12_PEP

June 18, 2025

Contenido creado por:

Isabel Maniega

---

[1]:
```
import warnings
warnings.filterwarnings('ignore')
```

---

1 Explicar e implementar las mejores prácticas de programación de scripts de Python

1. Introducción a PEP 8
2. PEP 257 – Convenciones de cadenas de documentos

Aprenderas sobre:

- PEP 20 (El zen de Python)
- PEP 8 (Guía de estilo para código Python)
- PEP 257 (Convenciones de cadenas de documentos)
- Cómo evitar errores comunes al escribir código;
- Cómo escribir código elegante y efectivo.

¿Qué es PEP?

PEP se refiere al documento en línea, que describe los estándares del lenguaje y proporciona información sobre muchos cambios y procesos relacionados con Python.

En la jungla de los PEP

Hay muchos PEP, cientos de ellos. Vea PEP 0: Índice de propuestas de mejora de Python (PEP) para descubrir por sí mismo que no son palabras ociosas.

Sería notable, pero desafortunadamente un gran desafío, cubrirlos todos en este curso. Por esta razón, hemos elegido cuatro de ellos que merecen un análisis más detallado y deben considerarse lecturas obligatorias. Estos son:

- PEP 1 – Propósito y pautas de los PEP, que proporciona información sobre el propósito de los PEP, sus tipos y presenta pautas generales;

- PEP 8 – Guía de estilo para código Python, que proporciona convenciones y presenta las mejores prácticas para la codificación Python;
- PEP 20 – El zen de Python, que presenta una lista de principios para el diseño de Python;
- PEP 257 – Convenciones de cadenas de documentación, que proporciona pautas para las convenciones y la semántica asociadas con las cadenas de documentación de Python.

Te animamos a que te sumerjas en los PEP por tu cuenta. Estamos seguros de que te resultará cada vez más curioso a medida que tu experiencia y conocimiento de programación aumenten.

1.1 Introducción a PEP 8

Como se mencionó anteriormente, PEP 8 es un documento que proporciona convenciones de codificación (guía de estilo de código) para el código Python.

PEP 8 se considera uno de los PEP más importantes y una lectura obligada para todo programador profesional de Python, ya que ayuda a que el código sea más consistente, más legible y más eficiente.

Aunque algunos proyectos de programación pueden adoptar sus propias pautas de estilo (en cuyo caso, dichas pautas específicas del proyecto pueden preferirse a las convenciones proporcionadas por PEP 8, especialmente en caso de conflictos o problemas de compatibilidad con versiones anteriores), las mejores prácticas de PEP 8 siguen siendo una lectura muy recomendable, ya que te ayudan a comprender mejor la filosofía detrás de Python y a convertirte en un programador más consciente y competente.

PEP 8 sigue evolucionando, ya que se están identificando e incluyendo nuevas convenciones adicionales y, al mismo tiempo, se están identificando algunas convenciones antiguas como obsoletas y se desaconseja su seguimiento.

El duende de las mentes pequeñas

"Una coherencia tonta es el duende de las mentes pequeñas". Esta es una cita del ensayo de Ralph Waldo Emerson "Autosuficiencia", donde Emerson insta a los lectores a ser coherentes en sus creencias y prácticas. En nuestro caso, significa que no debemos olvidarnos de una observación simple pero importante: **nuestro código será leído mucho más a menudo de lo que será escrito.**

Por un lado, la coherencia es un factor crucial que determina la legibilidad del código. Por otro lado, la incoherencia con PEP 8 puede ser a veces una mejor opción. Si las guías de estilo no son aplicables a su proyecto, puede ser mejor ignorarlas y decidir por sí mismo qué es lo mejor. Como dice PEP 8:

Una guía de estilo se trata de coherencia. La coherencia con esta guía de estilo es importante. La coherencia dentro de un proyecto es más importante. La coherencia dentro de un módulo o función es lo más importante. […] Sin embargo, sepa cuándo ser incoherente […]. En caso de duda, utilice su mejor criterio.

¿Cuándo debería ignorar algunas pautas específicas de PEP 8 (o al menos considerar hacerlo)?

- Si seguirlas significará romper la compatibilidad con versiones anteriores.
- Si seguirlas tendrá un efecto negativo en la legibilidad del código.
- Si seguirlas causará incoherencia con el resto del código. (Sin embargo, esta puede ser una buena oportunidad para reescribir el código y hacerlo compatible con PEP 8).

- Si no hay una buena razón para hacer que el código sea compatible con PEP 8, o el código es anterior a PEP 8.

PEP 8 tiene como objetivo mejorar la legibilidad del código y "hacerlo consistente en todo el espectro del código Python". Por lo tanto, es una buena idea mantener su código Python compatible con PEP 8, pero nunca debe adherirse ciegamente a estas recomendaciones. Siempre debe utilizar su mejor criterio.

Comprobadores de compatibilidad con PEP 8

Existen muchas herramientas útiles que pueden ayudarlo a validar el estilo de su código y verificarlo con las convenciones de estilo de PEP 8. Estas herramientas se pueden instalar y ejecutar localmente, o se puede acceder a ellas en línea. Queremos mostrarle solo dos de ellas, pero lo alentamos a que explore más por su cuenta:

- pycodestyle (antes llamado pep8, pero se cambió el nombre para evitar confusiones) - Comprobador de guía de estilo de Python; Le permite verificar su código Python para verificar su conformidad con las convenciones de estilo en PEP 8. Puede instalar la herramienta con el siguiente comando en la terminal:

```
$ pip install pycodestyle
```

Puede ejecutarla en un archivo o archivos para obtener información sobre la no conformidad (e indicar errores en el código fuente y su frecuencia).

Más información: https://github.com/PyCQA/pycodestyle Documentación: https://pycodestyle.pycqa.org/en/latest/

- También puede instalar **autopep8** para formatear automáticamente su código Python para que cumpla con las pautas de PEP 8. Para poder usarlo, necesita la instalación de pycodestyle en su máquina para indicar las partes del código que requieren correcciones de formato.

Más información: https://pypi.org/project/autopep8/

- **PEP 8 online** es un verificador de PEP 8 online creado por Valentin Bryukhanov que te permite pegar tu código o subir un archivo y validarlo con las pautas de estilo de PEP 8. La herramienta online está creada con Flask, Twitter Bootstrap y el módulo PEP8 (el mismo módulo que acabamos de describir).

Más información: http://pep8online.com/about

Recomendaciones para el diseño del código

PEP 8 está pensado para mejorar tu experiencia de codificación y hacer tu vida mucho más sencilla. Como se dijo antes, la forma en que escribes tu código tiene un gran impacto en su legibilidad. Sin embargo, no debes olvidar que también puede determinar su legalidad sintáctica.

En esta sección, nos centraremos en las recomendaciones de estilo relacionadas con aspectos como:

- sangría, uso de tabulaciones y espacios;
- longitud de línea, saltos de línea y líneas en blanco;
- codificación de archivos fuente e importaciones de módulos.

Sangría El nivel de sangría, entendido como el espacio en blanco inicial (es decir, espacios y tabulaciones) al comienzo de cada línea lógica, se utiliza para agrupar declaraciones.

Al escribir código en Python, debe recordar seguir estas dos reglas simples:

- Use cuatro espacios por nivel de sangría, y;
- Use espacios en lugar de tabulaciones.

Sin embargo, puede usar tabulaciones cuando desee mantener la coherencia con el código que ya ha sido sangrado con tabulaciones (si no es posible o eficiente hacerlo compatible con PEP 8).

Nota: Mezclar tabulaciones y espacios para la sangría no está permitido en Python 3. Esto generará una excepción TabError: "TabError: uso inconsistente de tabulaciones y espacios en la sangría".

Ejemplos:

Incorrect

```
# Bad:

def my_fun_one(x, y):
    return x * y

def my_fun_two(a, b):
  return a + b
```

Correct

```
# Good:

def my_function(x, y):
    return x * y
```

Líneas de continuación

Las líneas de continuación (es decir, líneas lógicas de código que desea dividir porque son demasiado largas o porque desea mejorar la legibilidad) están permitidas si se utilizan paréntesis, corchetes o llaves:

Incorrect

```
# Bad:

my_list_one = [1, 2, 3,
    4, 5, 6
]

a = my_function_name(a, b, c,
    d, e, f)
```

Correct

```
# Good:

my_list_one = [
    1, 2, 3,
    4, 5, 6,
    ]
```

```
a = my_function_name(a, b, c,
                     d, e, f)
```

Correct

```
# Good:

my_list_two = [
    1, 2, 3,
    4, 5, 6,
]


def my_fun(
        a, b, c,
        d, e, f):
    return (a + b + c) * (d + e + f)
```

Puede leer más sobre la sangría en el contexto de las líneas de continuación en https://www.python.org/dev/peps/pep-0008/#indentation.

Longitud máxima de línea y saltos de línea

Si es posible, debe limitar todas las líneas a un máximo de 79 caracteres, ya que esto le ayudará a evitar que se abarquen varias líneas de código. Si el abaratamiento de línea es inevitable, utilice la continuación de línea implícita de Python de la página anterior.

En el caso de cadenas de documentación y comentarios, la longitud de línea no debe superar los 72 caracteres.

```
print("Hello, world!")
```

21 characters in one line

```
print("Python is a very simple language, and has a very straightforward syntax.")
```

81 characters in one line

Por conveniencia, si un equipo (o equipos) que trabajan en un proyecto determinado lo acuerdan, según PEP 8, es posible aumentar la longitud de línea a 99 caracteres (esto no se relaciona con cadenas de documentación/comentarios, cuya longitud de línea debe seguir limitada a 72 caracteres).

Aun así, la biblioteca estándar de Python es conservadora en este asunto y requiere que no se utilicen más de 79 caracteres por línea (72 para comentarios/cadenas de documentación).

Saltos de línea y operadores

Aunque en Python se permite dividir líneas de código antes o después de operadores binarios (siempre que lo haga de manera consistente y que esta convención se haya utilizado en su código antes), se recomienda que siga las sugerencias de estilo de Donald Knuth y realice el corte antes de los operadores binarios, ya que esto da como resultado un código más legible y agradable a la vista.

Ejemplo:

```
# Recommended

total_fruits = (apples
                + pears
                + grapes
                - (black currants - red currants)
                - bananas
                + oranges)
```

Líneas en blanco

Las líneas en blanco, llamadas espacios verticales, mejoran la legibilidad de su código.

Permiten que la persona que lee su código vea la división del código en secciones, lo ayudan a comprender mejor la relación entre las secciones y a captar la lógica de los bloques de código dados con mayor facilidad.

De la misma manera, usar demasiadas líneas en blanco en su código hará que parezca escaso y más difícil de seguir, por lo que siempre debe tener cuidado de no usarlas en exceso.

PEP 8 recomienda que use:

– **dos líneas en blanco** para rodear las definiciones de funciones y clases de nivel superior:

```
class ClassOne:
    pass


Class ClassTwo:
    pass


def my_top_level_function():
    return None
```

– **una sola línea en blanco** para rodear las definiciones de métodos dentro de una clase:

```
class MyClass:
    def method_one(self):
        return None

    def method_two(self):
        return None
```

– **Líneas en blanco** en las funciones para indicar secciones lógicas (con moderación). Por ejemplo:

```
def calculate_average():
    how_many_numbers = int(input("How many numbers? "))

    if how_many_numbers > 0:
        sum_numbers = 0
        for i in range(0, how_many_numbers):
```

```
            number = float(input("Enter a number: "))
            sum_numbers += number

        average = 0
        average = sum_numbers / how_many_numbers

        return average
    else:
        return "Nothing happens."
```

Codificaciones predeterminadas

Se recomienda utilizar las codificaciones predeterminadas de Python (Python 3: UTF-8, Python 2: ASCII). No se recomiendan las codificaciones que no sean las predeterminadas y solo deben utilizarse con fines de prueba o en situaciones en las que los comentarios o las cadenas de documentación utilicen un nombre (por ejemplo, el nombre de un autor) que contenga un carácter que no sea ASCII.

PEP 8 establece que "todos los identificadores en la biblioteca estándar de Python DEBEN usar identificadores que solo sean ASCII y DEBEN usar palabras en inglés siempre que sea posible".

Nota: consulte PEP 3131 (Compatibilidad con identificadores que no sean ASCII) para obtener más información sobre la lógica, así como las ventajas y desventajas de usar identificadores que no sean ASCII.

Importaciones

Siempre debe colocar las importaciones al comienzo de su secuencia de comandos, entre los comentarios/cadenas de documentación del módulo y las constantes y variables globales del módulo, respetando el siguiente orden:

1. Importaciones de la biblioteca estándar;
2. Importaciones de terceros relacionadas;
3. Importaciones específicas de la biblioteca/aplicación local. Asegúrese de insertar una línea en blanco para separar cada uno de los grupos de importaciones anteriores.

PEP 8 recomienda que sus importaciones estén en líneas separadas, en lugar de apretujadas en una sola línea:

```
# Bad:

import sys, os


# Good:

import os
import sys
```

Aún así, es correcto realizar una importación de una línea usando la sintaxis `from … import …`:

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

Si es posible, utilice importaciones absolutas (es decir, importaciones que utilicen rutas absolutas separadas por puntos). Por ejemplo:

```
import animals.mammals.dogs.puppies
```

En Python, se prefieren estas importaciones, especialmente cuando la aplicación no es demasiado grande ni extremadamente compleja.

No debería (y en realidad no puede) utilizar importaciones relativas implícitas, ya que ya no están presentes en Python 3. También debería evitar utilizar importaciones con comodines, por ejemplo:

```
from animals import *
```

ya que inhiben la legibilidad del código y pueden interferir con algunos de los nombres que ya están presentes en el espacio de nombres.

Recomendaciones para comillas de cadena, espacios en blanco y comas finales

En esta sección, nos centraremos en las recomendaciones de estilo relacionadas con aspectos como:

- comillas de cadena;
- espacios en blanco en expresiones y declaraciones, y el uso de comas finales.

Comillas de cadena

Python nos permite usar cadenas entre comillas simples (p. ej., `'una cadena'`) y entre comillas dobles (p. ej., `"una cadena"`). Son lo mismo, y no hay ninguna recomendación especial en PEP que le indique qué estilo debe adoptar al escribir su código. Nuevamente, la regla más importante es: sea coherente con su elección.

Sin embargo, para mejorar la legibilidad, PEP 8 recomienda que intente evitar el uso de barras invertidas (caracteres de escape) en cadenas. Esto significa que:

- si su cadena contiene caracteres entre comillas simples, se recomienda que utilice cadenas entre comillas dobles;
- si su cadena contiene caracteres entre comillas dobles, se recomienda que utilice cadenas entre comillas simples.

En el caso de cadenas entre comillas triples, PEP 8 recomienda que siempre utilice caracteres entre comillas dobles para mantener la coherencia con la convención de cadenas de documentación detallada en PEP 257 (pronto le informaremos más sobre esto).

Espacio en blanco en expresiones y declaraciones

PEP 8 contiene una sección larga que muestra ejemplos de usos correctos e incorrectos de espacios en blanco en el código. En general, debe evitar usar demasiados espacios en blanco, ya que dificultan el seguimiento de su código.

Por lo tanto, por ejemplo, no utilice demasiados espacios en blanco inmediatamente dentro de paréntesis/corchetes/llaves, o inmediatamente antes de una coma/punto y coma/dos puntos:

```
# Bad:

my_list = ( dog[ 2 ] , 5 , { "year": 1980 } , "string" )
if 5 in my_list : print( "Hello!" ) ; print( "Goodbye!" )


# Good:

my_list = (dog[2], 5, {"year": 1980}, "string")
if 5 in my_list: print("Hello!"); print("Goodbye!")
```

Espacio en blanco en expresiones y declaraciones (cont.)

En el caso de una porción, los dos puntos deben tener la misma cantidad de espacio en ambos lados (debe actuar como un operador binario) a menos que se omita un parámetro de porción, en cuyo caso también se debe omitir el espacio.

Ejemplos:

```
# Bad:

bread[0 : 3], roll[1: 3 :5], bun[3: 5:], donut[ 1: :5 ]


# Good:

bread[0:3], roll[1:3:5], bun[3:5:], donut[1::5]
```

Comas finales

Nuevamente, no use demasiados espacios en blanco:

- después de una coma final seguida de un paréntesis de cierre, o
- inmediatamente antes de un paréntesis de apertura que marca el comienzo de la lista de argumentos de una invocación de función, o
- inmediatamente antes de un paréntesis de apertura que marca el comienzo de la indexación/segmentación.

Ejemplos:

```
# Bad:

my_tuple = (0, 1, 2, )
my_function (5)
my_dictionary ['key'] = my_list [index]
```

```
# Good:

my_tuple = (0, 1, 2,)
my_function(5)
my_dictionary['key'] = my_list[index]
```

Espacio en blanco en expresiones y declaraciones (cont.)

No utilice más de un espacio antes y después de los operadores, por ejemplo:

```
# Bad:

a         = 1
b         = a        + 2
my_string = 'string' * 2


# Good:

a = 1
b = a + 2
my_string = 'string' * 2
```

Rodee los operadores binarios con un solo espacio en ambos lados. Sin embargo, si en su código hay operadores que tienen diferentes prioridades, puede considerar agregar espacios solo alrededor de los operadores de menor prioridad, por ejemplo:

```
# Bad:

x=x+3
x -=1

x = x * 2 - 1
x = (x - 1) * (x + 2)


# Good:

x = x + 3
x -= 1

x = x*2 - 1  # Use your own judgement.
x = (x-1) * (x+2)  # Use your own judgement.
```

No rodee el operador = con espacios si se usa para indicar un argumento de palabra clave/valor predeterminado, por ejemplo:

```
# Bad:

def my_function(x, y = 2):
    return x * y


# Good:

def my_function(x, y=2):
    return x * y
```

Recomendaciones para el uso de comentarios

Los comentarios tienen como objetivo mejorar la legibilidad del código sin afectar el resultado del programa. Los buenos programadores documentan su código y explican los fragmentos de código más complejos, de modo que la persona que lea el código comprenda correctamente lo que sucede en el programa. Debe utilizar los comentarios con prudencia y, siempre que sea posible, escribir código que se comente por sí mismo (por ejemplo, dé nombres propios a las variables, funciones y elementos del código).

Hay algunas reglas que debe seguir al dejar comentarios en el código:

- Escriba comentarios que no contradigan el código ni confundan al lector. Son mucho peores que no incluir ningún comentario.
- Actualice sus comentarios cuando se actualice su programa.
- Escriba los comentarios como oraciones completas (escriba con mayúscula la primera palabra si no es un identificador y finalice la oración con un punto). Por ejemplo:

```
# Program that calculates body mass index (BMI).

height = float(input("Your height (in meters): "))
weight = float(input("Your weight (in kilograms): "))
bmi = round(weight / (height*height), 2)

print("Your BMI: {}".format(bmi))
```

- Al escribir comentarios en bloque con comentarios de varias oraciones, use dos espacios después de cada punto que finalice una oración, excepto después de la oración final.
- Escriba los comentarios en inglés (a menos que esté 100% seguro de que el código nunca será leído por personas que no hablen su idioma).
- Los comentarios no deben constar de más de 72 caracteres por línea (pero eso ya lo sabe).

Comentarios en bloque

Los comentarios en bloque suelen ser más largos y debe usarlos para explicar secciones de código en lugar de líneas específicas. Le permiten dejar información para el lector en varias líneas (y varias oraciones). Generalmente, los comentarios en bloque:

- deben hacer referencia al código que los sigue;
- deben tener la misma sangría que el código que describen.

Al escribir comentarios en bloque, comience cada línea con `#` seguido de un solo espacio y separe los párrafos con una línea que contenga solo el símbolo `#`. Por ejemplo:

```
def calculate_product():
    # Calculate the average of three numbers obtained from the user. Then
    # multiply the result by 4.17, and assign it to the product variable.
    #
    # Return the value passed to the product variable and use it
    # for the subsequent x to y calculations to speed up the process.
    sum_numbers = 0

    for number in range(0, 3):
        number = float(input("Enter a number: "))
        sum_numbers += number

    average = (sum_numbers / 3) * 4.17
    product = average
    return product

x = product * 1.73
y = x ** 2
x_to_y = (x*y) / 1.05
```

Comentarios en línea

Los comentarios en línea son comentarios que se escriben en la misma línea que tus declaraciones. Deben referirse a una sola línea de código o una sola declaración o brindar una explicación más detallada de la misma. No debes abusar de ellos.

En general, los comentarios en línea deben estar:

- separados por dos (o más) espacios de la declaración a la que se refieren;
- utilizados con moderación.

Pueden ayudarte a recordar rápidamente lo que hace una línea de código en particular o ser útiles cuando los lea alguien que no esté familiarizado con tu código. Por ejemplo:

```
counter = 0     # Initialize the counter.
```

Sin embargo, no utilices comentarios en línea (ni ningún otro tipo de comentario) para explicar cosas obvias o innecesarias. Por ejemplo:

```
a += 1     # Increment a.
```

Intente siempre hacer que su código se comente por sí mismo en lugar de agregar comentarios, incluso si parecen sensatos o necesarios, por ejemplo:

```
# Bad:

a = 'Adam'  # User's first name.
```

```
# Good:

user_first_name = 'Adam'
```

Cadenas de documentación

Las cadenas de documentación, o docstrings como se las suele llamar, te permiten proporcionar descripciones y explicaciones para todos los módulos, archivos, funciones, clases y métodos públicos que utilizas en tu código. Deberías utilizarlas en este contexto.

Trataremos las docstrings cuando hablemos de PEP 257 más adelante en el curso. Por el momento, solo debes recordar que son un tipo de comentario que comienza y termina con tres comillas dobles: “ “ “.

Ejemplos:

```
# A multi-line docstring:

def fun(x, y):
    """Convert x and y to strings,
    and return a list of strings.
    """
    ...

# A single-line docstring:

def fun(x):
    """Return the square root of x."""
    ...
```

Convenciones de nomenclatura: Introducción

Al programar, a menudo hay que nombrar identificadores y otras entidades en el código. Dar nombres adecuados y evitar los inapropiados sin duda aumentará la legibilidad del código y le ahorrará a usted (y a otros programadores que lean su código) mucho tiempo y esfuerzo.

Seguramente ya sigue algunas convenciones para dar nombres a variables, funciones y clases en su código; algunas de ellas pueden provenir de su experiencia previa en programación en otros lenguajes, otras pueden ser una elección puramente práctica, mientras que otras pueden estar determinadas por los requisitos del proyecto o las prácticas adoptadas por su empresa o equipo.

Lamentablemente, las convenciones de nomenclatura de Python no son completamente consistentes en toda la biblioteca de Python. Sin embargo, se recomienda que los nuevos módulos y paquetes se escriban de conformidad con las recomendaciones de nomenclatura de PEP 8 (a menos que una biblioteca existente siga un estilo diferente, en cuyo caso la coherencia interna es la solución preferida).

Estilos de nombres

Existen muchos estilos de nombres diferentes que se utilizan en programación, por ejemplo:

- `a` – una sola letra minúscula
- `A` – una sola letra mayúscula

En general, debes evitar usar nombres de una sola letra como l (la letra minúscula el), I (la letra mayúscula eye) y O (la letra mayúscula oh), porque pueden confundirse fácilmente con los números 1 y 0, y hacer que tu código sea mucho menos legible.

- `mysamplename` – minúsculas
- `my_sample_name` – minúsculas con guiones bajos (snake_case)
- `MYSAMPLENAME` – mayúsculas
- `MY_SAMPLE_NAME` – mayúsculas con guiones bajos (SNAKE_CASE)
- `MySampleName` – CamelCase (también conocido como palabras en mayúscula, StudlyCaps o CapWords) Una nota breve: cuando uses acrónimos, debes poner en mayúsculas todas las letras que forman el acrónimo, p. ej., `HTTPServerError`
- `mySampleName` – mayúsculas y minúsculas, que en realidad difiere de CamelCase solo por tener un carácter inicial en minúscula
- `My_Sample_Name` – palabras en mayúsculas con guiones bajos (considerado feo por PEP 8)
- `_my_sample_name` – un nombre que comienza con un solo guión bajo inicial indica un "uso interno" débil, por ejemplo, la instrucción de `SAMPLE import *` no importará objetos cuyos nombres comiencen con un guión bajo. - `my_sample_name_` -– se utiliza un único guión bajo final por convención para evitar conflictos con palabras clave de Python, p. ej., `class_`
- `__my_sample_name` – un nombre que comienza con un guión bajo inicial doble se utiliza para atributos de clase donde invoca la alteración de nombres, p. ej., dentro de `class MySampleClass`, `__room` se convertirá en `_MySampleClass__room`
- `__my_sample_name__` – un nombre que comienza y termina con un guión bajo doble se utiliza para objetos y atributos "mágicos" que residen en espacios de nombres controlados por el usuario, p. ej., `__init__`, `__import__` o `__file__`. No debe crear dichos nombres, sino usarlos solo como se documenta.

Convenciones de nomenclatura: recomendaciones

PEP 8 establece una convención de nomenclatura específica con respecto a un identificador específico.

Al asignar un nombre a una variable, debe utilizar una letra minúscula o una palabra o palabras, y separar las palabras con guiones bajos, p. ej., `x`, `var`, `mi_variable`. La misma convención se aplica a las variables globales.

Las funciones siguen las mismas reglas que las variables, es decir, al asignar un nombre a una función, debe utilizar una letra minúscula o una palabra o palabras separadas por guiones bajos, p. ej., `fun`, `mi_función`.

Al asignar un nombre a una clase, debe adoptar el estilo CamelCase, p. ej., `MiClaseDeMuestra`, o si solo hay una palabra, comience con una letra mayúscula, p. ej., `Muestra`.

Al dar un nombre a un método, debe utilizar una o más palabras en minúscula separadas por guiones bajos, p. ej., método, `mi_metodo_de_clase`. Siempre debe utilizar self como primer argumento para los métodos de instancia y cls como primer argumento para los métodos de clase.

Al dar un nombre a una constante, debe utilizar letras mayúsculas y separar las palabras con guiones bajos, p. ej., `TOTAL`, `MI_CONSTANTE`.

Al dar un nombre a un módulo, debe utilizar una o más palabras en minúscula, preferiblemente cortas, y separarlas con guiones bajos, p. ej., `samples.py`, mis_muestras..

Al dar un nombre a un paquete, debe utilizar una o más palabras en minúscula, preferiblemente cortas. No debe separar las palabras, p. ej., `paquete`, `mipaquete`.

Los nombres de las variables de tipo deben seguir la convención CamelCase y ser cortos, p. ej., `AnyStr` o `Num`.

Al asignar un nombre a una excepción, debe seguir la misma convención que con las clases (tenga en cuenta que las excepciones deben ser en realidad clases), es decir, utilice el estilo CamelCase.

Nota: puede utilizar un estilo diferente, p. ej., mayúsculas y minúsculas mixtas (`mySample`) para funciones y variables, pero solo si esto ayuda a mantener la compatibilidad con versiones anteriores y si ese es el estilo predominante.

Para obtener información más detallada sobre las convenciones de nombres de PEP 8, vaya a la página oficial de PEP 8: https://www.python.org/dev/peps/pep-0008/#prescriptive-naming-conventions.

Recomendaciones de programación

A menudo, existen múltiples formas de escribir código que realizarán la misma acción en Python; sin embargo, PEP 8, nuevamente, impone ciertas convenciones y brinda consejos sobre cómo debe seguir las mejores prácticas de programación para evitar la ambigüedad, mantener la coherencia con su código anterior y las bibliotecas de Python, y lograr un mejor rendimiento/eficiencia del código.

A continuación, se incluyen:

– Realice comparaciones con el objeto `None` con el uso de `is` o `is not`, no con los operadores de (des)igualdad (`==` y `!=`), por ejemplo:

```
# Bad:

if x == None:
    print("A")


# Good:

if x is None:
    print("A")
```

– no utilices los operadores de (des)igualdad al comparar valores booleanos con Verdadero o Falso. Nuevamente, utiliza `es` o `no es` en su lugar:

```
# Bad:

my_boolean_value = 2 > 1
if my_boolean_value == True:
    print("A")
else:
```

```
    print("B")


# Good:

my_boolean_value = 2 > 1
if my_boolean_value is True:
    print("A")
else:
    print("B")


# Better:

my_boolean_value = 2 > 1
if my_boolean_value:
    print("A")
else:
    print("B")
```

– para facilitar la lectura, utilice el operador `is not` en lugar de `not … is`:

```
# Bad:

if not x is None:
    print("It exists")


# Good:

if x is not None:
    print("It exists")
```

Nota: evite usar `if x`: para expresar `if x is not None:` cuando desee verificar si a una variable o argumento determinado establecido en None de manera predeterminada se le ha asignado un valor diferente.

– cuando desee "capturar" una excepción, haga referencia a excepciones específicas en lugar de usar solo la cláusula `except:`:

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

– al comprobar prefijos o sufijos, utilice los métodos de cadena `''.startswith()` y `''.endswith()`, ya que son más claros y menos propensos a errores. En general, es mejor utilizar métodos de cadena en lugar de importar el módulo de cadena.

```
# Bad:

if name[:4] == 'Adam':
    # do something


# Good:

if name.startswith('Adam'):
    # do something
```

Para obtener más sugerencias sobre cómo escribir mejor código y qué prácticas debería evitar, consulte la página oficial de Recomendaciones de programación de PEP 8.

Ahora, pongamos en práctica algunas de las cosas que hemos aprendido.

1.2 PEP 257 – Convenciones de cadenas de documentación

¿Qué es PEP 257?

PEP 257 es un documento creado como parte de la Guía del desarrollador de Python, que intenta estandarizar la estructura de alto nivel de las cadenas de documentación. Describe las convenciones, las mejores prácticas y la semántica (¡no las leyes ni las regulaciones!) asociadas con la documentación del código Python mediante cadenas de documentación. En resumen, intenta responder las siguientes dos preguntas:

- ¿Qué deben contener las cadenas de documentación de Python?
- ¿Cómo se deben utilizar las cadenas de documentación de Python?

¿Qué son las cadenas de documentación?

Una cadena de documentación es "una cadena literal que aparece como la primera declaración en una definición de módulo, función, clase o método. Dicha cadena de documentación se convierte en el atributo especial `__doc__` de ese objeto". (PEP 257)

En otras palabras, las cadenas de documentación son cadenas de documentación de Python que se utilizan en la definición de clase, módulo, función y método para proporcionar información sobre la funcionalidad de un fragmento de código más grande de forma prescriptiva.

Ayudan a los programadores (incluido usted) a recordar y comprender el propósito, el funcionamiento y las capacidades de bloques o secciones de código particulares.

Docstrings vs. comentarios

Antes de continuar, debemos entender esta distinción esencial (como sugieren sus nombres): los comentarios se utilizan para comentar el código, mientras que las docstrings se utilizan para documentar el código. Entonces, ¿cuál es la diferencia entre comentarios y docstrings y, en última instancia, entre comentar y documentar código?

Observa la tabla a continuación, donde queremos mostrarte algunas de las diferencias entre comentarios y docstrings en Python:

Comentarios	Docstrings
Los comentarios son declaraciones no ejecutables en Python, lo que significa que el intérprete de Python los ignora; no se almacenan en la memoria y no se puede acceder a ellos durante la ejecución del programa (es decir, se puede acceder a ellos mirando el código fuente).	Se puede acceder a las docstrings leyendo el código fuente y utilizando el atributo `__doc__` o la función `help()`.
El objetivo principal de los comentarios es aumentar la legibilidad y la comprensión del código, y explicar el código al usuario de una manera significativa. El usuario aquí se refiere tanto a otros programadores como a usted (por ejemplo, cuando vuelve a su código después de un tiempo), alguien que querrá o necesitará modificar, ampliar o mantener el código.	El objetivo principal de las cadenas de documentación es documentar su código, es decir, describir su uso, funcionalidad y capacidades a usuarios que no necesariamente necesitan saber cómo funciona.

Los comentarios no se pueden convertir en documentación; su propósito es simplificar el código, proporcionar información precisa y ayudar a comprender la intención de un fragmento o una línea en particular.	Las cadenas de documentación se pueden convertir fácilmente en documentación real, que describe el comportamiento de un módulo o una función, el significado de los parámetros o el propósito de un paquete específico.

Por supuesto, como verá en las siguientes páginas, hay mucho más que queremos contarle sobre las cadenas de documentación: cómo usarlas, por qué usarlas y dónde; y – como es de esperar – la diferencia entre comentarios y docstrings se hará aún más evidente.

¿Por qué comentar? ¿Por qué documentar?

Antes de profundizar en el tema de los docstrings, intentemos responder a la pregunta: ¿por qué es importante comentar y documentar el código?

Básicamente, no debemos olvidar esta simple regla de Guido van Rossum: "El código se lee más a menudo de lo que se escribe", lo que básicamente significa que el código que escribimos hoy probablemente será leído en el futuro, ya sea por usted, por otro programador o incluso por equipos de programadores.

Por lo tanto, es crucial que desarrollemos hábitos de programación y escritura de código que permitan a los desarrolladores y otros usuarios comprender los porqués y los cómo del código, ya que esto hará que la reutilización y la contribución al código sean mucho más fáciles.

Por lo tanto, deberíamos estar de acuerdo en que documentar el código ayuda a mantener un código más limpio, más legible y más sostenible, lo que significa que es una de las mejores prácticas que un desarrollador responsable y bueno debería adoptar como parte de su conjunto de herramientas de taller de programación diaria.

Un breve resumen de los comentarios

Esperamos que recuerdes que los comentarios en Python se crean usando el signo numeral (#). Deben ser bastante breves (no más de 72 caracteres por línea), comenzar con una letra mayúscula y terminar con un punto.

Si necesitas incluir un comentario más largo en tu código, puedes usar un comentario de varias líneas, en cuyo caso debes usar el signo numeral al comienzo de cada línea de comentario.

Generalmente, debes insertar comentarios cerca del código que estás describiendo para que el lector sepa a qué parte del código te refieres. Debes ser preciso: no incluyas información irrelevante o redundante; y, sobre todo, intenta diseñar y escribir tu código de tal manera que se comente a sí mismo de manera fácil y comprensible (por ejemplo, dale nombres de autocomentarios a las variables).

¿Cuándo usar comentarios?

Aparte de los casos más obvios, como las descripciones de código y algoritmos, los comentarios pueden tener otros propósitos útiles. Por ejemplo:

- pueden ayudarte a etiquetar aquellas secciones de código que se realizarán en el futuro o que se dejaron para mejoras posteriores, por ejemplo:

```
# TODO: Add a function that takes the val and prc arguments.
```

- Pueden ayudarte a comentar (y descomentar) aquellas secciones de código que quieras probar, por ejemplo:

```
def fun(val):
    return val * 2

user_value = int(input("Enter the value: "))
# fun(user_value)
# user_value = user_value + "foo"

print(fun(user_value))
```

- Pueden ayudarte a planificar tu trabajo y delinear ciertas secciones del código que estarás diseñando, por ejemplo:

```
# Step 1: Ask the user for the value.
# Step 2: Change the value to an int and handle possible exceptions.
# Step 3: Print the value multiplied by 0.7.
```

Algunas palabras sobre las sugerencias de tipo: PEP 484

Antes de pasar de hablar sobre los comentarios a profundizar en las cadenas de documentación, hay una característica más de Python de la que queremos hablarte brevemente: las sugerencias de tipo.

Las sugerencias de tipo son un mecanismo introducido con Python 3.5 y descrito en PEP 484 que te permite equipar tu código con información adicional sin usar comentarios. Es una característica opcional, pero más formalizada, que te permite usar el módulo de tipado integrado de Python para proporcionar información de sugerencias de tipo en tu código con el fin de dejar ciertas sugerencias, marcar ciertos posibles problemas que puedan surgir en el proceso de desarrollo y etiquetar nombres específicos con información de tipo.

En pocas palabras, las sugerencias de tipo te permiten indicar estáticamente la información de tipo relacionada con los objetos de Python, lo que significa que puedes, por ejemplo, agregar información de tipo a una función: indicar el tipo de un argumento que acepta la función o el tipo de valor que devolverá. Observa los siguientes ejemplos:

```
# No type information added:
def hello(name):
    return "Hello, " + name


# Type information added to a function:
def hello(name: str) -> str:
    return "Hello, " + name
```

La sugerencia de tipo es opcional, lo que significa que PEP 484 no lo obliga a dejar ninguna información relacionada con el tipado estático en su código. El primer ejemplo no tiene ninguna sugerencia de tipo.

En el segundo ejemplo, la anotación `str`, que indica que el argumento `name` pasado a la función `hello()` debe ser del tipo str, nos ayuda a minimizar el riesgo de ciertas situaciones (in)esperadas: reduce el riesgo de pasar un tipo de valor incorrecto a la función. La anotación `-> str` también indica que la función `hello()` devolverá un valor del tipo *str*, que, por supuesto, es una cadena.

¿Qué significa todo esto para usted y cómo puede aprovechar la sugerencia de tipo en Python?

- En primer lugar, la sugerencia de tipo puede ayudarlo a documentar su código. En lugar de dejar información relacionada con argumentos y respuestas en cadenas de documentación, puede usar el lenguaje en sí para cumplir con este propósito. Esta puede ser una forma elegante y útil de resaltar parte de la información más importante del código, especialmente al publicar código en un proyecto, compartirlo con otros desarrolladores o dejar sugerencias para usted mismo cuando tenga que volver al código fuente en el futuro. En algunos de los proyectos de desarrollo de software más grandes, las sugerencias de tipos son una práctica recomendada que ayuda a los equipos a comprender mejor las formas en que los tipos se ejecutan a través del código.
- Las sugerencias de tipos le permiten notar ciertos tipos de errores de manera más efectiva y escribir un código más atractivo y, sobre todo, más limpio. Al usar sugerencias de tipos, piensa con más cuidado en los tipos en su código, lo que ayuda a prevenir o detectar algunos de los errores que pueden resultar de la naturaleza dinámica de Python. (Sin embargo, no somos defensores de hacer que Python requiera tipado estático).
- Debe recordar que las sugerencias de tipos en Python no se utilizan en tiempo de ejecución, lo que significa que toda la información de tipos que deja en el código en forma de anotaciones se borra cuando se ejecuta el programa. En otras palabras, las sugerencias de tipos no tienen ningún efecto en el funcionamiento de su código. Por otro lado, cuando se utiliza junto con algún sistema de verificación de tipos o herramientas similares a lint que se pueden conectar al editor o IDE, puede ayudar a escribir código completando automáticamente la escritura y detectando y resaltando errores antes de que se ejecute el código.
- Dado que las sugerencias de tipos no tienen efecto en el código fuente, esto significa que no tienen impacto en los tiempos de rendimiento (Python ignora los caracteres en tiempo de ejecución, lo que no tiene influencia en la velocidad de interpretación/compilación).

En esta breve sección solo hemos abordado las preguntas más fundamentales relacionadas con el tema de las sugerencias de tipos en Python. Para obtener más información sobre el tema, lo alentamos a que eche un vistazo más de cerca a PEP 483: la teoría de las sugerencias de tipos, PEP 484: sugerencias de tipos (información sobre la sintaxis para anotaciones de tipos, análisis estático y refactorización, verificación de tipos) y PEP 3107: anotaciones de funciones (información sobre la sintaxis para agregar anotaciones de metadatos a las funciones de Python).

Ahora es el momento de pasar a las cadenas de documentación.

Docstrings – ¿dónde y cómo?

Los docstrings deben estar entre comillas dobles triples (`"""triple comillas dobles"""`). Por ejemplo:Docstrings – ¿dónde y cómo? Ya hemos dicho que los docstrings pueden usarse en clases, módulos, funciones y definiciones de métodos. Ahora queremos profundizar en esto: hay casos en

los que no solo pueden incluirse, sino que deben incluirse. Para ser más precisos, todos los módulos, funciones, clases y métodos públicos que son exportados por un módulo determinado deben tener docstrings.

Los métodos no públicos no necesitan contener docstrings. Sin embargo, se recomienda que dejes un comentario justo después de la línea def que describa lo que realmente hace el método. En el caso de los paquetes, estos también deben estar documentados, y puedes escribir docstrings de paquetes en el docstring del módulo del archivo `__init__.py` en la carpeta del paquete.

Como hemos dicho antes, los docstrings son cadenas literales que aparecen como la primera declaración en un módulo, función, clase o método. Sin embargo, es importante (y justo) añadir que los literales de cadena también pueden aparecer en muchos otros lugares del código Python y seguir sirviendo como documentación. Y aunque ya no sean accesibles como atributos de objetos en tiempo de ejecución, todavía se pueden extraer con algunas herramientas de software específicas (para obtener más información sobre ellas, consulte PEP 256, que proporciona información sobre Docutils, un sistema de procesamiento de docstrings de Python).

Y ahora, sin entrar en demasiados detalles, es suficiente decir que distinguimos dos tipos de estos "docstrings adicionales": estos son:

- docstrings de atributo, que se encuentran inmediatamente después de una declaración de asignación en el nivel superior de un módulo (atributos de módulo), clase (atributos de clase) o la definición del método `__init__` de una clase (atributos de instancia). Estas son interpretadas por las herramientas de extracción, como Docutils, como "las cadenas de documentación del destino de la declaración de asignación". (Si está interesado en aprender más sobre las cadenas de documentación de atributos, puede sumergirse en PEP 224 por su cuenta. En este punto, solo queremos que esté al tanto de ellas).
- cadenas de documentación adicionales, que se encuentran inmediatamente después de otra cadena de documentación. (La idea original de las cadenas de documentación adicionales se tomó de PEP 216, que a su vez fue reemplazada posteriormente por PEP 287; nuevamente, puede echar un vistazo a esas cadenas de documentación).

Cómo crear cadenas de documentación

Las cadenas de documentación deben estar entre comillas dobles triples (" ""comillas dobles triples""). Por ejemplo:

```
def my_function():
    """I am a docstring."""
    ...
```

Si necesita utilizar barras invertidas en sus cadenas de documentación, debe seguir el formato `r"""comillas dobles triples sin formato"""`. Si necesita utilizar cadenas de documentación Unicode, siga el formato `u"""cadenas de comillas triples Unicode"""`.

Cadenas de documentación de una línea frente a cadenas de varias líneas

Existen dos formas de cadenas de documentación. Y aunque cada una de ellas tiene el mismo propósito (es decir, se supone que proporciona documentación), la que vaya a utilizar dependerá de ciertas condiciones, como la cantidad de información que es necesaria proporcionar. Estas son:

- **One-line vs. multi-line docstrings**: se utilizan para descripciones simples y breves, y deben caber en una línea;

Single-line docstring

```
def my_function():
    """One-line description."""
    body_of_the_fucntion
    ...
```

- **multi-line docstrings** – Se utilizan para casos más difíciles y deben constar de una línea de resumen seguida de una línea en blanco y una descripción más elaborada.

Multi-line docstring

```
def my_function(a, b, c):
    """Sumary line followed by a blank line.
    More elaborate description
    ...
    ...
    """
    body_of_the_function
    ...
```

Hablemos un poco más de cada uno de ellos.

One-line docstrings

One-line docstrings deben usarse para descripciones bastante simples, obvias y breves. Deben ocupar solo una línea y estar rodeadas de comillas dobles triples (las comillas de cierre deben estar en la misma línea que las comillas de apertura, ya que esto ayuda a mantener el código limpio y elegante).

Notas importantes:

- una cadena de documentación debe comenzar con una letra mayúscula (a menos que un identificador comience la oración) y terminar con un punto;
- una cadena de documentación debe prescribir el efecto del segmento de código, no describirlo. En otras palabras, debe tomar la forma de un imperativo (p. ej., "Haz esto", "Devuelve aquello", "Calcula esto", "Convierte aquello", etc.), no una descripción (p. ej., "Hace esto", "Devuelve aquello", "Forma esto", "Extiende aquello", etc.). Por ejemplo:

```
def greeting(name):
    """Take a name and return its replicated form."""
    return name * 2
```

- Una cadena de documentación no debe simplemente repetir los parámetros de la función o del método. Por ejemplo:

Don't do this: `def my_function(x, y):` `"""my_function(x, y) -> list""" ...`

Instead, try to do something like this:

```
def my_function(x, y):
    """Compute the angles and return a list of coordinates."""
...
```

- No utilice una línea en blanco encima o debajo de una cadena de documentación de una línea a menos que esté documentando una clase, en cuyo caso debe colocar una línea en blanco después de todas las cadenas de documentación que la documentan:

```
def calculate_tax(x, y):

    """I am a one-line docstring."""

    return (x+y) * 0.25


def calculate_tax(x, y):
    """I am a one-line docstring."""
    return (x+y) * 0.25
```

Multi-line docstrings

Las cadenas de documentación de varias líneas se deben utilizar para casos no obvios y descripciones más detalladas de segmentos de código. Deben tener una línea de resumen, similar a la que tiene una cadena de documentación de una sola línea, seguida de una línea en blanco y una descripción más elaborada. La línea de resumen puede estar ubicada en la misma línea que las comillas dobles triples de apertura o en la línea siguiente. Las comillas finales se deben colocar en una línea separada.

Notas importantes:

- una cadena de documentación de varias líneas debe tener la misma sangría que las comillas de apertura, por ejemplo:

```
def king_creator(name="Greg", ordinal="I", country="Neverland"):
    """Create a king following the article title naming convention.

    Keyword arguments:
    :arg name: the king's name (default: Greg)
    :type name: str
    :arg ordinal: Roman ordinal number (default: I)
    :type ordinal: str
    :arg country: the country ruled (default: Neverland)
    :type country: str
    """
    if name == "Voldemort":
        return "Voldemort is a reserved name."
    ...
```

- debe insertar una línea en blanco después de todas las cadenas de documentación de varias líneas que documentan una clase;
- las cadenas de documentación de scripts (en el sentido de programas independientes/ejecutables de un solo archivo) deben documentar la función del script, la sintaxis de la línea de comandos, las variables de entorno y los archivos. La descripción debe ser equilibrada de manera que ayude a los nuevos usuarios a comprender el uso del script, así como

proporcionar una referencia rápida a todas las características del programa para el usuario más experimentado;

- las cadenas de documentación de módulos deben enumerar las clases, excepciones y funciones exportadas por el módulo;
- las cadenas de documentación de paquetes (entendidas como la cadena de documentación del módulo `__init__.py` del paquete) deben enumerar los módulos y subpaquetes exportados por el paquete;
- Las cadenas de documentación de las funciones y los métodos de clase deben resumir su comportamiento y brindar información sobre los argumentos (incluidos los argumentos opcionales), los valores, las excepciones, las restricciones, etc.
- Las cadenas de documentación de las clases también deben resumir su comportamiento, así como documentar los métodos públicos y las variables de instancia. Por ejemplo:

```
class Vehicle:
    """A class to represent a Vehicle.

    Attributes:
    -----------
    vehicle_type: str
        The type of the vehicle, e.g. a car.
    id_number: int
        The vehicle identification number.
    is_autonomous: bool
        self-driving -> True, not self-driving -> False


    Methods:
    --------
    report_location(lon=45.00, lat=90.00)
        Print the vehicle id number and its current location.
        (default longitude=45.00, default latitude=90.00)
    """

    def __init__(self, vehicle_type, id_number, is_autonomous=True):
        """
        Parameters:
        -----------
        vehicle_type: str
            The type of the vehicle, e.g. a car.
        id_number: int
            The vehicle identification number.
        is_autonomous: bool, optional
            self-driving -> True (default), not self-driving -> False
        """

        self.vehicle_type = vehicle_type
        self.id_number = id_number
        self.is_autonomous = is_autonomous
```

```
def report_location(self, id_number, lon=45.00, lat=90.00):
    """
    Print the vehicle id number and its current location.

    Parameters:
    -----------
    id_number: int
        The vehicle identification number.
    lon: float, optional
        The vehicle's current longitude (default is 45.00)
    lat: float, optional
        The vehicle's current latitude (default is 90.00)
    """


...
...
...
```

Tipos de formato de cadenas de documentos

Es posible que hayas notado que hemos utilizado dos formatos de cadenas de documentos diferentes para documentar la función `king_creator()` y la clase `Vehicle`. El primer tipo de formato se llama reStructuredText y es el estándar oficial de documentación de Python explicado y descrito en PEP 287. El segundo ejemplo utiliza el formato de cadenas de documentación NumPy/SciPy (para obtener más detalles, haga clic aquí), que es una combinación del formato de cadenas de documentación de Google y el formato reStructuredText.

Ambos tipos de formato son buenos para crear documentación formal y ambos son compatibles con Sphinx, uno de los generadores de documentación de Python más populares.

Sphinx es una gran herramienta para crear documentación para proyectos de desarrollo de software. Utiliza reStructuredText como lenguaje de marcado y tiene muchas funciones útiles, como compatibilidad con el formato de salida HTML, prueba automática de fragmentos de código, referencias cruzadas extensas y una estructura jerárquica que le permite definir un árbol de documentos. Échele un vistazo.

Cómo documentar un proyecto

Al documentar un proyecto de Python, dependiendo de la naturaleza del proyecto (es decir, privado, compartido, público, abierto, etc.), se puede generar una estructura de documentos. fuente/dominio público), primero y ante todo debe definir a sus usuarios y pensar en sus necesidades. Crear un perfil de usuario puede resultar útil en este caso, ya que le ayudará a identificar las formas en que los usuarios utilizarán su proyecto.

Esto significa que puede mejorar fácilmente su experiencia pensando en cómo van a utilizar su código e intentando predecir los problemas más comunes que pueden encontrar al hacerlo.

En general, un proyecto debe contener los siguientes elementos de documentación:

- un archivo readme, que proporciona un breve resumen del proyecto, su propósito y posiblemente algunas pautas de instalación;

- un archivo examples.py, que es un script que muestra algunos ejemplos de cómo utilizar el proyecto;
- una licencia en forma de archivo txt (particularmente importante para proyectos de código abierto y de dominio público)
- un archivo de cómo contribuir que proporciona información sobre las posibles formas de contribuir al proyecto (proyectos compartidos, de código abierto y de dominio público).

Como documentar el código puede ser una actividad bastante agotadora y que requiere mucho tiempo, se recomienda encarecidamente que utilice algunas de las herramientas que podrían ayudarlo a generar automáticamente la documentación en el formato deseado y a gestionar las actualizaciones y el control de versiones de la documentación de una manera eficaz y eficiente.

Hay muchas herramientas y recursos de documentación disponibles, como Sphinx, que ya hemos mencionado, o el popular pdoc, y muchos más. Le recomendamos que siga este camino.

Linters y fixers

¿Cómo se mantiene la buena calidad de su código? Bueno, ya sabe que puede seguir las guías de estilo como PEP 8 o PEP 257 y escribir su código de una manera legible y consistente. Puede (y posiblemente deba) adoptar la filosofía Zen de Python, con todos sus buenos consejos para escribir un código elegante y fácil de mantener, y utilizar el mecanismo de sugerencias de tipos. Puede observar cómo otros escriben código y documentarlo como parte de sus proyectos (consulte la biblioteca estándar de Python o la biblioteca Requests) y aprender de ellos. Finalmente, puede utilizar linters.

Correcto. Pero, ¿qué es un linter? Bueno, es una herramienta que lo ayuda a escribir su código, porque lo analiza para detectar anomalías de estilo y errores de programación en comparación con un conjunto de reglas predefinidas. En otras palabras, es un programa que analiza tu código y reporta problemas tales como errores estructurales y de sintaxis, rupturas de consistencia y falta de compatibilidad con las mejores prácticas o pautas de estilo de código como PEP 8. Los linters más populares son: Flake8, Pylint, Pyflakes, Pychecker, Mypy y Pycodestyle (anteriormente Pep8): la herramienta oficial de linters para verificar el código Python con las convenciones de PEP 8.

Un fixer, por otro lado, es un programa que te ayuda a corregir estos problemas y formatear tu código para que sea consistente con los estándares adoptados. Los fixers más populares son: Black, YAPF y autopep8.

La mayoría de los editores e IDE (por ejemplo, PyCharm, Spyder, Atom, Sublime Text, Visual Studio, Eclipse + PyDev, VIM o Thonny) admiten linters, lo que significa que puedes ejecutarlos en segundo plano mientras escribes código. Esto permite detectar, resaltar e identificar muchas áreas problemáticas en su código, como errores tipográficos, problemas de tabulación y sangría incorrectos, llamadas de función con una cantidad incorrecta de argumentos, inconsistencias estilísticas, patrones de código peligrosos y muchos más, y formatear automáticamente su código según una especificación predefinida.

Dicho esto, lo alentamos a que explore el territorio de los linters y los fixers y comience a usarlos para mantener un código Python de alta calidad y simplemente hacer su vida más fácil.

Cómo acceder a las cadenas de documentación

Casi hemos llegado al final de nuestro viaje con PEP 257 y las cadenas de documentación. La última pregunta que aún queda por responder por completo es: ¿cómo podemos acceder realmente a las cadenas de documentación?

Lo hacemos mediante el atributo `__doc__` de Python: si hay algún literal de cadena presente después de la definición de una función/módulo/clase/método, se asocia con el objeto como su atributo `__doc__`, y este atributo proporciona la documentación de ese objeto.

Ejecute el código en el editor para ver qué sucede. Su salida debería ser similar a esto:

[2]:
```
def my_fun(a, b):
    """The summary line goes here.

    A more elaborate description of the function.

    Parameters:
    a: int (description)
    b: int (description)

    Returns:
    int: Description of the return value.
    """
    return a*b

print(my_fun.__doc__)
```

```
The summary line goes here.

    A more elaborate description of the function.

    Parameters:
    a: int (description)
    b: int (description)

    Returns:
    int: Description of the return value.
```

Pero también hay otra forma de acceder a las cadenas de documentación: puedes usar la función `help()`. Haz una pequeña modificación en tu código: reemplaza la invocación de la función print con la siguiente línea:

[3]:
```
help(my_fun)
```

```
Help on function my_fun in module __main__:

my_fun(a, b)
    The summary line goes here.

    A more elaborate description of the function.

    Parameters:
    a: int (description)
    b: int (description)
```

```
    Returns:
    int: Description of the return value.
```

Como puedes ver, la salida es más larga y descriptiva.

Ahora intenta acceder a las cadenas de documentación de cualquiera de las funciones integradas de Python (por ejemplo, print()). Luego, importa un módulo y accede a la documentación del módulo. Experimenta con el método `__doc__` y la función help(). Observa qué salidas obtienes y en qué se diferencian entre sí. Úsalos para obtener más información sobre los objetos integrados de Python.

Has aprendido mucho. ¡Puedes estar orgulloso de ti mismo!

[4]:
```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

[5]:
```
# Type Hiting comentarios, name se espera un string
# retorna un string:
def hello(name: str) -> str:
    return "Hello, " + str(name)

hello(20)
```

[5]:
```
'Hello, 20'
```

Gracias por la atención

Isabel Maniega