Guía de Estilo para Codificación en Python

*Versión 1.0*

# Tabla de contenido

[Tabla de contenido 2](#_Toc530160638)

[Historial de versiones 4](#_Toc530160639)

[Introducción 5](#_Toc530160640)

[Diseño del código 5](#_Toc530160641)

[Indentación 5](#_Toc530160642)

[Longitud de líneas 6](#_Toc530160643)

[Líneas en blanco 7](#_Toc530160644)

[Codificación de archivos 7](#_Toc530160645)

[Importaciones 7](#_Toc530160646)

[Comillas y cadenas 8](#_Toc530160647)

[Espacios en expresiones y sentencias 8](#_Toc530160648)

[Otras recomendaciones 10](#_Toc530160649)

[¿Cuando usar comas finales? 12](#_Toc530160650)

[Comentarios 12](#_Toc530160651)

[Comentarios en bloque 12](#_Toc530160652)

[Comentarios en línea 13](#_Toc530160653)

[Cadenas de documentación 13](#_Toc530160654)

[Cadenas de documentación de una sola línea 14](#_Toc530160655)

[Cadenas de documentación de varias líneas 14](#_Toc530160656)

[Reglas de nombrado 21](#_Toc530160657)

[Estilos de nombramiento 21](#_Toc530160658)

[Normas de nombrado de Namtrik Development 22](#_Toc530160659)

[Nombres a evitar 22](#_Toc530160660)

[Nombres de paquetes y módulos 22](#_Toc530160661)

[Nombres de clases 22](#_Toc530160662)

[Nombres de tipos de variables 22](#_Toc530160663)

[Nombres de excepciones 23](#_Toc530160664)

[Nombres de variables globales 23](#_Toc530160665)

[Nombres de funciones y variables 23](#_Toc530160666)

[Argumentos de funciones y métodos 23](#_Toc530160667)

[Nombres de métodos y atributos de una clase 23](#_Toc530160668)

[Constantes 23](#_Toc530160669)

[Diseñando para herencia 23](#_Toc530160670)

[Recomendaciones 24](#_Toc530160671)

[Recomendaciones para codificación en Django 27](#_Toc530160672)

[Plantillas 27](#_Toc530160673)

[Vistas 28](#_Toc530160674)

[Modelos 28](#_Toc530160675)

[Misceláneos 29](#_Toc530160676)

[Herramientas 30](#_Toc530160677)

[Glosario de términos 30](#_Toc530160678)

[Referencias 32](#_Toc530160679)

# Historial de versiones

|  |  |  |
| --- | --- | --- |
| **Fecha** | **Descripción** | **Autor** |
| 14/11/2018 | Creación del documento | Cristhiam Gabriel Fernández Ruales |
| 19/11/2018 | Revisión del documento | Carlos Andrés Ordoñez Hernández |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

El presente documento es de uso privado de Namtrik Development. Este manual ha sido concebido para el uso del equipo de trabajo de Namtrik Development en el marco de los procesos de desarrollo de software basado en la norma ISO/IEC 29110. Este documento recopila y organiza publicaciones de diferentes autores divulgadas bajo licencia Creative Commons para quienes se reconocen sus derechos morales sobre el contenido. Queda totalmente prohibido el uso de este documento fuera de las actividades de Namtrik Development.

# Introducción

Este documento brinda las convenciones de escritura de código Python abarcando la librería estándar.

Esta guía es una adaptación de Python Enhancement Proposals 8 ([PEP8 – Style Guide for Python Code](https://www.python.org/dev/peps/pep-0008/)), Python Enhancement Proposals 257 ([PEP257 – Docstring Conventions](https://www.python.org/dev/peps/pep-0257/)), así como del documento de [Estilo de Codificación V2.1](https://docs.djangoproject.com/en/2.1/internals/contributing/writing-code/coding-style/) de Django Framework.

Las pautas que se proveen en este documento tienen como objetivo mejorar la legibilidad del código y hacerlo consistente.

# Diseño del código

## Indentación

Se deben usar 4 espacios por cada nivel de indentación.

Las líneas de código continuas deben alinearse verticalmente con el carácter que fue utilizado (paréntesis, llaves o corchetes).

**Sí**

*# Alineación con el paréntesis de apertura*

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

var\_three, var\_four)

**No**

*# Parámetros en primera línea cuando no se usa alineación vertical*

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

var\_three, var\_four)

*# Indentación indistinguible*

**def** long\_function\_name(

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

var\_four):

**print**(var\_one)

Vale la pena señalar que la combinación de una palabra clave de dos caracteres (por ejemplo, **if**) más un espacio, más un paréntesis de apertura crea una indentación natural de 4 espacios para las siguientes líneas del condicional de múltiples líneas. Esto puede producir un conflicto visual en el conjunto de código con indentación anidada dentro de la sentencia **if**, que también se indentaría de forma natural con 4 espacios. En estos casos se incluirá una línea de comentario que rompa dicho conflicto visual.

*# Agregue un comentario que facilite la distinción del código*

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

that\_is\_another\_thing):

*# Comentario que separa los argumentos de las instrucciones*

do\_something()

El paréntesis, corchete o llave que cierra una asignación debe estar alineado con el carácter inicial de la primera línea que lo abrió:

my\_list = [

1, 2, 3,

4, 5, 6,

]

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

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

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

)

Nunca se deben mezclar tabulaciones y espacios. Los espacios son el método recomendado para la indentación.

## Longitud de líneas

Limite todas las líneas del código fuente a un máximo de 79 caracteres.

Limitar el ancho de la ventana del editor permite tener varios archivos abiertos lado a lado y funciona muy bien cuando se usan herramientas de revisión de código que presentan las dos versiones en columnas adyacentes (por ejemplo, diff).

La forma recomendada para ajustar líneas muy largas es mediante el uso de la continuación de líneas dentro de paréntesis, corchetes o llaves. Las líneas largas se pueden dividir en varias líneas envolviendo expresiones entre paréntesis. Se recomienda este método en lugar de usar el carácter backslash para la continuación de la línea.

El carácter backslash puede ser apropiada en varias ocasiones. Por ejemplo, las declaraciones largas y múltiples que no pueden usar continuación con paréntesis, corchetes o llaves.

**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 similar es la sentencia **assert**.

Asegúrese de indentar la continuación de línea apropiadamente.

El lugar preferido para ajustar la línea alrededor de un operador binario es antes del operador, no después. Esto evita que los operadores se dispersen en diferentes columnas en la pantalla y que los operadores se alejen de sus operandos.

**Sí**

*# Fácil para unir operadores con operandos*

income = (gross\_wages

+ taxable\_interest

+ (dividends - qualified\_dividends)

- ira\_deduction

- student\_loan\_interest)

**No**

*# Los operadores se alejan de los operandos*

income = (gross\_wages +

taxable\_interest +

(dividends - qualified\_dividends) -

ira\_deduction -

student\_loan\_interest)

## Líneas en blanco

Rodee la definición de clases y funciones de alto nivel con dos líneas en blanco. Las definiciones de los métodos dentro de una clase están rodeadas por una sola línea en blanco.

Se permite el uso de líneas en blanco adicionales (con moderación) para separar grupos de funciones relacionadas. También es permitido el uso de líneas en blanco dentro de las funciones, con moderación, para indicar secciones lógicas.

## Codificación de archivos

El código fuente escrito en Python siempre deberá tener la codificación UTF-8.

## Importaciones

Las importaciones deben estar en líneas separadas, por ejemplo:

**Sí**

**import** **os**

**import** **sys**

**No**

**import** **os**, **sys**

Sin embargo, es correcto decir:

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

Las importaciones siempre deben estar en la parte superior del archivo, justo después de los comentarios y los docstrings del módulo, y antes de las variables globales y constantes del módulo.

Las importaciones deben agruparse en el siguiente orden:

* Importaciones de la librería estándar
* Importaciones de terceros relacionadas
* Importaciones específicas de aplicaciones o librerías locales

Ponga una línea en blanco entre cada grupo de importaciones.

En lo posible usar importaciones absolutas, ya que son más legibles y tienden a proporcionar mejores mensajes de error.

**import** **mypkg.sibling**

**from** **mypkg** **import** sibling

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

Las importaciones relativas (**from** **.** **import** sibling) o con comodines (**from** **<module>** **import** \*) no serán permitidas, ya que no dejan claro qué nombres están presentes en el espacio de nombres, lo que confunde tanto a los lectores como a muchas herramientas automatizadas.

Se recomienda usar [isort](https://github.com/timothycrosley/isort) para automatizar el ordenamiento de las importaciones.

# Comillas y cadenas

En Python, las cadenas entre comillas simples y las cadenas entre comillas dobles son iguales. Se recomienda usar comillas dobles de forma predeterminada. Sin embargo, cuando una cadena contiene caracteres de comillas simples o dobles, use la otra para evitar el uso de backslashes en la cadena. Esto mejora la legibilidad.

Para cadenas de comillas triples, siempre use comillas dobles para ser coherentes con lo expuesto anteriormente.

my\_string = "Hello world"

other\_string = "This string contains 'single-quote'"

**print**("""Hello

world""")

# Espacios en expresiones y sentencias

Evite los 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 } )

* Entre una coma final y un paréntesis de cierre:

**Sí**

foo = (0,)

**No**

bar = (0, )

* 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

* En un slice, los dos puntos actúan como un operador binario, y deben tener la misma cantidad de espacios en cada lado (tratándolo como el operador con la prioridad más baja). En un slice extendido, ambos dos puntos deben tener la misma cantidad de espacio. Excepción: cuando se omite un parámetro de segmento se omite el espacio.

**Sí**

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]

**No**

ham[lower + offset:upper + offset]

ham[1: 9], ham[1 :9], ham[1:9 :3]

ham[lower : : upper]

ham[ : upper]

* Inmediatamente antes del paréntesis de apertura que inicia la lista de argumentos de un llamado de función:

**Sí**

spam(1)

**No**

spam (1)

* Inmediatamente antes del corchete de apertura que inicia una indexación o un slice:

**Sí**

dct['key'] = lst[index]

**No**

dct ['key'] = lst [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

* Evite los espacios en blanco al final de cada línea. Debido a que generalmente son invisibles, puede ser confuso: por ejemplo, un backslash seguido de un espacio y una nueva línea no cuenta como un marcador de continuación de línea.
* Rodee siempre los siguientes operadores binarios con un solo espacio en cada lado: asignación (**=**), asignación (**+=**, **-=**, etc.), comparaciones (**==**, **<**, **>**, **!=**, **<>**, **<=**, **>=**, **in**, **not in**, **is**, **is not**), booleanos (**and**, **or**, **not**).
* Si se usan operadores con diferentes prioridades, considere agregar espacios en blanco alrededor de los operadores con las prioridades más bajas. Use su propio juicio; sin embargo, nunca use más de un espacio, y siempre tenga la misma cantidad de espacios en blanco en ambos lados de un operador binario.

**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)

* Las anotaciones de funciones deben usar las reglas normales para los dos puntos y siempre deben tener espacios alrededor de la flecha -> si está presente.

**Sí**

**def** munge(input: AnyStr): ...

**def** munge() -> AnyStr: ...

**No**

**def** munge(input:AnyStr): ...

**def** munge()->PosInt: ...

* No use espacios alrededor del signo = cuando se usa para indicar un argumento de palabra clave, o cuando se usa para indicar un valor predeterminado para un argumento de función no anotado.

**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)

Sin embargo, al combinar una anotación de un argumento con un valor predeterminado, use espacios alrededor del signo =

**Sí**

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

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

**No**

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

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

* Las declaraciones compuestas (declaraciones múltiples en la misma línea) están prohibidas.

**Sí**

**if** foo == 'blah':

do\_blah\_thing()

do\_one()

do\_two()

do\_three()

**No**

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

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

# ¿Cuando usar comas finales?

Las comas finales son opcionales, excepto que son obligatorias al hacer una tupla de un elemento. Para mayor claridad, se debe rodear la tupla entre paréntesis.

FILES = ('setup.cfg',)

Aunque las comas finales son redundantes, a menudo son útiles cuando se espera que una lista de valores, argumentos o elementos importados se extienda a lo largo del tiempo. El patrón es colocar cada valor en una línea por sí mismo, siempre agregando una coma al final, y agregar paréntesis, corchete o llave en la siguiente línea. Sin embargo, no tiene sentido tener una coma al final en la misma línea que el delimitador de cierre (excepto en el caso anterior de la tupla de valor único).

**Sí**

FILES = [

'setup.cfg',

'tox.ini',

]

initialize(FILES,

error=True,

)

**No**

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

initialize(FILES, error=True,)

# Comentarios

Los comentarios que contradigan el Código no son permitidos. Los comentarios deben actualizarse inmediatamente cuando cambie el código.

Los comentarios deben ser oraciones completas. La primera palabra debe escribirse iniciando con mayúscula, a menos que sea un identificador (nombre de clase, objetos, funciones, etc.) que inicia con minúscula.

Los comentarios en bloque generalmente consisten en uno o más párrafos construidos a partir de oraciones completas, y cada oración termina en un punto.

Los comentarios deben ser escritos en español (a excepción que se nombre un identificador), y deben ser lo suficientemente claros. Evite el uso de “nosotros” en el comentario; por ejemplo, “itera sobre” en lugar de “iteramos sobre”.

## Comentarios en bloque

Los comentarios en bloque generalmente se usan a sentencias o bloques de código que pueden ser confusos, 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).

*# Ejemplo de comentario en bloque.*

*#*

*# Genial, no?*

answer = 42

## Comentarios en línea

Use los comentarios en línea con moderación.

Un comentario en línea es un comentario en la misma línea que una sentencia. Los comentarios en línea deben estar separados por al menos dos espacios de la sentencia y deben iniciar con un # (numeral) y un solo espacio.

Evite los comentarios innecesarios, ya que distraen al expresar lo obvio.

**Sí**

x = x + 1 *# Compensación del borde*

**No**

x = x + 1 *# Incremento de x*

## Cadenas de documentación

Una cadena de documentación (docstrings) es una cadena de descripción que se produce como la primera sentencia en un módulo, función, clase o definición de método. Dicha docstring se convertirá en el atributo especial \_\_doc\_\_ de ese objeto.

Normalmente todos los módulos, todas las funciones y clases exportadas por un módulo deberían tener docstrings. Los métodos públicos (incluso el constructor \_\_init\_\_) también deben tener docstrings. Se puede documentar un paquete en el docstring del módulo en el archivo \_\_init\_\_.py en el directorio del paquete.

Los docstrings que aparecen en otras partes del código también pueden actuar como documentación. No son reconocidos por el compilador de bytecode de Python y no son accesibles como atributos de objetos de tiempo de ejecución, pero las herramientas de software pueden extraer dos tipos de docstrings adicionales:

1. Los docstrings que se producen inmediatamente después de una asignación simple en el nivel superior de un módulo, clase o método \_\_init\_\_ se denominan *docstrings de atributos*.
2. Los docstrings que se producen inmediatamente después de otro docstring se denominan *docstrings adicionales*.

Para mantener la coherencia, siempre use triple comillas dobles alrededor de los docstrings. Utilice r”””docstring content””” si utiliza algún backslash en el contenido del docstring. Para los docstrings Unicode use u”””docstring content”””.

### Cadenas de documentación de una sola línea

Los docstrings de una sola línea son para casos obvios que deberían caber en una única línea. Por ejemplo:

**def** kos\_root():

*"""Return the pathname of the KOS root directory."""*

**global** \_kos\_root

**if** \_kos\_root: **return** \_kos\_root

...

Observe que:

* Las triples comillas dobles se utilizan aunque la cadena encaja en una línea. Esto facilita su posterior expansión.
* Las comillas de cierre están en la misma línea que las comillas de apertura.
* No hay líneas en blanco ni antes ni después del docstring.
* Los docstrings son frases que terminan en punto. Describe la función o el efecto resultante de la función de forma imperativa (“Haz esto”, “Retorna esto”), no como una descripción; por ejemplo. No escriba “Retorna el nombre de ruta…”.
* Los docstrings de una sola línea no deben ser una copia que reitere los parámetros de la función o método. No haga

**def** function(a, b):

*"""function(a, b) -> list"""*

### Cadenas de documentación de varias líneas

Los docstrings de varias líneas consisten en una línea de resumen, al igual que un docstring de una sola línea, seguido de una línea en blanco, seguida de una descripción más detallada. La línea de resumen puede ser utilizada por herramientas de indexación automáticas; es importante que encaje en una sola línea y que esté separada del resto del docstring por una línea en blanco. La línea de resumen puede estar en la línea siguiente de las comillas de apertura. Todo el docstring debe estar indentado igual que las comillas en su primera línea.

Inserte una línea en blanco después de todos los docstring que documentan una clase; en general, los métodos de la clase están separados unos de otros por una sola línea en blanco, y el docstring debe compensar con el primer método por una línea en blanco.

El docstring de una script de consola debe poder utilizarse como su mensaje de “uso”, se muestra cuando se invoca el script con argumentos incorrectos o faltantes (o quizá con la opción “-h” para mostrar la ayuda). Dicho docstring debe documentar la función del script y la sintaxis de la línea de comandos, las variables de entorno y los archivos. Los mensajes de uso pueden ser bastante complicados y deberían ser suficientes para que un nuevo usuario use el script correctamente, así como una referencia rápida completa de todas las opciones y argumentos para el usuario sofisticado.

El docstring para un módulo generalmente debe enumerar las clases, excepciones y funciones (y cualquier otro objeto) que el módulo exporte, con un resumen de una línea cada uno. Estos resúmenes generalmente proporcionan menos detalles que la línea de resumen en el docstring del objeto. El docstring para un paquete (es decir, el docstring del módulo \_\_init\_\_.py del paquete) también debe enumerar los módulos y subpaquetes exportados por el paquete.

El docstring para una función o método debe resumir su comportamiento y documentar sus argumentos, valores de retorno, efectos secundarios, excepciones planteadas y restricciones sobre cuando se puede llamar (todo si es necesario). Deben indicarse argumentos opcionales. También se debe documentar si los argumentos clave son parte de la interfaz.

El docstring para una clase debe resumir su comportamiento y enumerar los métodos públicos y las variables de instancia. Si la clase está diseñada para ser una implementada por subclases y tiene una interfaz adicional para dichas subclases, esta interfaz debe aparecer por separado (en el docstring). El constructor de la clase debe documentarse. Los métodos individuales deben ser documentados por su propia docstring.

Si una clase implementa la interfaz de otra clase (hereda) y su comportamiento se hereda principalmente de esa clase, su docstring debe mencionar esto y resumir las diferencias. Use el verbo “sobrescribir” para indicar que un método de la subclase reemplaza un método de la superclase y no llama al método de la superclase; use el verbo “extender” para indicar que un método de la subclase llama al método de la superclase (además de su propio comportamiento).

No utilice la convención de Emacs para mencionar los argumentos de funciones o métodos en mayúsculas en el texto del docstring. Python distingue entre mayúsculas y minúsculas y los nombres de los argumentos se pueden usar para los argumentos de palabra clave, por lo que el docstring debe documentar los nombres de los argumentos correctos. Cada argumento debe ser enumerado en una línea separada. Por ejemplo:

*"""*¶

*Ejemplo completo de documentación de un módulo.*¶

¶

*Este módulo muestra la forma de documentar especificada por la 'Guía*¶

*de estilo* *para codificación en Python' de Namtrik Development. Las*¶

*cadenas de* *documentación pueden extenderse sobre varias líneas. Se*¶

*pueden crear secciones* *con un encabezado para la sección y dos puntos*¶

*seguidos por un bloque de texto* *indentado.*¶

¶

*Ejemplo:*¶

*Esta es una sección dentro del docstring.*¶

¶

*Los saltos de sección se crean al reanudar el texto sin indentación.*¶

*Los saltos de sección también se crean implícitamente cada vez que*¶

*inicia* *una nueva sección.*¶

¶

*Atributos:*¶

*module\_level\_variable1 (int) - Las variables del módulo deben ser*¶

*nombradas en el docstring del módulo y opcionalmente pueden*¶

*ser* *documentadas en una sola línea inmediatamente luego de la*¶

*variable.*¶

*"""*¶

¶

module\_level\_variable1 = 12345¶

¶

module\_level\_variable2 = 98765¶

*"""int - Línea de documentación de la variable del módulo."""*¶

¶

¶

**def** function\_with\_types\_in\_docstring(param1, param2):¶

*"""*¶

*Ejemplo de función con tipos documentados en el docstring.*¶

¶

*Argumentos:*¶

*param1 (int) - El primer parámetro.*¶

*param2 (str) - El segundo parámetro.*¶

*Retorna:*¶

*bool - El valor de retorno. True si es exitosa, False si no.*¶

*"""*¶

¶

¶

**def** function\_with\_pep484\_type\_annotations(param1: int,¶

param2: str) -> bool:¶

*"""*¶

*Otro ejemplo de función con tipos en el encabezado.*¶

¶

*Argumentos:*¶

*param1 - El primer parámetro.*¶

*param2 - El segundo parámetro.*¶

¶

*Retorna:*¶

*El valor de retorno. True si es exitosa, False si no.*¶

*"""*¶

¶

¶

**def** module\_level\_function(param1, param2=None, \*args, \*\*kwargs):¶

*"""*¶

*Ejemplo de una función de módulo.*¶

¶

*Los argumentos deben estar documentados en la sección de*¶

*'Argumentos'. Los* *nombres de cada parámetro son obligatorios. El*¶

*tipo y descripción son* *opcionales, pero deben ser incluidos si no*¶

*son obvios.*¶

¶

*El formato para un parámetro es:*¶

*nombre (tipo) - descripción*¶

*La descripción puede ocupar varias líneas. Las siguientes*¶

*Líneas* *deben indentarse. El tipo de dato es opcional.*¶

¶

*También se permiten varios párrafos en la descripción.*¶

¶

*Argumentos:*¶

*param1 (int) - El primer parámetro.*¶

*param2 (:obj:`str`, opcional) - El segundo parámetro. Por*¶

*defecto None. La segunda línea de la descripción debe ser*¶

*indentada.*¶

*\*args - Lista de argumentos de longitud variable.*¶

*\*\*kwargs - Argumentos arbitrarios de palabras clave.*¶

¶

*Retorna:*¶

*bool - True si es exitosa, sino False.*¶

*El tipo de retorno es opcional y puede ser especificado al*¶

*inicio de* *la sección 'Retorna' seguido por un guion.*¶

¶

*La sección 'Retorna' puede tener varias líneas y párrafos.*¶

*Las siguientes líneas deben estar indentadas de igual forma*¶

*que la* *primera.*¶

¶

*Se permite describir la estructura si el tipo de retorno tiene*

*una* *estructura compleja:*¶

*{*¶

*'param1': param1,*¶

*'param2': param2*¶

*}*¶

¶

*Excepciones:* ¶

*AttributeError - La sección 'Excepciones' es una lista de*¶

*todas las* *excepciones que se disparan en la interface.*¶

*ValueError - Si 'param2' es igual a 'param1'.*¶

*"""*¶

**if** param1 == param2:¶

**raise** **ValueError**('param1 may not be equal to param2')¶

**return** True¶

¶

¶

**def** example\_generator(n):¶

*"""*¶

*Los generadores tienen una sección 'Produce' en vez de 'Retorna'.*¶

¶

*Argumentos:*¶

*n (int) - El límite superior del rango a generar, desde 0 to*¶

*`n` - 1.*

¶

*Produce:*¶

*int - El próximo número en el rango de 0 to `n` - 1.*¶

¶

*Ejemplo:*¶

*Se permite la escritura de ejemplos si la función es compleja*¶

*para entender. Los ejemplos deben ilustrar el uso de la*¶

*función.*¶

¶

*>>> print([i for i in example\_generator(4)])*¶

*[0, 1, 2, 3]*¶

*"""*¶

**for** i **in** range(n):¶

**yield** i¶

¶

¶

**class** **ExampleError**(**Exception**):¶

*"""*¶

*Las Excepciones deben documentarse similar a las clases.*¶

¶

*Atributos:*¶

*msg (str) - Mensaje que describe la excepción. Debe ser*¶

*legible por* *las personas.*¶

*code (int) - Código de error de la excepción.*¶

*"""*¶

¶

**def** \_\_init\_\_(self, msg, code):¶

*"""*¶

*Ejemplo de docstring para un constructor.*¶

¶

*Nota:*¶

*No incluya el parámetro 'self' en la sección de*¶

*argumentos.*¶

¶

*Argumentos:*¶

*msg (str) - Mensaje que describe la excepción. Debe ser*¶

*legible por* *las personas.*¶

*code (:obj:`int`, opcional): Código de error.*¶

*"""*¶

self.msg = msg¶

self.code = code¶

¶

¶

**class** **ExampleClass**(object):¶

*"""*¶

*El docstring de resumen debe alcanzar en una sola línea.*¶

¶

*Los atributos públicos deben ser documentados en la sección*¶

*'Atributos'* *y seguir el mismo formato de la sección 'Argumentos'*¶

*de una función.* *Los atributos también pueden ser comentados en la*¶

*misma línea donde son* *declarados (vea el método \_\_init\_\_).*¶

¶

*Las propiedades creadas con el decorador '@property' deben ser*¶

*documentadas en el método 'get' de la propiedad.* ¶

¶

*Atributos:*¶

*attr1 (str) - Descripción del 'attr1'.*¶

*attr2 (:obj:`int`, opcional) - Descripción del 'attr2'.*¶

*"""*¶

¶

**def** \_\_init\_\_(self, param1, param2, param3):¶

*"""*¶

*Ejemplo de docstring para un constructor \_\_init\_\_.*¶

¶

*Nota:*¶

*No incluya el parámetro 'self' en la sección de*¶

*argumentos.*¶

¶

*Argumentos:*¶

*param1 (str) - Descripción del `param1`.*¶

*param2 (:obj:`int`, opcional) - Descripción del `param2`.*¶

*Se* *permiten varias líneas.*¶

*param3 (:obj:`list` de :obj:`str`) - Descripción del*¶

*`param3`.*¶

*"""*¶

self.attr1 = param1¶

self.attr2 = param2¶

self.attr3 = param3 *# Comentario en línea*¶

¶

@property¶

**def** readonly\_property(self):¶

*"""str: Las propiedades se documentan en su método 'get'."""*¶

**return** 'readonly\_property'¶

¶

@property¶

**def** readwrite\_property(self):¶

*""":obj:`list`: Propiedades con 'get' y 'set'* *se documentan solamente en el método 'get'.*¶

¶

*Si el método 'set' contiene un comportamiento particular debe*¶

*ser* *mencionado aquí.*¶

*"""*¶

**return** ['readwrite\_property']¶

¶

**def** readwrite\_property(self, value):¶

value¶

¶

**def** example\_method(self, param1, param2):¶

*"""*¶

*Los métodos de clase son similares a las funciones regulares.*¶

¶

*Nota:*¶

*No incluya el parámetro 'self' en la sección de*¶

*'Argumentos'.*¶

¶

*Argumentos:*¶

*param1: El primer parámetro.*¶

*param2: El segundo parámetro.*¶

¶

*Retorna:*¶

*True si todo sale bien, de otra forma False.*¶

*"""*¶

**return** True¶

¶

**def** \_\_special\_\_(self):¶

*"""Método especial."""*¶

**pass**¶

¶

**def** \_\_special\_without\_docstring\_\_(self):¶

**pass**¶

¶

**def** \_private(self):¶

*"""Método privado."""*¶

**pass**¶

¶

**def** \_private\_without\_docstring(self):¶

**pass**¶

Se usó el antígrafo (¶) para indicar el final de cada línea.

En docstrings de documentación de archivos de prueba indique el comportamiento esperado que demuestra cada prueba. No incluya preámbulos como “Pruebas que” o “Asegúrese que”.

# Reglas de nombrado

Se deben seguir los estándares de nombrado aquí mostrados para todos los proyectos desarrollados con Python en Namtrik Development. Los módulos o paquetes de terceros (incluyendo los frameworks) en los que se implemente un estilo diferente de nombrado deben de mantenerse con el estilo de origen.

## Estilos de nombramiento

Hay muchos estilos de nombramiento diferentes, cada uno se usa para algo diferente.

* b (una sola letra en minúscula)
* B (Una sola letra en mayúscula)
* lowercase (minúscula)
* lower\_case\_with\_underscores (minusculas con guiones bajos)
* UPPERCASE (mayúsculas)
* UPPER\_CASE\_WITH\_UNDERSCORES (mayúsculas con guines bajos)
* CapitalizedWords (también llamado CapWords o CamelCase)

Nota: Cuando se usa CapWords se debe poner en mayúsculas todas las letras de un acrónimo. Por ejemplo, HTTPServerError.

* mixedCase (la diferencia con CapWords es que inicia con minúsculas)
* Capitalized\_Words\_With\_Underscores (prohibido su uso)

Los prefijos para identificar a un grupo de funciones están prohibidos (por ejemplo, os.star()), ya que tanto los métodos como los atributos están precedidos con un objeto, al igual que los nombres de funciones lo están con el nombre del módulo.

También se deben nombrar los estilos precedidos por un guion bajo en combinación con cualquiera de los estilos anteriormente nombrados:

* \_single\_leading\_underscore: (prefijo de un solo guion bajo) indicador de uso interno. Por ejemplo, **from M import** \* no importa los objetos cuyos nombres inician con un guión bajo.
* single\_trailing\_underscore\_: (sufijo de un solo guion bajo) se usa para evitar conflictos con palabras reservadas de Python. Por ejemplo, Tkinter.Toplevel(master, class\_=’ClassName’).
* \_\_double\_leading\_underscore: (prefijo de doble guion bajo) al ser usado para nombrar el atributo de una clase, se invoca el decorado de nombres.
* \_\_double\_leading\_and\_trailing\_underscore\_\_: (doble guion bajo como prefijo y sufijo) usado para objetos o atributos que viven en nombres de espacio controlados por el usuario. Por ejemplo, \_\_init\_\_, \_\_import\_\_ o \_\_file\_\_. Está prohibido su uso.

## Normas de nombrado de Namtrik Development

### Nombres a evitar

Nunca use los caracteres “l” (letra ele en minúscula), “O” (letra o mayúscula) o “I” (letra i mayúscula) como simples caracteres para nombres de variables ya estos caracteres son indistinguibles en algunos tipos de fuentes.

### Nombres de paquetes y módulos

Los módulos deben tener un nombre corto en minúscula y guiones bajos para mejorar la legibilidad. Ya que los nombres de los módulos están ligados a los de los archivos, es importante que sean bastante cortos.

Los paquetes deben tener un nombre corto y en minúscula, use guiones bajos solamente si es necesario.

### Nombres de clases

Los nombres de clase deben escribirse usando CapWords.

### Nombres de tipos de variables

Los nombres de los tipos de variables deben escribirse usando CapWords prefiriendo nombres cortos: T, AnyStr, Num. Se recomienda agregar los sufijos \_co o \_contra a las variables utilizadas para declarar el comportamiento co-variante o contra-variante de manera correspondiente:

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

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

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

### Nombres de excepciones

Las excepciones deben ser nombradas de la misma forma que una clase. Sin embargo, se debe usar el sufijo “Error” en los nombres de las excepciones (si la excepción realmente corresponde a un error).

### Nombres de variables globales

Las variables globales únicamente deben ser usadas dentro de un módulo. El nombre es muy similar al de 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 (lo que se trata es de indicar que dichas variables globales no son públicas).

### Nombres de funciones y variables

Los nombres de las funciones deben estar escritos en minúscula, con palabras separadas por guiones bajos. Los nombres de variables deben seguir esta misma regla.

Solamente se permite el uso de otro estilo en contextos donde ya es el estilo predominante (por ejemplo, el llamado a un módulo o paquete de terceros).

### Argumentos de funciones y métodos

Siempre use **self** como el primer argumento de los métodos. Si el nombre de un argumento causa conflicto con una palabra clave reservada, se recomienda agregar un guion bajo como sufijo o usar un sinónimo.

### Nombres de métodos y atributos de una clase

Al igual que las funciones, los métodos y atributos de una clase deben escribirse con minúsculas con palabras separadas por guiones bajos.

Use un guion bajo solo para métodos y variables privados.

Para evitar conflictos de nombres con las subclases, use dos guiones bajos para invocar el *name mangling* de Python. Python maneja estos nombres con el nombre de la clase: si la clase se llama Foo y tiene un atributo llamado \_\_a, Foo.\_\_a no puede acceder a él. Generalmente los guiones bajos dobles se deben usar solo para evitar conflictos de nombres con los atributos en clases diseñadas para ser heredadas.

### Constantes

Las constantes deben escribirse en mayúsculas con guiones bajos que separen las palabras.

### Diseñando para herencia

Tenga en cuenta las siguientes normal al crear clases que serán heredadas:

* Los atributos públicos no deben tener un guion bajo como prefijo.
* Si un atributo público provoca problemas con una palabra reservada agregue un guion bajo como sufijo al nombre.
* Para atributos públicos de datos simples, es mejor exponer únicamente el nombre del atributo, sin complicados métodos de acceso (getters o setters). Si un atributo simple necesita tener un comportamiento diferente cuando este cambie, en ese caso use propiedades para ocultar la implementación funcional.

Nota 1: se prohíbe implementar efectos secundarios en las propiedades que ejecuten operaciones muy pesadas.

* Si una clase está destinada a ser utilizada como una subclase y tiene atributos que no quiere que dicha clase use, considere nombrarlos usando dos guiones bajos como prefijo y sin sufijos. Esto invocará el name mangling, en 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.

# Recomendaciones

* El código fuente debe estar escrito de tal manera que no provoque conflictos con otras implementaciones de Python (Pypy, Jython, IronPython, Cython, Psyco, etc.).

Por ejemplo, no se confíe de la eficiencia para la concatenación de cadenas de CPython para sentencias del tipo a += b o a = a + b. Esta optimización es frágil incluso en CPython y no se encuentra presente en todas las implementaciones. En esos casos la forma “”.join() debe ser utilizada en lugar de la anterior. Esto asegura que la concatenación ocurra en tiempo lineal en las diversas implementaciones.

* Las comparaciones con None deben realizarse siempre con **is** o **is not**, nunca con los operadores de igualdad (== o ===).
* Use el operador **is not** en vez de **not** … **is**, esto facilita la legibilidad del código.
* Siempre use una sentencia **def** en lugar de una sentencia de asignación que liga la expresión lambda directamente a un nombre:

**Sí**

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

**No**

f = **lambda** x: 2\*x

* Derive las excepciones desde Exception y no desde BaseException. La herencia desde BaseException es reservada para excepciones en donde capturarlas es casi siempre el método incorrecto de hacerlo.

Apunte siempre a responder la pregunta “¿Qué salió mal?” antes que únicamente afirmar que “Ha ocurrido un error”.

Recuerde que la regla de nombrado de clases se aplica a las excepciones, aunque se debe añadir el sufijo “Error” si se trata de una excepción de error. Las excepciones que no correspondan a un error no necesitan el sufijo especial.

* Al momento de disparar una excepción use la forma **raise** **ValueError**(‘message’). Los paréntesis le permitirán ajustar los argumentos si son muy largos.
* Al momento de capturar excepciones mencione sus nombres de forma específica en lugar de solo usar la clausula **except**; por ejemplo:

**try**:

**import** **platform\_specific\_module**

**except** **ImportError**:

platform\_specific\_module = None

Si quiere capturar todas las excepciones que señalizan errores del programa, utilice **except Exception**, en vez de solamente **except**.

* Al momento de asignar una excepxión a un nombre use la siguiente sintaxis:

**try**:

process\_data()

**except** **Exception** **as** exc:

**raise** DataProcessingFailedError(str(exc))

* Todas las cláusulas **try** deben estar limitadas al mínimo absoluto de líneas de código necesarias.

**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 el error generado por handle\_value()*

**return** key\_not\_found(key)

* Cuando un recurso es local para una sección de código en particular, utilice la sentencia **with** para asegurarse que es limpiada inmediatamente luego del uso.
* Los manejadores de contexto deben ser invocados desde funciones separadas siempre y cuando hagan 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 ultimo ejemplo no proporciona ninguna información para indicar que los métodos \_\_enter\_\_ y \_\_exit\_\_ están haciendo algo más que cerrar la conexión luego de la transacción. Se debe ser explícito.

* Sea consistente en las sentencias de retorno. Si todas las sentencias de retorno devuelve una expresión, cualquier sentencia de retorno en la que no se devuelva ningún valor debe declararse explícitamente como retorno nulo.

**Sí**

**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)

**No**

**def** foo(x):

**if** x >= 0:

**return** math.sqrt(x)

**def** bar(x):

**if** x < 0:

**return**

**return** math.sqrt(x)

* Use los métodos de manejo de cadenas en lugar del módulo string. Los métodos de manejo de cadenas son siempre más rápidos y comparten la misma API con las cadenas Unicode.
* Use ‘’.startswith() y ‘’.endswith() en lugar de aplicar string slicing para comprobar prefijos o sufijos.

**Sí**

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

**No**

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

* Al realizar comparaciones de tipos 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 comprobar si un objeto es una cadena, tenga en cuenta que se puede tratar de una cadena Unicode. En Python 2 str y unicode tienen una clase base en común, basestring, por lo que debe hacer:

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

En Python 3, unicode y basestring ya no existen (únicamente str) y un objeto de bytes no es más que un tipo de cadena.

* Para secuencias (cadenas, listas, tuplas), use el hecho que aquellas que se encuentran vacías son falsas:

**Sí**

**if** **not** seq:

**if** seq:

**No**

**if** len(seq):

**if** **not** len(seq):

* No escriba cadenas que se basen en un importante espacio al final. Dichos espacios son visualmente indistinguibles y algunos editores los eliminarán.
* No compare valores booleanos con True o False usando ==.

**Sí**

**if** greeting:

**No**

**if** greeting == True:

**Peor**

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

# Recomendaciones para codificación en Django

## Plantillas

En el código de las plantillas de Django, coloque un solo espacio entre los corchetes y el contenido de la etiqueta.

**Sí**

{{ foo }}

**No**

{{foo}}

## Vistas

En las vistas de Django, el primer argumento en una función de vista debe ser llamado request.

**Sí**

**def** my\_view(request, foo):

**No**

**def** my\_view(req, foo):

## Modelos

Los nombres de los campos deben escribirse en minúscula, usando guiones bajos en lugar de CamelCase.

**Sí**

**class** **Person**(models.Model):

first\_name = models.CharField(max\_length=20)

last\_name = models.CharField(max\_length=40)

**No**

**class** **Person**(models.Model):

FirstName = models.CharField(max\_length=20)

Last\_Name = models.CharField(max\_length=40)

La subclase Meta debe aparecer después de que los campos hayan sido definidos, con una sola línea en blanco que separa los campos y la definición de la clase.

**Sí**

**class** **Person**(models.Model):

first\_name = models.CharField(max\_length=20)

last\_name = models.CharField(max\_length=40)

**class** **Meta**:

verbose\_name\_plural = 'people'

**No**

**class** **Person**(models.Model):

first\_name = models.CharField(max\_length=20)

last\_name = models.CharField(max\_length=40)

**class** **Meta**:

verbose\_name\_plural = 'people'

**class** **Person**(models.Model):

**class** **Meta**:

verbose\_name\_plural = 'people'

first\_name = models.CharField(max\_length=20)

last\_name = models.CharField(max\_length=40)

El orden de las clases internas del modelo y los métodos estándar deben ser los siguientes (tenga en cuenta que no todos son necesarios):

* Todos los campos de la base de datos.
* Atributos personalizados del administrador.
* Subclase Meta.
* def \_\_str\_\_()
* def save()
* def get\_absolute\_url()
* Cualquier otro método.

Si se definen opciones para un determinado campo del modelo, defina cada opción como una tupla de tuplas, con un nombre en mayúsculas como atributo de clase dentro del modelo. Por ejemplo:

**class** **MyModel**(models.Model):

DIRECTION\_UP = 'U'

DIRECTION\_DOWN = 'D'

DIRECTION\_CHOICES = (

(DIRECTION\_UP, 'Up'),

(DIRECTION\_DOWN, 'Down'),

)

## Misceláneos

* Marque todas las cadenas para internacionalización. Revise la [documentación de i18n](https://docs.djangoproject.com/en/2.1/topics/i18n/) para más detalles.
* Elimine las sentencias de importación que ya no se utilizan cuando el código fuente cambie.
* Elimine sistemáticamente todos los espacios en blanco finales del código fuente.
* No ponga su nombre en el código que construye. La política de Namtrik Development es mantener los nombres de los integrantes del equipo en el archivo “README.md” del repositorio remoto, no dispersos en toda la base del código.

# Herramientas

|  |  |  |
| --- | --- | --- |
| Nombre | Descripción | Enlace |
| Visual Studio Code | Editor de texto gratuito con soporte para resaltado de código y plugins de terceros. Disponible para Linux, Mac y Windows. | [code.visualstudio.com](https://code.visualstudio.com/) |
| Sublime Text 3 | Editor de texto gratuito con soporte para resaltado de código y plugins de terceros. Disponible para Linux, Mac y Windows. | [sublimetext.com](https://www.sublimetext.com/) |
| Pylint | Inspector de calidad de código fuente y bugs para Python que implementa la guía de estilos PEP 8. | [pylint.org](https://www.pylint.org/) |
| Pipenv | Gestor de paquetes y entornos virtuales para proyectos en Python. | [docs.pipenv.org](https://docs.pipenv.org/) |
| isort | Organizador de importaciones para Python. | [bit.ly/2zX7gVh](https://bit.ly/2zX7gVh) |
| EditorConfig | Archivo que ayuda a definir y mantener la consistencia de estilos de codificación entre diferentes editores de texto, IDE’s o proyectos. | [editorconfig.org](https://editorconfig.org/) |

# Glosario de términos

**API:** conjunto de subrutinas, funciones y procedimientos que ofrece un software para ser utilizado por otro.

**Argumento:** variable utilizada para recibir valores de entrada en una rutina, subrutina o método.

**Barra invertida (Backslash):** carácter “\”.

**Booleano:** tipo de dato lógico que puede tomar el valor de verdadero o falso.

**Cadena (de caracteres):** secuencia ordenada de letras, de longitud variable, aunque, finita.

**Comentario:** composición destinada a agregar anotaciones dentro del código fuente.

**Comodín:** símbolo que sustituye a cualquier carácter o grupo de caracteres.

**Decorado de nombres (name mangling):** técnica usada por un compilador para resolver conflictos en los nombres de entidades. En Python el name mangling consiste en agregar como prefijo el nombre de la clase a un atributo que haya sido marcado para ser decorado.

**Diff:** herramienta de comparación de archivos, comúnmente utilizada en la gestión de repositorios.

**Excepción:** error de ejecución, comportamiento inesperado o fallo de un software.

**Getters:** métodos de una clase que permiten acceder a una de sus propiedades o atributos.

**Indentación:** espaciado de un bloque de texto hacia la derecha insertando espacios o tabulaciones, para separarlo del margen izquierdo.

**Librería:** conjunto de implementaciones funcionales que ofrecen una interfaz para ser usada por otro software.

**Manejadores de contexto (context managers):** palabras del lenguaje Python que encapsulan rutinas, generalmente de limpieza o liberación de recursos al terminar un bloque de código.

**Módulos:** parte autónoma de un software. En Python, los módulos corresponden a los archivos que incluyen diferentes rutinas, clases o variables.

**None:** objeto del lenguaje Python que denota falta de valor.

**Paquete:** serie de programas que se distribuyen de forma conjunta. En Python, los paquetes corresponden a los directorios que contienen diferentes módulos y que tienen una estructura específica.

**Script:** archive de rutinas o conjunto de rutinas de programación que ejecutan una operación.

**Setters:** métodos de una clase que permiten establecer el valor a una de sus propiedades o atributos.

**Slice/Slicing:** forma en la que Python proporciona el acceso a un rango específico de elementos de una secuencia (cadena, lista o tupla).

**String slicing:** forma en la que Python proporciona el acceso a un rango específico de caracteres de una cadena de caracteres.

**Unicode:** estándar de codificación de caracteres.

# Referencias

|  |  |
| --- | --- |
| Clave | Referencia |
| PEP 8 | PEP 8 – Style Guide for Python Code. Guido van Rossum, Barry Warsaw, Nick Coghlan - [bit.ly/2PB3pYZ](https://bit.ly/2PB3pYZ) |
| PEP 257 | PEP 257 – Docstring Conventions. David Goodget, Guido van Rossum - [bit.ly/1BmKBQv](https://bit.ly/1BmKBQv) |
| Django Coding Style | Coding style. Equipo de Django Framework - [bit.ly/2QJc0VM](https://bit.ly/2QJc0VM) |