# Ciclo 1
# Fundamentos de programación con Python
# Sesión 9: PEP 8  Guía de estilo para códigos Python

Programa Ciencias de la Computación e Inteligencia Artificial
Escuela de Ciencias Exactas e Ingeniería
Universidad Sergio Arboleda
Bogotá

# Agenda

1. Introducción
2. Sangría
3. Tabulador o espacio
4. Longitud máxima de línea
5. Líneas en blanco
6. Codificación del archivo fuente
7. Importaciones
8. Citas
9. Espacios en blanco
10. Otras recomendaciones
11. Cadenas de documentación

# 1. Introducción

- Esta guía de estilo evoluciona con el tiempo a medida que se identifican nuevas convenciones y las anteriores quedan obsoletas por los cambios en el propio lenguaje.

- Muchos proyectos tienen sus propias guías de estilo de codificación. En caso de conflicto, estas guías específicas del proyecto tienen prioridad para ese proyecto.

# 1. Introducción

- Esta guía de estilo evoluciona con el tiempo a medida que se identifican nuevas convenciones y las anteriores quedan obsoletas por los cambios en el propio lenguaje.
- Muchos proyectos tienen sus propias guías de estilo de codificación. En caso de conflicto, estas guías específicas del proyecto tienen prioridad para ese proyecto.
- Una guía de estilo es una cuestión 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.

# 2. Indentación (Sangría)

- Las líneas de continuación deben alinear los elementos envueltos ya sea verticalmente usando la unión implícita de líneas de Python dentro de paréntesis, corchetes y llaves, o usando una sangría colgante.

- Cuando se utiliza una sangría colgante se debe considerar lo siguiente; no debe haber argumentos en la primera línea y se debe utilizar una sangría adicional para distinguirse claramente como una línea de continuación.

# 2. Indentación (Sangría)

SI

```
# Aligned with opening delimiter.
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

# More indentation included to distinguish this from the rest.
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)

# Hanging indents should add a level.
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

NO

```
# Arguments on first line forbidden when not using vertical alignment.
foo = long_function_name(var_one, var_two,
    var_three, var_four)

# Further indentation required as indentation is not distinguishable.
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

# 3. Tabulador o espacios

- Los espacios son el método de sangría preferido.
- Los tabuladores deben utilizarse únicamente para mantener la coherencia con el código que ya está indentado con tabuladores.
- Python 3 no permite mezclar el uso de tabulaciones y espacios para la indentación.

# 4. Longitud máxima de la línea

- Limite todas las líneas a un máximo de 79 caracteres.
- En el caso de bloques de texto largos y fluidos con menos restricciones estructurales (docstrings o comentarios), la longitud de línea debería limitarse a 72 caracteres.
- La biblioteca estándar de Python es conservadora y requiere limitar las líneas a 79 caracteres (y los docstrings/comentarios a 72).
- La forma preferida de envolver líneas largas es utilizando la continuación de línea implícita de Python dentro de paréntesis, corchetes y llaves. Las líneas largas pueden dividirse en varias líneas envolviendo las expresiones entre paréntesis. Éstos deben usarse en lugar de usar una barra invertida para la continuación de línea.

# 4. Longitud máxima de la línea

- Las barras invertidas pueden ser apropiadas a veces. Por ejemplo, las declaraciones largas y múltiples que no pueden utilizar la continuación implícita, por lo que las barras invertidas son aceptables:

```
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())
```

# ¿Debe haber un salto de línea antes o después de un operador binario?

- Durante décadas, el estilo recomendado era romper después de los operadores binarios. Pero esto puede perjudicar la legibilidad de dos maneras: los operadores tienden a dispersarse por diferentes columnas de la pantalla, y cada operador se aleja de su operando y pasa a la línea anterior. En este caso, el ojo tiene que hacer un trabajo extra para saber qué elementos se suman y cuáles se restan:

```
# Yes: easy to match operators with operands
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

```
# No: operators sit far away from their operands
income = (gross_wages +
          taxable_interest +
          (dividends - qualified_dividends) -
          ira_deduction -
          student_loan_interest)
```

# 5. Líneas en blanco

- Rodea las definiciones de funciones y clases de nivel superior con dos líneas en blanco.
- Las definiciones de métodos dentro de una clase se rodean de una sola línea en blanco.
- Se pueden utilizar líneas en blanco adicionales (con moderación) para separar grupos de funciones relacionadas. Se pueden omitir las líneas en blanco entre un grupo de funciones relacionadas (por ejemplo, un conjunto de implementaciones ficticias).
- Utilice líneas en blanco en las funciones, con moderación, para indicar secciones lógicas.

# 6. Codificación del archivo fuente

- El código en el núcleo de la distribución de Python debería usar siempre UTF-8 (o ASCII en Python 2).
- Los archivos que usan ASCII (en Python 2) o UTF-8 (en Python 3) no deberían tener una declaración de codificación.
- En la biblioteca estándar, las codificaciones no predeterminadas deberían usarse sólo para propósitos de prueba o cuando un comentario o docstring necesite mencionar un nombre de autor que contenga caracteres no ASCII; de lo contrario, el uso de escapes \x, \u, \U, o \N es la forma preferida de incluir datos no ASCII en literales de cadena.

# 7. Importaciones

- Las importaciones deben ir normalmente en líneas separadas, por ejemplo:

```
import os
import sys
```

```
import os, sys
```

# 7. Importaciones

- Las importaciones se colocan siempre en la parte superior del archivo, justo después de los comentarios y docstrings del módulo, y antes de los globales y constantes del módulo.

- Las importaciones deben agruparse en el siguiente orden:

  - importaciones de la biblioteca estándar
  - importaciones de terceros relacionadas
  - importaciones específicas de la aplicación local/biblioteca

# 7. Importaciones

- Debe poner una línea en blanco entre cada grupo de importaciones.
- Se recomiendan las importaciones absolutas, ya que suelen ser más legibles y tienden a comportarse mejor (o al menos dan mejores mensajes de error) si el sistema de importación está mal configurado (como cuando un directorio dentro de un paquete acaba en sys.path):

```
import mypkg.sibling
from mypkg import sibling
from mypkg.sibling import example
```

# ➢ Nivel de módulo de los nombres de dunder

- Los "dunders" de nivel de módulo (es decir, nombres con dos guiones bajos iniciales y dos finales) como __all__, __author__, __version__, etc. deben colocarse después de la cadena de instrucciones del módulo pero antes de cualquier sentencia import, excepto de las importaciones __future__. Python ordena que las importaciones futuras deben aparecer en el módulo antes de cualquier otro código excepto los docstrings.

```
"""This is the example module.

This module does stuff.
"""

from __future__ import barry_as_FLUFL

__all__ = ['a', 'b', 'c']
__version__ = '0.1'
__author__ = 'Cardinal Biggles'

import os
import sys
```

# 8. Citas

- En Python, las cadenas con comillas simples y las cadenas con comillas dobles son lo mismo. Este PEP no hace ninguna recomendación al respecto. Elija una regla y aténgase a ella. Sin embargo, cuando una cadena contiene caracteres de comillas simples o dobles, utilice la otra para evitar las barras invertidas en la cadena. Esto mejora la legibilidad.

# 9. Espacios en blanco en expresiones y afirmaciones

- Evite los espacios en blanco en las siguientes situaciones:
- Inmediatamente dentro de paréntesis, corchetes o llaves:

Yes:

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

No:

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

# 9. Espacios en blanco en expresiones y afirmaciones

- Evite los espacios en blanco en las siguientes situaciones:
- Entre una coma final y un paréntesis siguiente:

Yes:

```
foo = (0,)
```

No:

```
bar = (0, )
```

# 9. Espacios en blanco en expresiones y afirmaciones

- Evite los espacios en blanco en las siguientes situaciones:
- Inmediatamente antes de una coma, punto y coma o dos puntos:

Yes:

```
if x == 4: print x, y; x, y = y, x
```

No:

```
if x == 4 : print x , y ; x , y = y , x
```

# 9. Espacios en blanco en expresiones y afirmaciones

- Sin embargo, en un corte los dos puntos actúan como un operador binario, y deben tener la misma cantidad a ambos lados (tratándolo como el operador de menor prioridad). En un corte extendido, ambos dos puntos deben tener la misma cantidad de espacio aplicada. Excepción: cuando se omite un parámetro de corte, se omite el espacio.

Yes:

```
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]
```

# 9. Espacios en blanco en expresiones y afirmaciones

- Evite los espacios en blanco en las siguientes situaciones:
- Inmediatamente antes del paréntesis abierto que inicia la lista de argumentos de una llamada a una función

Yes:

```
spam(1)
```

No:

```
spam (1)
```

# 9. Espacios en blanco en expresiones y afirmaciones

- Evite los espacios en blanco en las siguientes situaciones:
- Inmediatamente antes del paréntesis abierto que inicia una indexación o un corte

Yes:

```
dct['key'] = lst[index]
```

No:

```
dct ['key'] = lst [index]
```

# 9. Espacios en blanco en expresiones y afirmaciones

- Evite los espacios en blanco en las siguientes situaciones:
- Más de un espacio alrededor de un operador de asignación (u otro) para alinearlo con otro.

Yes:

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

No:

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

# 10. Otras recomendaciones

- Evite los espacios en blanco al final del texto. Como suele ser invisible, puede ser confuso: por ejemplo, una barra invertida seguida de un espacio y una nueva línea no cuenta como marcador de continuación de línea.
- Siempre rodea estos operadores binarios con un solo espacio a cada lado: asignación (=), asignación aumentada (+=, -= etc.), comparaciones (==, <, >, !=, <>, <=, >=, en, no en, es, no es), booleanos (y, o, no).

# 10. Otras recomendaciones

- Si se utilizan operadores con diferentes prioridades, considere la posibilidad de añadir un espacio en blanco alrededor de los operadores de menor prioridad. Use su propio criterio; sin embargo, nunca use más de un espacio, y siempre tenga la misma cantidad de espacio en blanco a ambos lados de un operador binario.

Yes:

```
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)
```

# 10. Otras recomendaciones

- No utilice espacios alrededor del signo = cuando se utilice para indicar un argumento de palabra clave o un valor de parámetro por defecto.

Yes:

```
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)
```

# 10. Otras recomendaciones

- Las anotaciones de función deben utilizar las reglas normales para los dos puntos y siempre tienen espacios alrededor de la flecha -> si está presente.

Yes:

```
def munge(input: AnyStr): ...
def munge() -> AnyStr: ...
```

No:

```
def munge(input:AnyStr): ...
def munge()->PosInt: ...
```

# 10. Otras recomendaciones

- Al combinar una anotación de argumento con un valor por defecto, utilice espacios alrededor del signo = (pero sólo para aquellos argumentos que tengan tanto una anotación como un valor por defecto).

Yes:

```
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): ...
```

# 10. Otras recomendaciones

- Por lo general, se desaconsejan las sentencias compuestas (varias sentencias en la misma línea).

Yes:

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

Rather not:

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

# ➢ Cuándo utilizar las comas finales

- Las comas finales son normalmente opcionales, excepto que son obligatorias cuando se hace una tupla de un elemento (y en Python 2 tienen semántica para la sentencia print). Para mayor claridad, se recomienda rodear estas últimas entre paréntesis (técnicamente redundantes).

Yes:

```
FILES = ('setup.cfg',)
```

OK, but confusing:

```
FILES = 'setup.cfg',
```

# Cuándo utilizar las comas finales

Cuando las comas finales son redundantes, suelen ser útiles cuando se utiliza un sistema de control de versiones, cuando se espera que una lista de valores, argumentos o elementos importados se amplíe con el tiempo. El patrón es poner cada valor (etc.) en una línea por sí mismo, añadiendo siempre una coma al final, y añadir el paréntesis/corche/corchete de cierre en la siguiente línea. Sin embargo, no tiene sentido tener una coma final en la misma línea que el delimitador de cierre (excepto en el caso anterior de las tuplas singleton).

Yes:

```
FILES = [
    'setup.cfg',
    'tox.ini',
    ]
initialize(FILES,
           error=True,
           )
```

No:

```
FILES = ['setup.cfg', 'tox.ini',]
initialize(FILES, error=True,)
```

# Comentarios

- Los comentarios que contradicen el código son peores que la ausencia de comentarios. Es prioritario mantener los comentarios actualizados cuando el código cambia.
- Los comentarios deben ser frases completas. Si un comentario es una frase u oración, su primera palabra debe ir en mayúsculas, a menos que se trate de un identificador que comience con una letra minúscula (¡nunca alteres el caso de los identificadores!).
- Si un comentario es corto, puede omitirse el punto al final. Los comentarios en bloque suelen consistir en uno o varios párrafos formados por frases completas, y cada frase debe terminar en un punto.
- Se deben utilizar dos espacios después del punto final de la frase.

# Comentarios del bloque

- Los comentarios en bloque suelen aplicarse a parte del código (o a todo el código) que les sigue, y están sangrados al mismo nivel que ese código. Cada línea de un comentario de bloque comienza con un # y un solo espacio (a menos que se trate de texto sangrado dentro del comentario).

- Los párrafos dentro de un comentario de bloque están separados por una línea que contiene un solo #.

# Comentarios en línea

- Utilice 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. Deben comenzar con un # y un solo espacio.
- Los comentarios en línea son innecesarios y, de hecho, distraen la atención si indican lo obvio.

Don't do this:

```
x = x + 1                 # Increment x
```

But sometimes, this is useful:

```
x = x + 1                 # Compensate for border
```

# 10. Cadenas de documentación

- Las convenciones para escribir buenas cadenas de documentación (también conocidas como "docstrings") están inmortalizadas en PEP 257.
- Escriba docstrings para todos los módulos, funciones, clases y métodos públicos. Los docstrings no son necesarios para los métodos no públicos, pero deberías tener un comentario que describa lo que hace el método. Este comentario debe aparecer después de la línea def.
- El PEP 257 describe buenas convenciones de docstrings. Tenga en cuenta que lo más importante es que el """ que termina un docstring de varias líneas debe estar en una línea por sí mismo, por ejemplo

```
"""Return a foobang

Optional plotz says to frobnicate the bizbaz first.
"""
```

El futuro digital es de todos

MinTIC

# Preguntas

UNIVERSIDAD SERGIO ARBOLEDA

MisiónTIC2022