PYTHON

Best Practice and Standarization

PEP 20 - The Zen of Python

PEP 20 - THE ZEN OF PYTHON

- PEP 20 – El Zen de Python
  - El Zen de Python es una colección de 19 aforismos que reflejan la filosofía detrás de Python, sus principios rectores y su diseño.
  - Tim Peters escribió este poema de 19 líneas en la lista de correo de Python en 1999, y se convirtió en la entrada n.° 20 en las Propuestas de mejora de Python en 2004.
  - Es uno de los huevos de Pascua incluidos en el intérprete de Python.
  - Escribir 'import this' en una consola interactiva de Python.

PEP 20 - THE ZEN OF PYTHON

```
Python 3.12.6 (tags/v3.12.6:a4a2d2b, Sep  6 2024, 20:11:23) [MSC v.1940
Type "help", "copyright", "credits" or "license" for more information.
>>> 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!
>>>
```

PEP 20 - THE ZEN OF PYTHON

- **(1) Lo bello es mejor que lo feo**
  - Python tiene ciertas reglas de estilo que se recomienda a los programadores que sigan. Estas son, entre otras cosas: una longitud máxima de línea de 79 caracteres, convenciones de nombres de variables, colocar declaraciones en líneas separadas y muchas otras.

PEP 20 - THE ZEN OF PYTHON

Example: Write a program that calculates the hypotenuse of a right-angled triangle.

✖

```
1 from math import sqrt
2 sidea = float(input("The length of the 'a' side:"))
3 sideb = float(input("The length of the 'b' side:"))
4 sidec = sqrt(a**2+b**2)
5 print("The length of the hypotenuse is", sidec )
```

✔

```
1 from math import sqrt
2
3 side_a = float(input("The length of the 'a' side: "))
4 side_b = float(input("The length of the 'b' side: "))
5 hypotenuse = sqrt(a**2 + b**2)
6
7 print("The length of the hypotenuse is", hypotenuse)
8
```

PEP 20 - THE ZEN OF PYTHON

- **(2) Explícito es mejor que implícito**
  - El código que escribas debe ser explícito y legible.
  - Siempre que quieras usar una característica implícita del lenguaje, pregúntate si realmente la necesitas. Tal vez haya una mejor manera de implementar la funcionalidad. Si no es así, piensa en dejar un comentario en el código para explicar qué está pasando, de modo que a otros programadores les resulte más fácil entender tu código.
  - En Python, es preferible usar no solo la forma más simple de expresar una idea de programación, sino también la más explícita, concreta y específica.
  - Por lo tanto, a veces es una buena idea agregar más "verbosidad" a tu código, ya que todo cuenta para la legibilidad. Dar nombres de variables y funciones que se expliquen por sí solos, o agregar más explicitud a las importaciones o argumentos de funciones puede ser una buena práctica.

PEP 20 - THE ZEN OF PYTHON

Example: Import `apples` and `bananas` from the *fruit.py* module.

```
1 from fruit import *
2
3 apples(2, 3.45)
4
```

```
1 from fruit import apples, bananas
2
3 apples(quantity=2, price=3.45)
4
```

PEP 20 - THE ZEN OF PYTHON

- **(3) Lo simple es mejor que lo complejo**
  - La simplicidad es la clave del éxito.
  - Normalmente se prefiere una solución más simple a una compleja y, por lo general, el enfoque minimalista gana.
  - Considera no adoptar un enfoque orientado a objetos cuando no sea necesario. Usa menos líneas de código si eso es posible.
  - Si necesita implementar una solución más compleja, divida los problemas en partes más pequeñas y simples.

PEP 20 - THE ZEN OF PYTHON

Example: Sort the `numbers` list in ascending order.

✖

```
1  import heapq
2
3  numbers = [-1, 12, -5, 0, 7, 21, 15, 1]
4  heapq.heapify(numbers)
5
6  sorted_numbers = []
7
8  while numbers:
9      sorted_numbers.append(heapq.heappop(numbers))
10
11 print(sorted_numbers)
12
```

✔

```
1  numbers = [-1, 12, -5, 0, 7, 21, 15, 1]
2  numbers.sort()
3
4  print(numbers)
5
```

PEP 20 - THE ZEN OF PYTHON

- **(4) Lo complejo es mejor que lo complicado**
  - Cuando no sea posible encontrar soluciones simples, tenga en cuenta las limitaciones que conlleva la simplicidad y utilice en su lugar soluciones complejas.
  - Distinguir entre lo complejo, que consta de muchos elementos, y lo complicado, que significa que es difícil de entender, es otro aspecto que se debe tener en cuenta al escribir código.
  - En otras palabras, hay ocasiones en las que se puede preferir una solución compleja a una simple, especialmente si esta última genera malentendidos, dudas o malas interpretaciones. Debe evitarlas.
  - Por otro lado, siempre se prefiere lo complejo a lo complicado. Cuando su código se vuelva grande y demasiado difícil de entender y comprender, divídalo en partes bien separadas, de modo que sea más fácil de gestionar y manejar.
  - Evitar malentendidos, falta de claridad y mala comprensión.

PEP 20 - THE ZEN OF PYTHON

Example: Perform five additions of two numbers.

✖

```python
first_number = int(input("Enter the first number: "))
second_number = int(input("Enter the second number: "))
addition_result = first_number + second_number
print(first_number, "+", second_number, "=", addition_result)
first_number = int(input("Enter the first number: "))
second_number = int(input("Enter the second number: "))
addition_result = first_number + second_number
print(first_number, "+", second_number, "=", addition_result)
first_number = int(input("Enter the first number: "))
second_number = int(input("Enter the second number: "))
addition_result = first_number + second_number
print(first_number, "+", second_number, "=", addition_result)
first_number = int(input("Enter the first number: "))
second_number = int(input("Enter the second number: "))
addition_result = first_number + second_number
print(first_number, "+", second_number, "=", addition_result)
first_number = int(input("Enter the first number: "))
second_number = int(input("Enter the second number: "))
addition_result = first_number + second_number
print(first_number, "+", second_number, "=", addition_result)
```

✔

```python
def addition(x, y):
    print(x, "+", y, "=", x+y)

for i in range(5):
    first_number = int(input("Enter the first number: "))
    second_number = int(input("Enter the second number: "))
    addition(first_number, second_number)
```

PEP 20 - THE ZEN OF PYTHON

- **(5) El código plano es mejor que el código anidado**
  - El código anidado hace que sea más difícil de seguir y comprender. Anidar dos o tres niveles de profundidad puede seguir siendo bueno, pero cualquier cosa más allá de eso se vuelve confuso e ilegible.
  - Aunque en Python puedes tener cualquier nivel de bucles anidados o declaraciones if, cualquier cosa por encima de tres debería ser una señal clara de que quizás sea un buen momento para comenzar a refactorizar tu código.
  - El código plano es más fácil de usar y se vuelve mucho más fácil de mantener.

PEP 20 - THE ZEN OF PYTHON

Example: Display a message whether or not x is within the range from 4 to 6.

✖

```
x = float(input("Enter a number: "))

if x > 0:
    if x > 1:
        if x > 2:
            if x > 3:
                if x >= 4:
                    if x <= 6:
                        print("x is a number between 4 and 6.")
else:
    print("x is not a number between 4 and 6.")

```

✔

```
x = float(input("Enter a number: "))

if x >= 4 and x <=6:
    print("x is a number between 4 and 6.")
else:
    print("x is not a number between 4 and 6.")

```

PEP 20 - THE ZEN OF PYTHON

- **(6) El código disperso es mejor que el denso**
  - No escriba demasiado código en una línea, no coloque demasiada información en una pequeña cantidad de código, no escriba líneas de código demasiado largas, utilice los espacios en blanco de manera responsable: todo esto afecta la legibilidad y la comprensión de su programa.
  - La forma más fácil y común de introducir la escasez en su código es introducir la anidación. Probablemente por eso este aforismo viene justo después del que nos dice que prefiramos el código plano al código anidado. La clave para la legibilidad es lograr un equilibrio entre los dos: reducir la anidación y luego tratar de reducir la densidad.

PEP 20 - THE ZEN OF PYTHON

Example: Print the message “Hello, World!” if the value passed to the x variable equals 1.

```
1 x = 1
2 if x == 1 : print("Hello, World!")
3
```

```
1 x = 1
2 if x == 1:
3     print("Hello, World!")
4
```

PEP 20 - THE ZEN OF PYTHON

- **(7) La legibilidad es importante**
  - Tu código no solo lo leen las computadoras, también lo leen los humanos. De hecho, es la esencia de la filosofía Python, y todo el diseño y la cultura de Python giran en torno a la afirmación de que "el código se lee con más frecuencia de la que se escribe" (Guido Van Rossum).
  - Todos los aforismos incluidos en PEP20 allanan el camino hacia la legibilidad, en mayor o menor medida, como uno de los factores más cruciales que se deben tener en cuenta al crear código. Siempre que sientas la tentación de renunciar a la legibilidad, recházala. No subestimes el poder de la legibilidad, especialmente cuando tienes que volver a tu código después de un tiempo o dejar el código para que otros lo desarrollen en el futuro.
  - Dar nombres significativos a las variables, funciones, módulos y clases; diseñar bloques de código de manera adecuada; usar comentarios cuando sea necesario; Mantener el código ordenado y elegante: todo esto contribuye a que el código sea legible y fácil de usar.
  - Recuerde: la legibilidad de su código refleja cuán responsable es como programador. No solo refleja bien la calidad del código, sino también a su autor.

PEP 20 - THE ZEN OF PYTHON

Example: Write a program that calculates a product's gross price.

```
def f(i):
    l = i + (0.08 * i)
    return l

```

```
# Calculates the gross price of products in Wonderland.

def calculate_gross_price(net_price):
    gross_price = net_price + (0.08 * net_price)
    return gross_price

```

PEP 20 - THE ZEN OF PYTHON

- **(8) Los casos especiales no son lo suficientemente especiales como para romper las reglas.**
  - La disciplina, la coherencia y el cumplimiento de las normas y convenciones son elementos importantes en el desarrollo de código profesional y responsable. No debería haber excepciones que nos permitan romper los principios que rigen las mejores prácticas de codificación.
  - Ningún caso especial, como la presión del tiempo o la complejidad de un problema determinado, debería ser una excusa para escribir código que no siga las pautas.
  - No se trata solo de legibilidad, aunque debería ser una de las primeras cosas en las que piense, sino también de ceñirse a las decisiones relacionadas con el diseño y el desarrollo que haya tomado, ya sea la coherencia que puede garantizar la compatibilidad con versiones anteriores, mantener las convenciones de nombres sin cambios o cualquier otra cosa.

PEP 20 - THE ZEN OF PYTHON

Example: Write a function that multiplies two numbers and a function that adds two numbers.

✖

```
1  def multiply_two_numbers(first_number, second_number):
2      return first_number * second_number
3
4  print(multiply_two_numbers(7, 9))
5
6
7  def addingTwoNumbers(firstNumber, secondNumber):
8      return firstNumber + secondNumber
9
10 print(addingTwoNumbers(7, 9))
11
```

✔

```
1  def multiply_two_numbers(first_number, second_number):
2      return first_number * second_number
3
4  print(multiply_two_numbers(7, 9))
5
6
7  def add_two_numbers(first_number, second_number):
8      return first_number + second_number
9
10 print(add_two_numbers(7, 9))
11
```

PEP 20 - THE ZEN OF PYTHON

- **(9)…Aunque la practicidad supera a la pureza**
  - ¿Se pueden romper las reglas o no?
  - El objetivo final es resolver problemas reales y escribir código que realice alguna tarea particular (esperada).
  - Si los posibles beneficios (por ejemplo, mejor rendimiento) son mayores que los posibles efectos negativos (por ejemplo, capacidad de mantenimiento afectada), los problemas de codificación del mundo real pueden encontrar una excusa para hacer una excepción a las reglas. **La practicidad entonces se vuelve más importante que la pureza.**
  - Si necesita escribir una línea de código de 85 caracteres porque dividirla en dos líneas separadas afecta la legibilidad, hágalo. Si necesita mantener la compatibilidad con código escrito previamente y usar CamelCase en lugar de snake_case, hágalo. **A veces hay que romper las reglas, hay que hacer excepciones.**

PEP 20 - THE ZEN OF PYTHON

- **(10) Los errores nunca deberían pasar desapercibidos…**
- **(11) "…A menos que se silencien explícitamente".**
  - A continuación, analice una situación potencialmente peligrosa:
    - Al no hacer conversión de tipo, el programa puede no hacer aquello que debe sin producir un error de ejecución inmediato.

```
number = input("Enter a number: ")
multiply_number_by_two = number * 2
print("Your number multiplied by two is:", multiply_number_by_two)
```

PEP 20 - THE ZEN OF PYTHON

- Haciendo el siguiente cambio el programa dará un error de tipo ValueError en la conversión.

```
number = int(input("Enter a number: "))
multiply_number_by_two = number * 2
print("Your number multiplied by two is:", multiply_number_by_two)
```

- Siempre es mejor que el programa 'proteste', indicando que algo ha salido mal que continuando y enmascarando los problemas.
- Es más fácil depurar un programa que falla que uno que silencia los errores.

PEP 20 - THE ZEN OF PYTHON

```
try:
    print(1/0)
except Exception as e:
    pass

```

An improved version, handling a specific kind of an error:

```
try:
    print(1/0)
except ZeroDivisionError:
    print("Don't divide by zero!")

```

```
try:
    number = int(input("Enter an integer number: "))
except:
    number = 0

```

PEP 20 - THE ZEN OF PYTHON

- **(12) Ante la ambigüedad, rechace la tentación de adivinar.**
  - Probar siempre su código antes de lanzarlo a producción e implementarlo para los clientes.
  - Probar su código le permite ahorrar tiempo, no perderlo. Si encuentra un error en una etapa temprana, le costará menos tiempo y dinero solucionarlo. Si no prueba su código y resulta que hay un error en una etapa avanzada del desarrollo, las correcciones pueden ser una tarea bastante costosa y que requiera mucho tiempo.
  - Otra cosa es que debe evitar escribir código ambiguo, lo que significa que no debe dejar lugar a conjeturas. Dé a sus variables nombres que se puedan comentar por sí mismos y deje comentarios donde sea necesario. Si está importando un módulo, haga que la importación sea explícita. Si un fragmento en particular es complejo o complicado, explique su funcionamiento. ¡Nunca deje comentarios ni utilice nombres que sean incorrectos, confusos o engañosos!
  - Si sospechas que hay algo incorrecto en el código que estás leyendo, o sientes que hay algo que no está claro en él, no adivines su funcionamiento, ¡pruébalo!

PEP 20 - THE ZEN OF PYTHON

Let's analyze the following example:

```
fun(1, 2, 3)
fun(a=1, b=2, c=3)
```

The two function invocations may be the same, but not necessarily. It's not possible to know without seeing the function definition. If the function definition is like the one below, the results could differ:

```
def fun(x=0, y=0, z=0, a=1, b=2, c=3):
    pass
```

PEP 20 - THE ZEN OF PYTHON

- **(13) Debería haber una forma obvia de hacerlo, preferiblemente solo una.**
- **(14) "Aunque puede que esa forma no sea obvia al principio, a menos que seas holandés".**
  - Puede haber múltiples formas de lograr el mismo objetivo. Por ejemplo, si quieres tomar el nombre y apellido del usuario y mostrarlos en la pantalla, puedes hacerlo de una de las siguientes maneras:

```
first_name = input("Enter your first name: ")
last_name = input("Enter your last name: ")

print("Your name is:", first_name, last_name)
print("Your name is:" + " " + first_name + " " + last_name)
print("Your name is: {} {}".format(first_name, last_name))
```

PEP 20 - THE ZEN OF PYTHON

- Es bueno seguir los estándares y convenciones de uso del lenguaje. Por ejemplo, si has estado usando snake_case para nombrar las variables en tu código hasta ahora, puede ser una mala idea comenzar a usar CamelCase para el resto de tu código dentro de un mismo programa. A menos que lo hagas con un propósito específico, y las ventajas de este enfoque sean mayores que las desventajas.
- Cada función, cada clase, cada método –cada entidad– debe tener una única responsabilidad cohesiva. ¿Por qué? Porque este enfoque te ayuda a obtener más claridad y a producir un código más limpio, hace que sea más fácil y económico mantenerlo, y menos vulnerable a errores.

PEP 20 - THE ZEN OF PYTHON

- **(15) Ahora es mejor que nunca**
- **(16) "Aunque nunca es mejor que ahora mismo."**
  - Python te permite traducir rápidamente tus ideas en código funcional. Siempre que experimentes el efecto eureka o tengas tu momento de inspiración, escribe tus pensamientos y codifícalos en Python. Luego puedes refinarlo, desarrollarlo o rediseñarlo muy fácilmente.
  - No existe nada perfecto. Puedes trabajar duro para acercarte a la perfección, refinar tu código, refactorizarlo varias veces, pero nunca será perfecto. Si cedes a la tentación de completar un programa y publicarlo solo cuando esté perfecto, es muy probable que nunca lo hagas. En ocasiones es mejor 'nunca' que 'ahora mismo'.
  - Si no puedes hacer algo necesario ahora, marca el código como obsoleto (DeprecationWarning) o como no implementado (NotImplementedError)
  - No sacar a producción el código hasta que esté listo, no cuando parezca que está listo.

PEP 20 - THE ZEN OF PYTHON

- **(17) Si la implementación es difícil de explicar, es una mala idea**
- **(18) "Si la implementación es fácil de explicar, puede ser una buena idea".**
  - Si puedes explicar lo que esperas de un programa, lo que quieres que haga, se puede diseñar. Si te resulta difícil explicar sus características y funcionalidades, puede ser una señal de que tal vez tu idea debería ser repensada y digerida nuevamente.
  - La simplicidad y el minimalismo son nuevamente las claves (aunque no deben afectar la legibilidad). Lo simple es mejor que lo complejo, pero lo complejo es mejor que lo complicado. **Cuanto más simple, mejor.**
  - Sin embargo, aunque algo sea fácil de explicar, **no** significa que sea bueno. Simplemente es más fácil juzgar si lo es o no.

PEP 20 - THE ZEN OF PYTHON

- **(19) Los espacios de nombres son una idea genial. ¡Hagamos más de eso!**
  - Python proporciona un mecanismo de espacio de nombres bueno y bien organizado para administrar la disponibilidad de los identificadores que desea utilizar y evitar conflictos con nombres ya existentes en diferentes ámbitos.
  - Cada ámbito es un espacio de nombres →Espacios de nombres locales y globales.
  - En un espacio de nombres sólo puede existir un elemento con un determinado nombre → Cada espacio de nombres es un diccionario con las variables del aquel.

PEP 20 - THE ZEN OF PYTHON

- Python los agrega implícitamente a un diccionario interno que reside dentro de un ámbito particular, es decir, la región de un programa Python donde los espacios de nombres son accesibles. Si desea acceder a esa variable, Python busca su nombre en el diccionario y devuelve el valor que se le pasa. Si la variable no existe y, por lo tanto, no se encuentra, se genera la excepción **NameError**.
- Funciones, clases, objetos, módulos, paquetes… todos son espacios de nombres. Este hecho da como resultado lo siguiente: un espacio de nombres más específico no puede ser alterado por un espacio de nombres menos específico, ya que residen dentro de dos ámbitos diferentes (por ejemplo, una variable local dentro de una función no influye en una variable global*). Sin embargo, un espacio de nombres más específico tiene acceso a un espacio de nombres menos específico (por ejemplo, se puede acceder a una variable global desde dentro de una función).

PEP 20 - THE ZEN OF PYTHON

- Nota: Usar la palabra clave global antes de una variable global dentro de la función es un mecanismo que le permite alterar esa variable, incluso aunque resida en un ámbito diferente (mala práctica).

Use the namespaces to make your code clearer and more readable. For example, do this:

✔

```
from instruments import guitars

guitars.fender(page)
guitars.ibanez(vai)
```

Instead of this:

✖

```
from instruments.guitars import fender, ibanez

fender(page)
ibanez(vai)
```