NORMATIVA DE CODIFICACION ESTANDAR PEP-8

1- ESTANDAR PARA LA ORGANIZACIÓN DE ARCHIVOS

Archivo del código fuente

a) **Formato de leyenda de derechos de autor**: Se recomienda incluir una línea comentada al principio de cada archivo de código fuente que indique la fecha, el autor y la universidad a la que se pertenece. Ejemplo
#Copyright © 2024 [autor]. Todos los derechos reservados
#Fecha: [creación o última modificación]
#universidad

b) **Gestión de archivos fuente y control de versiones:** Es importante establecer claramente que todos los archivos fuente, así como otros archivos relevantes (ejecutables, bibliotecas de funciones, etc.), deben ser gestionados mediante un sistema de control de versiones. Por lo tanto, todos los archivos deberían ser versionados y gestionados a través de Git y GitHub.

Ficheros de clase

a) **Uso de fichero distinto para cada clase:** Cada archivo solo tendrá una clase, el nombre de la clase podrá coincidir con el nombre del archivo

b) **Inclusión de cabecera en cada fichero de clase:** Se recomienda incluir una cabecera en cada archivo de clase que proporcione información resumida sobre la clase, su utilización y cualquier precaución o consideración importante para su revisión o actualización posterior. Esta cabecera podría contener información como el propósito de la clase, una breve descripción de su funcionalidad, los parámetros de entrada y salida (si corresponde), y cualquier otra información relevante para los desarrolladores que puedan trabajar con esa clase en el futuro.

2- ESTANDAR PARA NOMBRES Y VARIABLES

Clases y miembros de clases

a) **Nombrado de clases y tipos de datos:** Se usarán para el nombrado de las clases el estilo PascalCase (primera letra en mayúscula) y para el nombrado de los argumentos camelCase (primera letra en minúscula).

b) **Nombrado de campos, métodos, propiedades y constantes:** Se usarán minúsculas para los campos (camelCase), minúsculas para las constantes (LOWER_CASE), y camelCase para métodos y propiedades.

Visibilidad

a) **Acceso a variables:** Las variables podrán ser públicas o podrán ser utilizadas mediante propiedades según convenga en el uso de dicha variable. Teniendo en cuenta que El uso de propiedades puede proporcionar un mejor control sobre el acceso y la modificación de los datos.

b) **Formato del texto del código fuente:** Cada declaración debe de estar en una línea separada para una mejor visibilidad y legibilidad del código. Por otro lado, las declaración e inicializaciones pueden estar en la misma línea.

c) **Inicialización de campos o variables:** Las variables siempre estará inicializadas a un valor, se establecerán en la cabecera de la clase (si hay) o en la cabecera de la función/método, para los bucles se podrán inicializar en las líneas anteriores siempre y cuando sea la variable de índice.

d) **Uso de 'self' en nombre de campos:** Dentro de una clase todo método deberá de empezar con 'self' para hacer referencia a los atributos y métodos de una clase.

3- ESTANDAR PARA LOS METODOS

Estructura de los métodos:

a) **Tamaño máximo de los métodos:** Se ha establecido que el tamaño máximo por línea sea de 79 caracteres. Se dividirá el cuerpo del método en bloques lógicos, cada uno encargado de realizar una tarea específica. No hay límite en cuanto a poner comentarios, pero sí que se sugiere que el código sea lo más auto explicativo para evitar ensuciar de más el código.

b) **Identación y llaves:** Para esto se establece que entre bucles y condicionales se aplicaran 4 espacios o un tabulador para que sea más legible el código. Se podrán hacer condicionales y bucles de una sola línea siempre y cuando sea legible y no sea muy sucio.

Declaración de los métodos

a) **Declaración multilínea de métodos:** Si un método necesita más de una línea, la siguiente línea siempre debe empezar en línea con el primer atributo del método.

b) **Descripción de métodos:** Los métodos deben de tener una descripción inicial en donde se explique su funcionalidad para así facilitar la compresión del código.

4- ESTANDAR PARA EL TRATAMIENTO DE EXCEPCIONES

Tratamiento de excepciones

a) **Detección de errores en los argumentos de entrada:** Se verificará la validez de los argumentos de entrada para asegurar que cumplan con los requisitos esperados por el método o función., Se implementarán mecanismos para capturar y manejar excepciones que puedan surgir debido a datos de entrada incorrectos o inesperados. Se definirán métodos claros para comunicar los errores detectados en los argumentos de entrada, proporcionando mensajes descriptivos y útiles para facilitar la depuración y corrección de los problemas.

b) **Visualización de excepciones:** Se evaluará si es más adecuado que las excepciones sean tratadas localmente por los componentes individuales del código o si es preferible centralizar su manejo en la capa más alta de la aplicación. Se analizará cómo esta decisión afectará la claridad y la mantenibilidad del código. Tener un enfoque coherente en todo el sistema garantizará una comprensión más clara de cómo se manejan las excepciones y quién es responsable de su gestión. Se definirá cómo se propagarán las excepciones a través de los distintos niveles del sistema. Determinar si las excepciones deben ser gestionadas localmente o propagadas hacia arriba en la jerarquía de llamadas influirá en cómo se manejan y comunican los errores en diferentes partes del código.

c) **Control de métodos propensos a excepciones:** Se requerirá la implementación de bloques try-except para capturar y manejar las excepciones que puedan ocurrir durante la ejecución del método. Se promoverá la validación exhaustiva de los datos de entrada para prevenir situaciones que puedan conducir a excepciones. Esto garantizará que los métodos funcionen correctamente incluso con datos imprevistos. Se fomentará la implementación de mecanismos de recuperación de errores para mantener la integridad del sistema ante posibles fallas. Esto puede incluir la repeticion de operaciones, la notificación adecuada al usuario o la aplicación de estrategias alternativas para continuar con la ejecución del programa.

5- OTROS ESTANDARES

Separacion de argumentos

a) **Separacion de argumento:** Los argumentos deben separarse por una coma seguida de un espacio en blanco (‘‘, ’’), tanto en la definición de la función como en su llamada.
b) **Colocacion de paréntesis:** El paréntesis de apertura debe estar contiguo al nombre de la función, seguido de los argumentos, y el paréntesis de cierre debe estar alineado verticalmente con el paréntesis de apertura.

c) **Normas especiales para estructuras de control:** Para las sentencias for, while, if etc., se debe seguir la misma convención de separación de argumentos y colocación de paréntesis que para las funciones.

6- MODIFICACION PYLINTRC

Aquí se muestran todos los cambios realizados sobre el archivo pylintrc. Solo se muestra como se establece no como estaban predefinidos.

a) Naming style matching correct arguments names:

Arguments-naming-style=camelCase

b) Bad variable names which should always be refused, separated by a comma.

bad-names = bb, tete, xd, efe

c) Naming style matching correct class constant names.

class-const-naming-style=LOWER_CASE

d) # Naming style matching correct constant names.

const-naming-style=LOWER_CASE

e) # Naming style matching correct function names.

function-naming-style=camelCase

f) # Naming style matching correct method names.

method-naming-style=camelCase

g) # Naming style matching correct module names.

module-naming-style=camelCase

h) Naming style matching correct variable names.

variable-naming-style=camelCase

i) # Maximum number of attributes for a class (see R0902).

max-attributes=10

j) # Maximum number of boolean expressions in an if statement (see R0916).

max-bool-expr=8

k) # Maximum number of branch for function / method body.

max-branches=10

l) # Maximum number of locals for function / method body.

max-locals=15

m) # Maximum number of return / yield for function / method body.

max-returns=10