- Bitácora
- Traducciones
- Tutorial de Python

Guía de estilo del código Python

Por Guido van Rossum, Barry Warsaw

Traducción al castellano por Raúl González Duque
el día 10 de Agosto de 2007

Introducción

En este documento se listan distintas convenciones utilizadas en el código Python comprendido en la librería estándar de la distribución principal de Python. Por favor refierase al PEP que describe las guías de estilo del código C para la implementación en C de Python[1].

Este documento es una adaptación del ensayo original de Guido Guía de Estilo de Python[2], con algunos añadidos de la guía de estilo de Barry[5]. En los puntos en los que exista conflicto, se aplican las reglas de estilo de Guido para los propósitos de este PEP. Este PEP puede estar aún incompleto (de hecho, es posible que nunca llegue a completarse).

La Consistencia Estúpida es el Demonio de las Mentes Pequeñas

Una de las ideas clave de Guido es que el código se suele leer mucho más de lo que se escribe. Las guías de estilo que proporciona este documento están dirigidas a mejorar la legibilidad del código y hacerlo consistente a lo largo del amplio espectro del código Python. Como dice el PEP 20[6], "La legibilidad cuenta".

Una guía de estilo nos ayuda a lograr consistencia. Ser consistente con esta guía de estilo es importante. La consistencia dentro de un proyecto es aún más importante. La consistencia dentro de un módulo o función es la más importante.

Pero lo más importante es saber cuándo ser inconsistente -- algunas veces la guía de estilo simplemente no es aplicable. Cuando tengas dudas, usa tu juicio. Mira ejemplos y decide qué te parece mejor. ¡Y no dudes en preguntar!

Dos buenas razones para romper una regla en particular son:

1. Que al aplicar la regla el código se haga menos legible, incluso para alguien que esté acostumbrado a leer código que sigue las normas.
2. Para ser consistente con código relacionado que también la rompe (quizás por razones históricas) -- aunque esto también es una oportunidad de arreglar el desaguisado de otra persona (al más puro estilo XP).

Formateo del código

Indentación

Usa 4 espacios por cada nivel de indentación.

Para código realmente antiguo que no quieras estropear, puedes continuar usando tabuladores de 8 espacios.

¿Tabuladores o espacios?

Nunca mezcles tabuladores y espacios.

La forma más popular de indentar en Python es utilizar sólo espacios. La segunda forma más popular es usar sólo tabuladores. El código indentado con una mezcla de tabuladores y espacios debería reformatearse y usar espacios exclusivamente. Cuando se invoca el intérprete de línea de comandos de Python con la opción -t, este muestra avisos sobre código que mezcla tabuladores y espacios. Cuando se utiliza -tt estos avisos se convierten en errores. ¡Estas opciones son altamente recomendables!

Para nuevos proyectos, se recomienda firmemente el uso de espacios en lugar de tabuladores. La mayoría de los editores tienen características que hacen esto bastante sencillo.

Tamaño máximo de línea

Limita todas las líneas a un máximo de 79 caracteres.

Todavía existen muchos dispositivos por ahí que están limitados a 80 caracteres por línea; además limitando el ancho de las ventanas a 80 caracteres posibilitas el tener varias ventanas una al lado de otra. El ajuste de línea por defecto en este tipo de dispositivos no da buenos resultados. Por lo tanto, por favor limita todas las líneas a un máximo de 79 caracteres. Para cadenas de texto largas (cadenas de documentación o comentarios), es aconsejable limitar el ancho a 72 caracteres.

La forma preferida de dividir líneas largas es utilizar la característica de Python de continuar las líneas de forma implícita dentro de paréntesis, corchetes y llaves. Si es necesario, puedes añadir un par de paréntesis extra alrededor de una expresión, pero algunas veces queda mejor usar una barra invertida. Asegurate de indentar la línea siguiente de forma apropiada. 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")
        Blob.__init__(self, width, height,
                      color, emphasis, highlight)
```

Líneas en blanco

Separa las funciones no anidadas y las definiciones de clases con dos líneas en blanco.

Las definiciones de métodos dentro de una misma clase se separan con una línea en blanco.

Se pueden usar líneas en blanco extra (de forma reservada) para separar grupos de funciones relacionadas. Las líneas en blanco se pueden omitir entre un grupo de funciones con una sola línea (por ejemplo, con un conjunto de funciones sin implementación).

Usa líneas en blanco en las funciones, de forma limitada, para indicar secciones lógicas.

Python acepta el caracter control-L (o lo que es lo mismo ^L) como un espacio en blanco; muchas herramientas utilizan este caracter como separador de páginas, por lo que puedes usarlos para separar páginas de secciones relacionadas en tu archivo.

Codificación de caracteres (PEP 263)

El código de la distribución base de Python siempre debería utilizar las codificaciones ASCII o Latin-1 (también conocida como ISO-8859-1). Para Python 3.0 y superiores, se recomienda UTF-8 en lugar de Latin-1, ver PEP 3120.

Los archivos que usan ASCII (o UTF-8, para Python 3.0) no deberían tener una línea de especificación del juego de caracteres. Sólo debería usarse Latin-1 (o UTF-8) cuando se necesite mencionar a un autor cuyo nombre requiera de caracteres Latin-1 en un comentario o cadena de documentación; si no es así, la forma adecuada de incluir datos no ASCII en literales de cadena es mediante los caracteres de escape \x, \u y \U.

Para Python 3.0 y superiores, se recomienda la siguiente política para la librería estándar (ver PEP 3131): Todos los identificadores en la librería estándar Python DEBEN usar sólo caracteres ASCII, y DEBERÍAN usar palabras en inglés siempre que esto sea posible (en muchos casos se utilizan abreviaturas y términos técnicos que no forman parte del inglés). Además, los literales de cadena y los comentarios también deben estar en ASCII. Las únicas excepciones son (a) baterías de pruebas que comprueben las características no ASCII, y (b) nombres de autores. Los autores cuyos nombres no estén basados en el alfabeto latino DEBEN proporcionar una transcripción de sus nombres a este alfabeto.

Se recomienda a los proyectos de código abierto con audiencia global que adopten políticas similares.

Imports

- Normalmente los imports deberían colocarse en distintas líneas, por ejemplo:

  Sí:

  ```
  import os
  import sys
  ```

  No:

  ```
  import sys, os
  ```

  aunque también es correcto hacer esto:

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

- Los imports se colocan siempre en la parte superior del archivo, justo después de cualquier comentario o cadena de documentación del módulo, y antes de las variables globales y las constantes del módulo.

  Los imports deberían agruparse siguiendo el siguiente orden:

  1. imports de la librería estándar
  2. imports de proyectos de terceras partes relacionados

3. imports de aplicaciones locales/imports específicos de la librería

Deberías añadir una línea en blanco después de cada grupo de imports.

Si es necesario especificar los nombres públicos definidos por el módulo con __all__ esto debería hacerse después de los imports.

- Es muy desaconsejable el uso de imports relativos para importar código de un paquete. Utiliza siempre la ruta absoluta del paquete para todos los imports. Incluso ahora que el PEP 328 [7] está totalmente implementado en Python 2.5, el usar imports relativos se desaconseja seriamente; los imports absolutos son más portables y normalmente más legibles.
- Cuando importes una clase de un módulo, normalmente es correcto hacer esto

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

Si esto causa colisiones de nombres con objetos locales, utiliza en su lugar

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

y usa "myclass.MyClass" y "foo.bar.yourclass.YourClass" en el código

Espacios en blanco en expresiones y sentencias

Evita espacios en blanco extra en las siguientes situaciones:

- Inmediatamente después de entrar en un paréntesis o antes de salir de un paréntesis, corchete o llave.

  Sí:

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

  No:

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

- Inmediatamente antes de una coma, 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 de abrir un paréntesis para una lista de argumentos de una llamada a una función:

  Sí:

  ```
  spam(1)
  ```

  No:

  ```
  spam (1)
  ```

- Inmediatamente antes de abrir un paréntesis usado como índice o para particionar (slicing):

  Sí:

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

  No:

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

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

  Sí:

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

  No:

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

Otras Recomendaciones

- Rodea siempre los siguientes operadores binarios con un espacio en cada lado: asignación (=), asignación aumentada (+=, -= etc.), comparación (==, <, >, !=, <>, <=, >=, in, not in, is, is not), booleanos (and, or, not).

- Usa espacios alrededor de los operadores aritméticos:

  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 signo '=' cuando se use para indicar el nombre de un argumento o el valor de un parámetro 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)
  ```

- Generalmente se desaconsejan las sentencias compuestas (varias sentencias en la misma línea).

  Sí:

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

  Preferiblemente no:

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

- Aunque a veces es adecuado colocar un if/for/while con un cuerpo pequeño en la misma línea, nunca lo hagas para sentencias multi-clausula.

  Preferiblemente 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

Los comentarios que contradicen el código son peores que no tener ningún comentario. ¡Manten siempre como prioridad el mantener los comentarios actualizados cuando cambies el código!

Los comentarios deberían ser frases completas. Si un comentario es una frase o sentencia, la primera palabra debería estar en mayúsculas, a menos que sea un identificador que comience con una letra en minúsculas (¡nunca alteres el identificador sustituyendo mayúsculas o minúsculas!).

Si un comentario es corto, se puede omitir el punto al final. Los comentarios de bloque generalmente consisten en uno o más párrafos construidos con frases completas, y cada frase debería terminar con un punto.

Deberías usar dos espacios después de un punto de final de línea.

Cuando escribas en inglés, aplica las reglas de Strunk y White (N.T: Los autores de "The Elements of Style", un conocido libro de estilo de escritura en inglés).

Para los programadores en Python que no sean de países de habla inglesa: por favor escribe tus comentarios en inglés, a menos que estés un 120% seguro de que el código nunca será leído por gente que no hable tu idioma.

Comentarios de bloque

Los comentarios de bloque generalmente se aplican al código que se encuentra a continuación, y se indentan al mismo nivel que dicho código. Cada línea de un comentario de bloque comienza con un # y un único espacio (a menos que haya texto indentado dentro del comentario).

Los párrafos dentro de un comentario de bloque se separan por una línea conteniendo un único #.

Comentarios en línea

Utiliza comentarios en línea de forma moderada.

Un comentario en línea es un comentario que se encuentra en la misma línea que una sentencia. Los comentarios en línea deberían separarse por al menos

dos espacios de la sentencia que comentan. Deberían comenzar con un # y un espacio.

Los comentarios en línea son innecesarios y de hecho sólo sirven para distraer si indican cosas obvias. No hagas cosas como esta:

```
x = x + 1                 # Incrementa x
```

Aunque a veces, algo como esto puede ser de utilidad:

```
x = x + 1                 # Compensa por el borde
```

Cadenas de Documentación

Las convenciones para escribir buenas cadenas de documentación (también llamados "docstrings") se inmortalizaron en el PEP 257 [3].

- Escribe docstrings para todos los módulos, functiones, clases, y métodos públicos. Los docstrings no son necesarios para métodos no públicos, pero deberías tener un comentario que describa lo que hace el método. Este comentario debería aparecer antes de la línea "def".
- El PEP 257 describe las convenciones para buenas cadenas de documentación. Es importante notar, que el """ que finaliza un docstring de varias líneas debería situarse en una línea separada, y preferiblemente precederse por una línea en blanco, por ejemplo:

  ```
  """Return a foobang

  Optional plotz says to frobnicate the bizbaz first.

  """
  ```

- Para cadenas de documentación de una línea, es adecuado mantener el """ de cierre en la misma línea.

Mantenimiento de datos de versión

Si necesitas incluir datos de Subversion, CVS, o RCS en tus archivos de código fuente, hazlo de la siguiente forma.

```
__version__ = "$Revision: 56036 $"
# $Source$
```

Estas líneas deberían incluirse antes de la cadena de documentación del módulo, antes de cualquier otro código, y separadas por una línea en blanco encima y debajo.

Convenciones de nombres

Las convenciones de nombres de la librería de Python son un pequeño desastre, así que nunca conseguiremos que sea completamente consistente -- aún así, aquí tenéis los estándares de nombrado recomendados en la actualidad. Los nuevos módulos y paquetes (incluyendo los frameworks de terceras personas) deberían acomodarse a estos estándares, pero donde exista una librería con un estilo distinto, se prefiere la consistencia interna.

Descriptivo: Estilos de Nombrado

Hay un montón de estilos de nombrado distintos. Ayuda el poder reconocer qué estilo de nombrado se utiliza, independientemente de para lo que se utilice.

Podemos encontrarnos con los siguientes estilos de nombrado más comunes:

- b (una sola letra en minúsculas)
- B (una sola letra en mayúsculas)
- minusculas
- minusculas_con_guiones_bajos
- MAYUSCULAS
- MAYUSCULAS_CON_GUIONES_BAJOS
- PalabrasEnMayusculas (CapWords o CamelCase -- llamado así por el parecido de las mayúsculas con las jorobas de los camellos[4]). Algunas veces también se llaman StudlyCaps.

  Nota: Cuando se usan abreviaturas en CapWords, mantén en mayúsculas todas las letras de la abreviatura. Por lo tanto HTTPServerError es mejor que HttpServerError.

- capitalizacionMezclada (¡se diferencia de PalabrasEnMayusculas porque la primera letra está en minúsculas!)
- Mayusculas_Con_Guiones_Bajos (¡feo!)

También existe un estilo en el cuál se utiliza un prefijo único corto para agrupar nombres relacionados de forma conjunta. Esto no se suele utilizar mucho en Python, pero se menciona con el objeto de que el texto sea lo más completo posible. Por ejemplo, la función os.stat() devuelve una tupla cuyos elementos tradicionalmente han tenido nombres como st_mode, st_size, st_mtime y

similares. (Esto se hace para enfatizar la correspondencia con los campos de la estructura de la llamada a sistema POSIX, lo cual es de utilidad para los programadores que están familiarizados con ella.)

La librería X11 utiliza una X inicial para todas sus funciones públicas. En Python, este estilo generalmente se considera innecesario porque los atributos y métodos se preceden por el objeto al que pertenencen, y los nombres de funciones se preceden del nombre del módulo.

Además se reconocen las siguientes formas especiales usando guiones bajos al inicio o final del nombre (generalmente estos se pueden combinar con cualquier convención de capitalización):

- _guion_bajo_al_inicio: indicador débil de "uso interno". Por ejemplo "`from M import *`" no importa objetos cuyos nombres comiencen con un guión bajo.
- guion_bajo_al_final_: usado por convención para evitar conflictos con palabras clave de Python, por ejemplo

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

- __doble_guion_bajo_al_inicio: cuando se use para nombrar un atributo de clase, invoca la característica de "planchado de nombres" (dentro de la clase FooBar, __boo se convierte en _FooBar__boo; ver abajo).
- __doble_guion_bajo_al_inicio_y_fin__: objectos "mágicos" o atributos que viven en espacios de nombres controlados por el usuario. Por ejemplo __init__, __import__ o __file__. Nunca inventes nombres como estos; utilizalos sólo como están documentados.

Prescriptivo: Convenciones de Nombrado

Nombres a Evitar

Nunca utilices los caracteres 'l' (letra ele minúscula), 'O' (letra o mayúscula), o 'I' (letra i mayúscula) como nombres de variables de un solo caracter.

En algunas fuentes, estos caracteres son indistinguibles de los numerales uno y cero. Cuando estes tentado de usar 'l', utiliza una 'L' en su lugar.

Nombres de Paquetes y Módulos

Los módulos deberían tener nombres cortos formados en su totalidad por letras minúsculas. Se puede utilizar guiones bajos en el nombre del módulo si mejora la legibilidad. Los paquetes Python también deberían tener nombres cortos

formados de letras minúsculas, aunque se desaconseja el uso de guiones bajos.

Dado que los nombres de los módulos se mapean a nombres de archivos, y algunos sistemas de ficheros no diferencian entre mayúsculas y minúsculas y truncan los nombres largos, es importante que los nombres de los módulos que se elijan sean suficientemente cortos -- esto no es un problema en Unix, pero puede ser un problema cuando el código se porta a versiones antiguas de Mac o Windows, o a DOS.

Cuando un módulo de extensión escrito en C o C++ tenga un módulo Python asociado que proporcione una interfaz de más alto nivel (por ejemplo, más orientado a objetos), el módulo C/C++ debería comenzar con un guión bajo (por ejemplo _socket).

Nombres de Clases

Casi sin excepciones, los nombres de clases usan la convención CapWords. Las clases de uso interno tienen además un guión bajo al principio del nombre.

Nombres de Excepciones

Dado que las excepciones deberían ser clases, se aplica la convención relativa al nombrado de clases. De todas formas, deberías usar el sufijo "Error" en los nombres de las excepciones (si la excepción es realmente un error).

Nombres de Variables Globales

(Esperamos que estas variables estén destinadas a ser usadas sólo dentro de un módulo.) Las convenciones son prácticamente las mismas que para las funciones.

Los módulos diseñados para ser utilizados usando "from M import *" deberían usar el mecanismo __all__ para prevenir que se exporten variables globales, o bien usar la convención antigua de añadir un guión bajo como prefijo a dichas variables globales (puede que quieras hacer esto para indicar que estas variables son "variables no públicas del módulo").

Nombres de Funciones

Los nombres de funciones deberían estar en letras minúsculas, con palabras separadas mediante guiones bajos según sea necesario para mejorar la legibilidad.

Sólo se acepta capitalizacionMezclada en contextos en los que ya se trate del

estilo principal (por ejemplo threading.py), para mantener la compatibilidad hacia atrás.

Argumentos de funciones y métodos

Utiliza siempre 'self' como primer argumento de los métodos de instancia.

Utiliza siempre 'cls' como primer argumento de métodos de clase.

Si el nombre de un argumento de una función colisiona con una palabra reservada, generalmente es mejor añadir un guión bajo al final del nombre en lugar de usar una abreviatura o modificar la grafía. Por lo tanto "print_" se considera mejor que "prnt". (Quizás utilizar sinónimos sea una forma incluso mejor.)

Nombres de métodos y variables de instancia

Utiliza las reglas de los nombres de funciones: minúsculas con palabras separadas por guiones bajos cuando sea necesario para mejorar la legibilidad.

Utiliza guiones bajos al inicio sólo para métodos no públicos y variables de instancia.

Para evitar colisiones de nombres con subclases, utiliza dos guiones bajos al principio del nombre para invocar las reglas de planchado de nombres de Python.

Python une estos nombres con el nombre de la clase: si la clase Foo tiene un atributo llamado __a, no se puede acceder utilizando Foo.__a. (Un usuario insistente podría acceder utilizando Foo._Foo__a.) Generalmente, sólo se debería usar un doble guión al inicio para evitar conflictos de nombres con atributos en clases diseñadas para que se herede de ellas.

Nota: existe controversia sobre el uso de __nombres (ver debajo).

Diseñar para la herencia

Decide siempre si los métodos y variables de instancia de una clase (llamados de forma colectiva "atributos") deberían ser públicos o no públicos. Si dudas, elige que sea no público; es más sencillo convertirla en público más tarde que hacer no público un atributo que antes era público.

Los atributos públicos son aquellos que esperas que sean utilizados por los clientes de tu clase, y para los cuales te comprometes a evitar cambios que provoquen incompatibilidades hacia atrás. Los atributos no públicos son

aquellos que no están destinados a que sean utilizados por terceras personas; no ofreces ninguna garantía de que los atributos no públicos no vayan a cambiar o a eliminarse.

No utilizamos el término "privado" aquí, dado que ningún atributo es realmente privado en Python (sin un trabajo añadido generalmente innecesario).

Otra categoría de atributos son aquellos que son parte de la "API de las subclases" (a menudo llamado "protected" en otros lenguajes). Algunas clases se diseñan para que se herede de ellas, bien para extender o bien para modificar aspectos del comportamiento de la clase. Cuando se diseña una clase de esta forma, es necesario tomar algunas decisiones explícitas sobre qué atributos van a ser públicos, cuáles son parte de la API de la subclase, y cuáles están pensados para ser utilizados exclusivamente dentro de la clase actual.

Teniendo esto en cuenta, aquí están las líneas a seguir:

- Los atributos públicos no deberían tener guiones bajos al inicio del nombre.
- Si el nombre de tu atributo público colisiona con el de una palabra reservada, añade un único guión bajo al final del nombre del atributo. Esto es preferible a abreviar o cambiar la grafía de la palabra.

  Nota 1: Ver la recomendación anterior para métodos de clase.
- Para campos públicos simples, es mejor exponer sólo el nombre del atributo, sin complicaciones de métodos para accederlos y modificarlos (accessors y mutators). Ten en cuenta que Python proporciona una forma sencilla de modificarlo, en caso de que dentro de un tiempo lo necesites. En ese caso, utiliza propiedades para ocultar la implementación funcional con una sintaxis simple de acceso a atributos de datos.

  Nota 1: Las propiedades sólo funcionan en las nuevas clases.

  Nota 2: Intenta mantener el comportamiento funcional libre de efectos colaterales, aunque funcionalidad colateral como el cacheo generalmente es aceptable.

  Nota 3: Evita usar propiedades para realizar operaciones que requieran de grandes cálculos; el utilizar la notación de atributos hace que la persona que accede al atributo piense que el acceso es (relativamente) barato en tiempo computacional.

Recomendaciones para la programación

- El código debería escribirse de forma que no pusiera en desventaja otras implementaciones de Python (PyPy, Jython, IronPython, Pyrex, Psyco, y similares).

  Por ejemplo, no te apoyes en la eficiencia de la implementación de la concatenación de cadenas con a+=b o a=a+b en CPython. Estas órdenes se ejecutan más lentamente en Jython. En partes de la librería en las que el rendimiento sea importante, se debería utilizar en su lugar la forma ''.join(). Esto asegura que el tiempo necesario para la concatenación es del mismo orden en diferentes implementaciones.

- Las comparaciones con singletons como None siempre deberían llevarse a cabo con 'is' o 'is not', nunca con operadores de igualdad.

  También es importante tener cuidado con escribir "if x" cuando lo que realmente queríamos decir es "if x is not None" -- por ejemplo, cuando comprobemos si a una variable o argumento que tiene como valor por defecto None se le ha dado otro valor. ¡El otro valor podría tener un tipo (como un contenedor) que podría ser falso en un contexto booleano!

- Utiliza excepciones basadas en clases.

  Las excepciones en forma de cadenas están prohibidas en el código nuevo, dado que esta característica del lenguaje se eliminará en Python 2.6.

  Los módulos o paquetes deberían definir sus propias clases excepción específicas para el dominio del problema, las cuales deberían heredar de la clase Exception. Incluye siempre una cadena de documentación para la clase. Por ejemplo:

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

  En este caso se aplican las convenciones de nombrado de las clases, aunque deberías añadir el sufijo "Error" a las clases excepción, si la excepción se trata de un error. Las excepciones que no sean errores, no necesitan ningún sufijo especial.

- Cuando lances una excepción, utiliza `"raise ValueError('mensaje')"` en lugar de la forma antigua `"raise ValueError, 'mensaje'"`.

  Se prefiere la forma con paréntesis porque cuando los argumentos de la excepción son largos o incluyen formateo de cadenas, no necesitas usar caracteres para indicar que continúa en la siguiente línea gracias a los paréntesis. La forma antigua se eliminará en Python 3000.

- Cuando captures excepciones, menciona excepciones específicas cuando sea posible en lugar de utilizar una clausula 'except:' genérica.

  Por ejemplo, utiliza:

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

  Una cláusula 'except:' genérica capturaría las excepciones SystemExit y KeyboardInterrupt, complicando el poder interrumpir el programa usando Control-C, y pudiendo ocultar otros problemas. Si quieres capturar todas las excepciones relativas a errores en el programa, utiliza 'except Exception:'.

  Una buena regla a seguir es limitar el uso de las claúsulas 'except' genéricas a dos casos:

  1. Si el manejador de la excepción imprimirá o registrará el traceback; de esta forma al menos el usuario sabrá que ha ocurrido un error.

  2. Si el código tiene que realizar algún trabajo de limpieza, pero en ese caso tenemos que hacer que la excepción se propague hacia arriba usando 'raise'. La construcción 'try...finally' es una mejor forma de tratar con este caso.

- Adicionalmente, para todas las clausulas try/except, es necesario limitar la clausula 'try' a la mínima expresión de código necesaria. De nuevo, esto se hace para evitar ocultar bugs.

  Sí:

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

  No:

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

- Utiliza métodos de cadenas en lugar del módulo string.

Los métodos de las cadenas son siempre mucho más rápidos y comparten la misma API con las cadenas unicode. Ignora esta regla si es necesaria compatibilidad hacia atrás con versiones de Python anteriores a la 2.0.

- Utiliza ''.startswith() y ''.endswith() en lugar del particionado de cadenas para comprobar prefijos y sufijos.

  startswith() y endswith() son más limpios y menos propensos a errores. Por ejemplo:

  Sí:

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

  No:

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

  La excepción a esta norma se da si es necesario que el código funcione con Python 1.5.2 (¡esperemos que no lo necesites!).

- Las comparaciones del tipo de objeto siempre deberían utilizar isinstance() en lugar de comparar tipos directamente.

  Sí:

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

  No:

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

  Cuando comprobemos si un objeto es una cadena, ¡hay que tener en cuenta que puede que se trate de una cadena unicode! En Python 2.3, str y unicode tienen una clase padre común, basestring, por lo que puedes usar:

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

  En Python 2.2, el módulo types define el tipo StringTypes para este propósito en concreto, por ejemplo:

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

  En Python 2.0 y 2.1, tendríamos que hacer:

  ```
  from types import StringType, UnicodeType
  if isinstance(obj, StringType) or \
     isinstance(obj, UnicodeType) :
  ```

- Para secuencias, (cadenas, listas, tuplas), aprovecha el que las secuencias

vacias son equivalentes al falso booleano.

Sí:

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

No:

```
if len(seq)
if not len(seq)
```

- No compares valores booleanos con True o False usando ==

  Sí:

  ```
  if greeting:
  ```

  No:

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

  Peor:

  ```
  if greeting is True:
  ```

Referencias

[1] PEP 7, Guía de Estilo para Código C, van Rossum

[2] http://www.python.org/doc/essays/styleguide.html

[3] PEP 257, Convenciones para Cadenas de Documentación, Goodger, van Rossum

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

[5] Guía de estilo de GNU Mailman de Barry http://barry.warsaw.us/software/STYLEGUIDE.txt

[6] PEP 20, El Zen de Python

[7] PEP 328, Imports: Multi-Line and Absolute/Relative