Python

Un lenguaje de programación interpretado cuya filosofía hace hincapié en la legibilidad de su código.

Autor:

Kevin Cárdenas.

2023

Índice

1. Introducción

Python es un lenguaje de programación interpretado cuya filosofía enfatiza la legibilidad del código. Es un lenguaje multiparadigma, ya que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional. Además, es un lenguaje interpretado, con tipado dinámico y multiplataforma.

Para hablar de código limpio en Python es esencial conocer *The Zen of Python*, que es un conjunto de principios para escribir código limpio en Python.

- *Beautiful is better than ugly.*

  Hermoso es mejor que feo. Se debe escribir código legible y entendible para cualquier persona con un mínimo de conocimiento en programación.

- *Explicit is better than implicit.*

  Explícito es mejor que implícito. Se debe escribir código que describa claramente su funcionalidad, utilizando nombres significativos para variables y funciones.

- *Simple is better than complex.*

  Simple es mejor que complejo. Se debe evitar la sobre-complicación, usando el menor número de líneas, funciones y clases necesarias para lograr el objetivo.

- *Complex is better than complicated.*

  Complejo es mejor que complicado. Aunque a veces el código puede ser complejo, debe ser siempre entendible.

- *Flat is better than nested.*

  Plano es mejor que anidado. Se debe evitar tener funciones dentro de funciones, clases dentro de clases, y anidar excesivamente condicionales o ciclos.

- *Sparse is better than dense.*

  Disperso es mejor que denso. Se debe modularizar el código, evitando tener

muchas líneas de código en un solo archivo o muchas responsabilidades en un solo módulo.

- *Readability counts.*

  La legibilidad cuenta. El código debe ser legible y entendible para cualquier persona con un mínimo de conocimiento en programación.

- *Special cases aren't special enough to break the rules.*

  Los casos especiales no son lo suficientemente especiales como para romper las reglas. Se debe evitar tener casos especiales que puedan romper el código.

- *Although practicality beats purity.*

  Aunque la practicidad vence a la pureza. Es preferible tener un código funcional a un código perfecto.

- *Errors should never pass silently.*

  Los errores nunca deben pasar silenciosamente. Se deben manejar los errores adecuadamente para evitar que pasen desapercibidos.

- *Unless explicitly silenced.*

  A menos que se silencien explícitamente. Si se decide silenciar los errores, debe hacerse de forma explícita.

- *In the face of ambiguity, refuse the temptation to guess.*

  Ante la ambigüedad, rechaza la tentación de adivinar. Se debe evitar la ambigüedad en el código para que funcione como se espera y sea fácil de entender.

- *There should be one– and preferably only one –obvious way to do it.*

  Debería haber una, y preferiblemente solo una, forma obvia de hacerlo. Se debe buscar la mejor, más simple y legible forma de hacer las cosas.

- *Although that way may not be obvious at first unless you're Dutch.*

  Aunque esa forma puede no ser obvia al principio a menos que seas holandés. La forma más obvia de hacer las cosas puede requerir práctica y experiencia para ser descubierta.

- *Now is better than never.*

  Ahora es mejor que nunca. Se debe evitar procrastinar al escribir código.

- *Although never is often better than* right *now.*

  Aunque nunca es a menudo mejor que *ahora* mismo.

- *If the implementation is hard to explain, it's a bad idea.*

  Si la implementación es difícil de explicar, es una mala idea. El código no debe ser difícil de explicar o entender.

- *If the implementation is easy to explain, it may be a good idea.*

  Si la implementación es fácil de explicar, puede ser una buena idea.

- *Namespaces are one honking great idea – let's do more of those!*

  Los espacios de nombres son una gran idea, ¡hagamos más de esos!

Es también recomendable seguir el PEP (Python Enhancement Proposal), que es una guía de estilo para escribir código en Python. PEP 8 es la guía de estilo que proporciona recomendaciones sobre cómo nombrar variables, funciones, clases, módulos, cómo escribir código, cómo indentar, cómo escribir comentarios, entre otros.

Además, es beneficioso escribir código *pythonic*, que es escribir código que sigue los principios de *The Zen of Python* y que adhiere al PEP 8.

1.1. PEP 8

El PEP 8 es una guía de estilo para escribir código en Python. Esta guía proporciona recomendaciones sobre cómo nombrar variables, funciones, clases, módulos, cómo escribir código, cómo indentar, cómo escribir comentarios, entre otros.

Sin embargo, esta guía también menciona que:

- La base de una guía de estilo es la 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.

Para identificar inconsistencias: en ocasiones, al escribir código, se puede sentir que algo no está bien, que algo no está bien escrito o nombrado. En ese momento se debe revisar el código y verificar si se está siendo inconsistente. A veces, las recomendaciones de las guías de estilo no son aplicables. Cuando esto sucede, se debe usar el sentido común. En particular: ¡no rompa la compatibilidad con versiones anteriores solo para cumplir con este PEP!

PEP 8 proporciona estas razones para ayudar a decidir si se debe ignorar una directriz en particular en su código:

- La aplicación de la directriz haría que el código fuera menos legible, incluso para alguien acostumbrado a leer código que sigue esta PEP.
- Para ser coherente con el código circundante que también lo rompe (tal vez por razones históricas) – aunque esto también es una oportunidad para limpiar el desorden de otra persona.
- Cuando el código en cuestión es anterior a la introducción de la directriz y no hay ninguna otra razón para modificarlo.
- Cuando el código necesita seguir siendo compatible con versiones anteriores de Python que no soportan la característica recomendada por la guía de estilo.

1.2. Variables y Convenciones de Nombres

Las variables son contenedores de información, y se pueden crear variables de diferentes tipos de datos. Las variables se pueden crear usando el signo igual (=), y el nombre de la variable debe seguir las convenciones de nombres de Python.

Las convenciones de nombres son un conjunto de reglas que se deben seguir para nombrar variables, funciones, clases, módulos, etc. En Python, las convenciones de nombres son las siguientes:

- **Variables y Funciones**: Los nombres de las variables y funciones deben ser en minúsculas, y si el nombre está compuesto por más de una palabra,

estas deben estar separadas por guiones bajos. A esto se le llama *snake case* (e.g., `mi_variable`, `mi_funcion()`).

- **Clases**: Los nombres de las clases deben iniciar con mayúsculas, y si el nombre de la clase está compuesto por más de una palabra, la primera letra de cada palabra debe ser mayúscula sin separaciones entre ellas. A esto se le llama *Pascal Case* (e.g., `MiClase`).

- **Módulos**: Los nombres de los módulos deben ser en minúsculas, y si el nombre del módulo está compuesto por más de una palabra, estas deben estar separadas por guiones bajos (e.g., `mi_modulo.py`).

- **Constantes**: Los nombres de las constantes deben ser en mayúsculas, y si el nombre de la constante está compuesto por más de una palabra, estas deben estar separadas por guiones bajos (e.g., `MI_CONSTANTE`).

Palabras Reservadas Es importante evitar el uso de palabras reservadas de Python al nombrar variables, funciones, clases, o módulos. Las palabras reservadas son aquellas que tienen un significado especial en Python, como `for`, `if`, `while`, `import`, etc.

Tipos de Datos y Conversiones Python tiene varios tipos de datos incorporados como `int`, `float`, `str`, `list`, `tuple`, `dict`, `set`, etc. Puedes convertir entre tipos de datos usando las funciones de conversión como `int()`, `float()`, `str()`, etc.

Variables Globales y Locales Las variables globales son aquellas definidas fuera de una función, mientras que las variables locales son definidas dentro de una función. Es importante entender el alcance (*scope*) de las variables para evitar errores en tu código.

Operadores de Asignación Los operadores de asignación son los siguientes:

- `=`: Asigna el valor de la derecha a la variable de la izquierda.

- `+=`: Suma el valor de la derecha a la variable de la izquierda y asigna el resultado a la variable de la izquierda.
- `-=`: Resta el valor de la derecha a la variable de la izquierda y asigna el resultado a la variable de la izquierda.
- `*=`: Multiplica el valor de la derecha a la variable de la izquierda y asigna el resultado a la variable de la izquierda.
- `/=`: Divide el valor de la derecha a la variable de la izquierda y asigna el resultado a la variable de la izquierda.
- `%=`: Divide el valor de la derecha a la variable de la izquierda y asigna el resto a la variable de la izquierda.
- `**=`: Eleva el valor de la izquierda a la potencia del valor de la derecha y asigna el resultado a la variable de la izquierda.
- `//=`: Divide el valor de la izquierda entre el valor de la derecha y asigna el resultado a la variable de la izquierda.

Por ejemplo

```
CONSTANTE = 10
def tabla_multiplicar(numero):
    for i in range(1, 11):
        print(f"{numero} x {i} = {numero * i}")

tabla_multiplicar(CONSTANTE) # imprime la tabla de ↩
    multiplicar del 10
```

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

print(suma(5, 10)) # imprime 15
```

```python
def factorial(numero):
    if numero == 0:
        return 1
    else:
        return numero * factorial(numero - 1)

print(factorial(5)) # imprime 120
```

```python
PERSONAS = {
    "Kevin": 20,
    "Juan": 30,
    "Maria": 40
    }
def busqueda(nombre):
    if nombre in PERSONAS:
        print(f"{nombre} tiene {PERSONAS[nombre]} anios")
    else:
        print(f"{nombre} no existe en la base de datos")
```