experimentar con estilos de anotación útiles.

  Se recomienda que los experimentos de terceros con anotaciones usen un decorador asociado para indicar cómo se debe interpretar dicha anotación.

  Antiguos desarrolladores del núcleo intentaban utilizar anotaciones inconsistentes, el estilo ad-hoc. Por ejemplo:

  - `[str]` era ambigua en cuanto representaba a una lista de cadenas o un valor que podía ser un str o None.
  - La notación `open(file:(str,bytes))` era usada para un valor que podía ser tanto `bytes` como `str` antes que una tupla de doble valor conteniendo un valor del tipo `str` seguido de uno del tipo `bytes`.
  - La anotación `seek(whence:int)` exhibía una mezcla de sobre-especificación y bajo-especificación: `int` es demasiado restrictivo (cualquiera con `__index__` sería aceptado) y no es lo suficientemente restrictiva (solo los valores 0, 1, y 2 están permitidos). Igualmente, la anotación `write(b:bytes)` era también muy restrictiva (cualquiera que soporte el protocolo de búfer sería aceptado).
  - Anotaciones como `readl(n: int=None)` eran auto-contradictorias ya que `None` no es un `int`. Aquellas como `source_path(self, fullname:str) -> object` presentaban confusión acerca de qué tipo de valor debía ser retornado.
  - Además de lo anterior, las anotaciones eran inconsistentes en el uso de tipos concretos versus tipos abstractos: `int` versus `Integral` y set/frozenset versus

MutableSet/Set.

- Algunas anotaciones en las clases base abstractas eran especificaciones incorrectas. Por ejemplo, las operaciones set-to-set requieren otro para ser otra instancia de Set antes que simplemente un `Iterable`.
- Un nuevo tema era que las anotaciones se volvieron parte de la especificación pero no estaban siendo probadas.
- En la mayoría de los casos, los “docstrings” ya incluían las especificaciones del tipo y lo hacían con mayor claridad que las anotaciones de las funciones. En los casos restantes, los “docstrings” se mejoraban al mismo tiempo que las anotaciones se removían.
- Las anotaciones de funciones observadas eran demasiado “ad-hoc” e inconsistentes para trabajar con un sistema coherente de chequeado automático de tipos o validación de argumentos. Dejando estas anotaciones en el código lo harían más difícil para realizar cambios posteriormente por lo que esas utilidades automáticas no podrían ser soportadas.

Copyright

Licencia