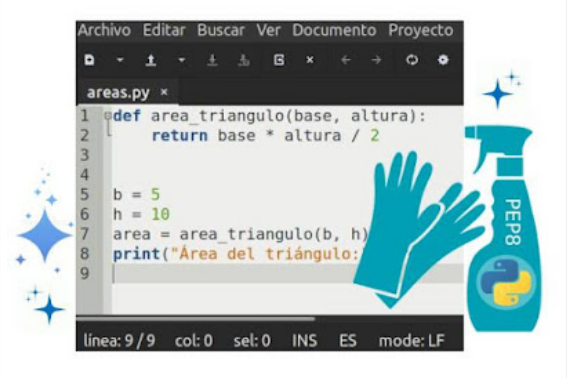
Python 3 para impacientes

"Simple es mejor que complejo" (Tim Peters)

domingo, 8 de septiembre de 2019

Programas con estilo en Python



Dentro de los documentos que recogen las propuestas de mejora del lenguaje Python (PEP) se encuentra la *Guía de estilo para código Python* (PEP 8) que contiene un conjunto de convenciones y pautas dirigidas a los programadores para mejorar la legibilidad de los programas y dotar los proyectos de cierta consistencia y coherencia.

Seguir las recomendaciones de **PEP 8** además de garantizar código limpio y ordenado, beneficia a todas las personas que trabajan en un mismo proyecto facilitando, en especial, la labor de los programadores cuando tienen que leer y comprender el código escrito por otras personas; incrementando el rendimiento y la imagen de profesionalidad de los equipos.

Sin más preámbulo pasamos a ver las recomendaciones más importantes de **PEP 8** y algunas herramientas útiles para revisar y corregir el estilo de nuestros programas.

Recomendaciones de PEP 8

Convenciones de nombres

1. Para declarar variables, funciones, clases, paquetes, etc. utilizar nombres que describan su uso: **imprimir_factura**, **edad**, **importe**, **IVA**, **Cliente**

2. Cuando se utilicen nombres cortos de un carácter evitar los caracteres **l**, **I** y **O** para no confundirlos con otros caracteres parecidos: **1**, **0**.

3. Para los nombres de variables usar una letra, una palabra o varias palabras, en minúsculas. Es recomendable separar las palabras con guiones bajos para mejorar la legibilidad: **z**, **contador**, **kms_cuadrados**.

4. Para los nombres de constantes usar una letra, una palabra o varias palabras, en mayúsculas. Es recomendable separar las palabras con guiones bajos para ganar en legibilidad: **E**, **GRAVEDAD**, **VELOCIDAD_LUZ**.

5. Para los nombres de funciones, de métodos y de módulos usar una palabra o varias palabras, en minúsculas. Para mayor claridad es recomendable separar las palabras con guiones bajos: **invertir**, **aplicar_dto**, **mi_modulo.py**

6. Para los nombres de clases usar una palabra o varias palabras, en minúsculas pero con la inicial en mayúsculas: **Usuario**,**VehiculoDeportivo**.

7. Para los nombres de excepciones usar una palabra o varias palabras, en minúsculas pero con la inicial en mayúsculas. Cuando se trate de un error los nombres deberían contener el sufijo "Error": **ErrorCalculoFechas**

8. Para los nombres de paquetes usar una palabra o varias palabras, en minúsculas: **paqcorreo**,

Buscar

Buscar

Python para impacientes

Python
IPython
EasyGUI
Tkinter
JupyterLab
Numpy

Anexos

Guía urgente de MySQL
Guía rápida de SQLite3

Entradas + populares

Dar color a las salidas en la consola
En Python para dar color a las salidas en la consola (o en la terminal de texto) existen varias posibilidades. Hay un método basado ...

Instalación de Python, paso a paso
Instalación de Python 3.6 A finales de 2016 se produjo el lanzamiento de Python 3.6 . El propósito de esta entrada es mostrar, pas...

Añadir, consultar, modificar y suprimir elementos en Numpy
Acceder a los elementos de un array. [], [,], ... Acceder a un elemento de un array. Para acceder a un elemento se utiliz...

Variables de control en Tkinter
Variables de control Las variables de control son objetos especiales que se asocian a los widgets para almacenar sus valore...

Cálculo con arrays Numpy
Numpy ofrece todo lo necesario para obtener un buen rendimiento cuando se trata de hacer cálculos con arrays. Por como está concebid...

Tkinter: interfaces gráficas en Python
Introducción Con Python hay muchas posibilidades para programar una interfaz gráfica de usuario (GUI) pero Tkinter es fácil d...

Operaciones con fechas y horas. Calendarios
Los módulos datetime y calendar amplían las posibilidades del módulo time que provee funciones para manipular expresiones de ti...

Convertir, copiar, ordenar, unir y dividir arrays Numpy
Esta entrada trata sobre algunos métodos que se utilizan en Numpy para convertir listas en arrays y viceversa; para copiar arrays d...

Tkinter: Tipos de ventanas
Ventanas de aplicación y de diálogos En la entrada anterior tratamos los distintos gestores de geometría que se utilizan para di...

El módulo random
El módulo random de la librería estándar de Python incluye un conjunto de funciones

funconver.

9. El doble guion bajo al principio y final de un nombre está reservado para determinados objetos y atributos con nombres predefinidos (como **__name__**, **__file__**, **__init__**).

Diseño

Líneas de código

10. Las líneas de código deben tener una longitud máxima de **79** caracteres.

11. Las líneas largas que superen la longitud máxima se dividirán en varias líneas utilizando, preferentemente, la continuación de línea implícita que conlleva el uso de paréntesis, corchetes y llaves:

```
lista1 = ['uno', 'dos', 'tres', 'cuatro', 'cinco',
          'seis', 'siete']
```

12. En otros casos para dividir una línea que supere el límite debe utilizarse la barra diagonal invertida "\"-

```
if variable1 > variable2 and variable3 > variable4 \
   or variable1 > variable4:
```

13. Cuando se trate de operaciones matemáticas extensas otra forma de dividir una línea, que da más claridad al cálculo que se realiza, consiste en situar cada operando en una línea precedido del operador. Desde el punto de vista matemático tiene más sentido leer primero el operador y después la variable a la que afecta.

```
total = (variable1
         + variable2
         - variable3
         - variable4)
```

Codificación

14. En Python 3 los archivos de programas que utilicen la **codificación UTF-8** no requieren incluir la siguiente declaración al principio del código:

```
# -*- coding: utf-8 -*-
```

La mayoría de los editores incorporan una opción para establecer esta codificación por defecto:

- En el editor Geany la opción se encuentra en el **menú Editar**, **Preferencias**, **Archivos**, **Codificaciones**, Codificación predeterminada (para los archivos nuevos).
- En Visual Studio Code la opción está en el **menú Archivo**, **Preferencias**, **Editor de texto**, **Archivos**, Encoding.

Tanto **Geany** como **Visual Studio Code** muestran en la barra de estado la codificación de los archivos fuente con los que se trabaja en un momento dado.

Líneas en blanco

15. Antes y después de la definición de una función o una clase se deben dejar dos líneas en blanco.

```
def funcion1():
    pass


def funcion2():
    pass


variable1 = 0
variable2 = 1
```

que permiten obtener de distintos modos números a...

Archivo

septiembre 2019 (1)

python.org

pypi.org

Sitios

- ActivePython
- Anaconda
- Bpython
- Django
- Flask
- Ipython
- IronPython
- Matplotlib
- MicroPython
- Numpy
- Pandas
- Pillow
- PortablePython
- PyBrain
- PyCharm
- PyDev
- PyGame
- Pypi
- PyPy
- Pyramid
- Python.org
- PyTorch
- SciPy.org
- Spyder
- Tensorflow
- TurboGears

16. Antes y después de la definición los métodos de una clase dejar una línea en blanco.

```
class Clase1():
    """clase Clase1"""
    varclase1 = "variable de clase1"

    def metodo1(self, var1):
        self.var1 = var1

    def metodo2(self):
        self.var1 += 1
```

17. En funciones y métodos extensos en los que sea posible agrupar el código en bloques que identifiquen diferentes procesos, para facilitar la comprensión dejar una línea en blanco de separación entre ellos. No agregar líneas en blanco al final del código.

Sangría

18. Sangrar el código significa mover un bloque del mismo hacia la derecha insertando espacios o tabuladores para mejorar su legibilidad. Para sangrar el código utilizar preferentemente el carácter del espacio en blanco. Se recomienda utilizar sangrias de **4 espacios en blanco** en cada nivel. (Python 3 no permite mezclar sangrías de espacios con las basadas en tabuladores).

```
if var1 > valor:
    print('Cadena de texto1')
    if not var2:
        print('Cadena de texto2')
```

19. Cuando una sentencia ocupa más de una línea, el código que sigue en la siguiente línea puede estar alineado con el carácter delimitador de apertura.

```
lista1 = ['texto1', 'texto2', 'texto3',
          'texto4', 'texto5', 'texto6']
```

20. El paréntesis, el corchete y la llave de cierre pueden alinearse bajo el primer carácter del primer elemento de la lista de la línea anterior:

```
lista1 = [
    elemento1, elemento2,
    elemento3, elemento4
    ]
```

21. El cierre también puede estar alineado con el primer carácter de la primera línea donde comience la sentencia:

```
lista1 = [
    elemento1, elemento2,
    elemento3, elemento4
]
```

22. En declaraciones de funciones o clases que abarcan más de una línea se puede dejar una sangría doble (de 8 espacios) en el código que sigue en la siguiente línea para que se distingan adecuadamente los argumentos:

```
def funcion(
        valor1, valor2, valor3):
    total = valor1 + valor2 + valor3
    total = total * 25
```

23. Si se escriben argumentos en la primera línea los que continúan en la siguiente debe estar alineados verticalmente con los primeros:

```
def funcion(var1, var2,
            var3):
    total = var1 + var2 + var3
    return(total * 5)
```

24. En la asignación de una función a una variable los argumentos se escriben agregando una sangría de cuatro espacios. Este tipo de sangría se llama colgante porque en la declaración, que abarca varias líneas, todas las líneas están sangradas excepto la primera. En la primera línea donde aparece el nombre de la función no se incluyen argumentos (o ningún elemento si se tratara de la asignación de una lista, etc.).

```
datos1 = obtener_datos(
    var1, var2,
    var3, var4)
datos2 = 34
```

25. Otra posibilidad que contempla **PEP 8** cuando se asigna una función a una variable consiste e escribir los argumentos dejando una sangría con un número de espacios menor de 4.

```
datos1 = obtener_datos(
  var1, var2,
  var3, var4)
datos2 = 34
```

26. En declaraciones **if** que ocupan varias lineas la suma de los dos caracteres "**if**" más el espacio en blanco que le sigue más un paréntesis en la primera línea definen el espacio equivalente a una sangría de 4 espacios para las siguientes líneas. Esta coincidencia puede suponer un conflicto visual cuando finaliza la condición y a partir de la siguiente línea se continúa añadiendo código manteniendo la misma sangría. Se puede optar por no añadir una sangría extra si no afecta esta situación a la legibilidad:

```
if (condicion1 and condicion2 and
    condicion3 and condicion4):
    var1 = var2
```

27. Otra posibilidad consiste en insertar un comentario que delimite la declaración **if** del resto de líneas:

```
if (condicion1 and condicion2 and
    condicion3 and condicion4):
    # Si todas las condiciones son ciertas se asigna var2 a var1
    var1 = var2
```

28. O añadir una sangría adicional en la declaración **if** en la segunda línea:

```
if (condicion1 and condicion2 and
        condicion3 and condicion4):
    var1 = var2
```

Comentarios

29. Las líneas que contengan comentarios o cadenas de documentación deben tener una longitud máxima de **72 caracteres**.

```
# Lista de funciones


def funcion1(x, y):
    """
    Función: funcion1
    Devuelve el resultado de la operación x+y
    """
    return x + y


print(funcion1(10, 20))
print(funcion1.__doc__)
```

30. Los comentarios deben estar construidos con oraciones completas que hay que actualizar cuando el código sufra algún cambio. Un comentario comienza con el carácter **#** seguido de un espacio en blanco al que sigue el texto comenzando su escritura con mayúsculas y terminando cada oración con un punto. Si el comentario está formado por varias oraciones hay que separarlas con con dos espacios en blanco. Si el comentario es corto no es necesario terminar conun punto.

```
# Esta es la primera oración.  Y sigue otra separada con dos espacios.
```

31. Los comentarios de un bloque de código sangrado deben comenzar su escritura manteniendo el mismo nivel de sangrado. Si el comentario está formado por varios párrafos tendremos que dejar entre ellos una línea en blanco que comience con el carácter **#**.

```
if condicion1 and condicion2:
    # Primer párrafo del comentario.
    #
    # Segundo párrafo del comentario.
    variable1 = variable2 ** 2
    fucion1(variable1)
```

32. En líneas que contengan código utilizar con moderación los comentarios. Deben servir para aclarar algún punto no para redundar en lo obvio. Comenzar la escritura del comentario dejando dos espacios en blanco.

```
variable1 = variable2 + (x * y * z)  # Comentario de línea
```

33. Una **cadena de documentación** o **docscring** es un texto delimitado o entre triples comillas - simples (**'''**) o dobles (**"""**)- situado al principio de un módulo, una clase, una función o un método que sirve para explicar la utilidad del código. En cadenas de documentación multilineas terminar el texto situando la triple comillas de cierre aislada en la línea final. Preferentemente, emplear la triple comillas doble y en docstring de una línea cerrar la cadena en la misma línea. Para más información consultar el documento PEP 257.

```
def funcion1(param1, param2):
    """Función que devuelve ...

    Texto explicativo detallando procesos...
    Si param1 > param2:
    valor = param1 * param2
    En caso contrario:
    valor = param1 / param2

    """
    if param1 > param2:
        total = param1 * param2
    else:
        total = param1 / param2

    return total


""" Cadena de documentación de una línea """
```

Espacios en blanco

34. En asignaciones que utilizan el signo igual (**=**) o (**+=**, **-=**, etc.) agregar un espacio en blanco delante y después del signo. Aplicar la misma recomendación con los operadores de comparación (**==**, **<**, **>**, **!=**, **<>**, **<=**, **>=**, **in**, **not in**, **is**, **is not**), con los operadores booleanos (**and**, **or**, **not**) y con las flechas (**->**) en las anotaciones de las funciones.

```
var1 = 1
var2 += 1
if var1 == var2:
    var3 = 3
if not var1 and var2 is int:
    print(var1)


def funcion1() -> cadena:
    pass
```

35. En asignaciones a parámetros de funciones o métodos no añadir espacio en blanco delante y detrás del signo igual (**=**).

```
def funcion1(var1=0, var2=0):
    pass
```

36. Cuando se utilicen operadores con diferentes prioridades, considerar agregar espacios en blanco alrededor de los operadores con las prioridades más bajas. También se puede seguir la misma pauta con declaraciones **if** donde hay múltiples condiciones.

```
var1 = 1*var2 + 2/var3
var2 = (x+y) * (x-y)

if var1>0 and var!=10:
    pass
```

37. En sentencias que utilizan el método **[::]** que referencia a un subconjunto de elementos de una secuencia (como un grupo de caracteres de una cadena de texto, elementos de una lista, etc) no agregar espacios delante o detrás de los dos puntos (**:**) a no ser que mejore la legibilidad de una expresión. En referencias a elementos de listas, tuplas y diccionarios no incluir espacios en blanco, ni antes ni después.

```
tupla1 = (lista1[1:9], lista1[1:9:3], tupla2[1])
var1 = lista1[x+1 : x+y+1]
dicc1['a'] = lista1[0]
```

38. En expresiones y declaraciones entre paréntesis, corchetes y llaves no agregar inmediatamente después de la apertura o antes del cierre espacios en blanco. Después de una coma añadir un espacio en blanco, excepto entre una coma final y el cierre. No agregar espacios en blanco delante de una coma, punto y coma o dos puntos. No se recomienda dejar espacios delante del paréntesis de apertura que recoge los argumentos de una función. Tampoco después de una declaración al final de la línea, ni entre variables y sus valores para que queden alineados verticalmente.

```
lista1 = [1, 2, 3, 4, 5, (6, 7)]
tupla2 = ('a', 'b', 'c')
tupla3 = ('a',)
dicc1 = {'a': '1', 'b': '2', 'c': '3'}
var1 = funcion1(var2, var3)

# No se recomienda
variable1 = 1
par1      = True
```

Otras recomendaciones

39. Aunque la sintaxis de Python lo permita no es recomendable escribir más de una declaración por línea:

```
# Recomendado
if var1 == 'a':
    hacer()
try:
    print('Hecho')
except ValueError:
    print('Error')

# No recomendado
if var1 == 'a': hacer()
try: hacer()
except: print('error')
finally: finalizar()
```

40. Se aconseja utilizar comas finales en tuplas de un elemento y en listas que se extiendan por varias líneas, tras cada elemento.

```
tupla1 = (var1,)
lista1 = (
    'uno.dat',
    'dos.dat',
    'tres.dat',
    )
```

41. Para asignaciones de cadenas largas utilizar tantas cadenas entre comillas como sea necesario, ocupando cada una de ellas una línea y delimitadas por paréntesis en lugar de cadenas de documentación.

```
# Recomendado
cadena_larga = (
    "Uno, dos, tres, cuatro, cinco, seis, siete, "
    "ocho, nueve y diez."
)

# No recomendado
cadena_larga = """Uno, dos, tres, cuatro, cinco, seis, siete, \
    ocho, nueve y diez.”"""
```

42. Para evaluar si una variable booleana tiene el valor **True** utilizar **if var1** en lugar de **if var1 == True**:

```
# Recomendado
if var1:
    print('Es verdadero')

# No recomendado
if var1 == True:
    print('Es verdadero')
```

43. Para evaluar si una cadena, lista o tupla están vacías o no tiene elementos utilizar **if var1** o **if not var1** en lugar de **if len(var1)** o **if not len(var1)**.

```
variable1 = ''
lista1 = []
if not variable1 and not lista1:
    print('Tanto la variable como la lista están vacías')
```

44. Para evaluar si una variable tiene o no un valor definido utilizar **if var1 is not None** o **if var is None** en lugar de **if var1 != None** o **if var1 == None**.

```
variable1 = None
if variable1 is None:
    print('La variable no tiene ningún valor definido')
```

45. Para evaluar si una cadena de texto comienza o termina por otra utilizar los métodos **startswith**() y **endswith**() en lugar del método **[:]**.

```
# Recomendado
cadena = 'Python para impacientes'
if cadena.startswith('Python'):
    print('La cadena de texto comienza con "Python"')

# No recomendado
if cadena[0:6] == 'Python':
    print('La cadena de texto comienza con "Python"')
```

46. Para operar con cadenas utilizar preferentemente los métodos de cadenas en lugar del módulo **string**.

47. Para comparar distintos tipos de objetos utilizar **isinstance**() en lugar de **type**().

```
variable1 = 3.5
if isinstance(variable1, (int, float)):
    print('La variable es de tipo numérica')
```

48. En claúsulas **try/except** limitar **try** al mínimo de líneas y usar los nombres de las excepciones para capturar los errores.

```
var1 = 10
try:
    var1 = var1 / 2
    print(var1)
except ValueError:
    print('Error')
```

Herramientas para revisar y corregir

49. Es recomendable utilizar un lint para revisar si el estilo de nuestros programas cumple **PEP 8**. Los más conocidos son **pycodestyle** y **flake8** que están disponibles en el repositorio PyPI. Ambos buscan errores de estilo en un programa y muestran mensajes que ayudan a subsanarlos.

Para instalar **pycodestyle**:

```
$ pip install pycodestyle
```

Para añadir una opción al menú del editor **Geany** que compruebe con **pycodestyle** el estilo de un programa: acceder con un archivo .py abierto al **menú Construir**, **Establecer comandos de construcción**, y en la **línea 3** de **Comandos de Python** añadir la etiqueta **Estilo** asociada al siguiente comando y aceptar los cambios:

```
pycodestyle --max-line-length=79 "%f"
```

En el **menú Construir** se ha añadido la opción **Estilo** para comprobar el estilo de los fuentes cuando sea necesario.

Desde la línea de comandos también es posible verificar el estilo:

```
$ pycodestyle programa.py
```

50. Y para arreglar aquellos programas del pasado sin estilo se puede utilizar la herramienta **black** que reformatea el código aplicando las recomendaciones de **PEP 8**.

Para instalar black:

```
$ pip install black
```

El siguiente programa areas.py no cumple **PEP 8**:

areas.py:

```
def area_triangulo(base,altura):
    return base*altura/2

b= 5
h=10
area=area_triangulo(b,h)
print('Área del triángulo:', area)
```

Para reformatear con **black** el código de **areas.py** aplicando las recomendaciones, ejecutar:

```
$ black --line-length=79 areas.py

black areas.py
reformatted areas.py
All done!
1 file reformatted.
```

Con posterioridad podemos examinar el archivo **areas.py** para comprobar los cambios de estilo aplicados:

```
def area_triangulo(base, altura):
    return base * altura / 2


b = 5
h = 10
area = area_triangulo(b, h)
print("Área del triángulo:", area)
```

Para consultar información de ayuda de otras opciones de **black**:

```
$ black -h
```

Ir al índice del tutorial de Python

Publicado por Pherkad en 11:23

Etiquetas: Python3

Entrada más reciente Inicio Entrada antigua