UD01.03: Primer Programa y Fundamentos del Lenguaje

Contenido

1. Guía de estilo: PEP 8

1.1. ¿Qué es PEP 8?

PEP 8 (Python Enhancement Proposal 8) es el documento oficial que define las convenciones de estilo para escribir código Python. Fue creado para asegurar que el código sea legible, coherente y fácil de mantener. Seguir el PEP 8 es una práctica estándar en la industria.

1.2. Reglas y convenciones principales de PEP 8

- **Indentación:** Usa 4 espacios por nivel.
- **Longitud de línea:** No superar los 79 caracteres.
- **Nombres:**
    - Variables y funciones: snake_case (ejemplo: nombre_completo).
    - Clases: CamelCase (ejemplo: MiClase).
    - Constantes: MAYUSCULAS_CON_GUIONES (ejemplo: PI_REDONDEADO).
- **Espacios:** Usa espacios en blanco alrededor de operadores binarios (ejemplo: a = b + c).
- **Comentarios:** Deben ser claros y concisos.

1.3. Herramientas para aplicar PEP 8

Existen herramientas que automatizan la verificación y el formateo de código:

- **Analizadores de código o Linters (pylint, flake8):** Analizan tu código y te avisan de errores de estilo.
- **Formateadores (black, autopep8):** Reorganizan tu código automáticamente para que cumpla con las reglas.

1.4. Beneficios de seguir PEP 8

- **Mejora la legibilidad:** Un código con un estilo consistente es más fácil de leer y entender.
- **Facilita la colaboración:** Permite que varios desarrolladores trabajen en un mismo proyecto sin que haya choques de estilo.
- **Reduce errores:** Un código claro es más fácil de depurar.

2. Estructura y Sintaxis Básica

2.1. Estructura general de un programa en Python

Un programa en Python está compuesto por:

1. **Importaciones:** Carga de módulos externos.
2. **Definiciones:** Creación de funciones, clases o constantes.
3. **Bloque principal:** La lógica de la aplicación que se ejecuta al iniciar el programa.

2.2. Los comentarios

Son una herramienta fundamental para documentar el código y hacerlo más comprensible para otros y para uno mismo en el futuro.

- **Comentarios de línea:** Se inician con # y el intérprete los ignora.

```
# Esto es un comentario
x = 10  # Asignación de un valor a la variable x
```

- **Comentarios multilínea (docstrings):** Se usan para documentar módulos, clases y funciones, y se escriben entre triples comillas (""" o ''').

```
"""
Este módulo realiza operaciones matemáticas básicas.
"""
def sumar(a, b):
    """Esta función devuelve la suma de dos números."""
    return a + b
```

2.3. La indentación en Python

La indentación no es opcional en Python, es parte de la sintaxis. A diferencia de otros lenguajes que usan llaves {}, Python utiliza la indentación para delimitar bloques de código.

- **¿Qué es la indentación?** Dejar espacios al principio de una línea para indicar que pertenece a una estructura superior (como una función o un bucle).
- **Reglas generales:**
    - Se utilizan **4 espacios por nivel** (PEP 8).
    - Una indentación incorrecta produce un error (IndentationError).
- **Verificación automática:** Los IDEs y editores de código (como PyCharm y VS Code) te avisan automáticamente de una indentación incorrecta.

2.4. Escritura flexible de instrucciones

- **Con barra invertida (\):** Para dividir una instrucción larga en varias líneas, se puede usar \ al final de cada línea.

```
resultado = 10 + \
    20 + \
    30
```

- **Con paréntesis, corchetes o llaves:** Es la forma más recomendada. Si la expresión está dentro de (), [] o {}, puedes dividirla en varias líneas sin usar \.

```
lista = [1,
    2,
    3]
```

- **Varias instrucciones en una línea (;):** Aunque es posible separar varias instrucciones con punto y coma, **esta práctica está desaconsejada** por razones de legibilidad.

3. Primeros pasos con el lenguaje

3.1. Palabras reservadas del lenguaje

Son palabras con un significado especial y no pueden ser usadas como identificadores (nombres de variables, funciones, etc.). Ejemplos: if, else, for, while, def, class, import, return.

Constantes predefinidas Son valores especiales del lenguaje que no se pueden modificar.

- True y False: Valores booleanos.
- None: Representa la ausencia de valor.

3.2. Funciones Integradas (Built-in) en Python

Python incluye un conjunto de **funciones integradas** (también conocidas como *built-in*). Estas funciones se pueden utilizar directamente sin necesidad de importar ningún módulo adicional. Su conocimiento es esencial para realizar tareas básicas y constituyen el fundamento para el aprendizaje del lenguaje.

Algunas de las funciones integradas más comunes son print(), input(), len(), type(), str() e int(). A continuación, se detallan las dos primeras, ya que son fundamentales para la interacción con el usuario y la visualización de información.

3.2.1. La función print()

La función print() es una de las herramientas más utilizadas en Python. Su propósito principal es **mostrar texto o el valor de las variables en la consola**.

Uso básico

Para utilizar print(), se coloca el elemento a mostrar entre paréntesis. Dicho elemento puede ser una cadena de texto (encerrada entre comillas simples o dobles) o una variable.

```
# Muestra un mensaje simple

print("¡Hola, mundo!")


# Muestra el valor de una variable

nombre = "Ana"

print(nombre)
```

Visualización de múltiples elementos

Es posible mostrar varios elementos de forma simultánea, separándolos con comas. Por defecto, la función print() los unirá con un espacio entre ellos.

```
edad = 30

print("Mi nombre es", nombre, "y tengo", edad, "años.")
```

El resultado será: Mi nombre es Ana y tengo 30 años.

Parámetros adicionales

La función print() admite dos **parámetros opcionales** que modifican su comportamiento:

- **sep (separador):** Permite especificar el carácter que se utilizará para separar los elementos a mostrar. Si este parámetro no se usa, el valor por defecto es un espacio (' ').

```
# Usando un guion como separador

print("Nombre", "Edad", "Ciudad", sep="-")

# Resultado: Nombre-Edad-Ciudad


# Usando una coma y espacio

print("Uno", "Dos", "Tres", sep=", ")

# Resultado: Uno, Dos, Tres
```

- **end (final):** Define el carácter que se debe añadir al final de la línea. Su valor por defecto es un salto de línea ('\n'). Al modificarlo, se puede lograr que las impresiones siguientes continúen en la misma línea.

```
print("Esto es la primera parte.", end=" ")

print("Y esta es la segunda.")

# Resultado: Esto es la primera parte. Y esta es la segunda.
```

Este parámetro es útil para generar impresiones que se actualizan de forma dinámica.

Formato de salida con print()

Además de los parámetros sep y end, hay tres formas principales de darle formato a las cadenas de texto que se muestran con print(). Cada una tiene sus propias ventajas y se usan para insertar valores de variables o expresiones dentro de una cadena.

1. El método .format() 🎨

Este método, que se aplica directamente a una cadena, usa **marcadores de posición** con llaves {}. En estas llaves, puedes indicar la posición de los argumentos que le pasas al método o usar nombres para una mayor claridad.

- **Por posición:** Los valores se insertan en el orden en que se pasan al método.

nombre = "Carlos"

edad = 25

print("Hola, mi nombre es {} y tengo {} años.".format(nombre, edad))

Resultado: Hola, mi nombre es Carlos y tengo 25 años.

- **Por nombre:** Puedes nombrar los marcadores para que el código sea más legible, especialmente con muchos valores.

producto = "teclado"

precio = 59.99

print("El {p} tiene un costo de ${pr}.".format(p=producto, pr=precio))

Resultado: El teclado tiene un costo de $59.99.

2. Cadenas con formato f-string (f-strings) ✨

Las **f-strings** (cadenas literales formateadas) son la forma más moderna y recomendada para dar formato en Python 3.6 y versiones posteriores. Son muy fáciles de leer y escribir. Para crearlas, solo tienes que poner una f delante de la cadena y colocar las variables o expresiones directamente dentro de las llaves {}.

- **Simplicidad y claridad:** No necesitas usar el método .format(). La sintaxis es muy intuitiva.

nombre = "Sofía"

```
ciudad = "Madrid"

print(f"Me llamo {nombre} y vivo en {ciudad}.")

# Resultado: Me llamo Sofía y vivo en Madrid.
```

- **Expresiones dentro de llaves:** Puedes realizar cálculos o llamar a funciones directamente dentro de las llaves.

```
lado = 5

print(f"Un cuadrado con lado {lado} tiene un área de {lado * lado}.")

# Resultado: Un cuadrado con lado 5 tiene un área de 25.
```

3. El operador % (formato de estilo C) ⚙️

Este es el método de formato más antiguo de Python. Aunque todavía se usa, se considera obsoleto para código nuevo. Funciona de forma similar al printf de C, usando símbolos como %s para cadenas y %d para números enteros, seguidos del operador % y una tupla con los valores.

- **Sintaxis:** Usa códigos de formato para cada tipo de dato.

```
nombre = "Pedro"

edad = 42

print("Mi nombre es %s y tengo %d años." % (nombre, edad))

# Resultado: Mi nombre es Pedro y tengo 42 años.
```

3.2.2. La función input()

La función input() permite **obtener información que el usuario introduce a través del teclado**. Cuando el programa ejecuta input(), se detiene y espera a que el usuario escriba algo y presione **Enter**.

Uso básico

El valor introducido por el usuario siempre se devuelve como una **cadena de texto** (str), independientemente del tipo de dato que se introduzca (números, letras, etc**.). Es importante tener en cuenta esta característica.** Cuando se estudien los tipos de datos se tendrán las herramientas necesarias para tratar correctamente el resultado obtenido.

```
# Solicita al usuario que escriba su nombre
nombre_usuario = input("Por favor, introduzca su nombre: ")
print("Hola,", nombre_usuario)
```

Parámetro prompt

Se puede pasar a input() un **mensaje opcional** (conocido como *prompt* o aviso) que se mostrará al usuario antes de que el programa espere la entrada. Esto facilita al usuario la identificación del tipo de dato que se espera.

```
# El texto "Introduzca su edad:" es el prompt
edad_str = input("Introduzca su edad: ")
print("Ha escrito:", edad_str)
```

3.3. Uso de main() y if __name__ == "__main__"

Aunque Python no impone una estructura rígida, este bloque de código es una de las convenciones más importantes y se considera una **mejor práctica** en la programación profesional. Su propósito es claro: definir un bloque de código que **solo se ejecutará** cuando el archivo se ejecute directamente, y no cuando sea importado como un módulo en otro archivo.

Para entender su importancia, piensa en la **modularidad**. A medida que las aplicaciones crecen, es común dividir el código en diferentes archivos, donde cada uno contiene funciones específicas que pueden ser reutilizadas por otros archivos.

El Problema de no Usarlo

Imagina que tienes un archivo llamado funciones_utiles.py con una función y una llamada de prueba al final:

```
# Archivo: funciones_utiles.py

def calcular_area_circulo(radio):
    return 3.14 * radio * radio

print("La función calcular_area_circulo está siendo ejecutada.")
print(f"Área de un círculo con radio 5: {calcular_area_circulo(5)}")
```

Si otro archivo (main_app.py) necesita usar esa función, simplemente lo importa:

```
# Archivo: main_app.py

import funciones_utiles
print("¡El programa principal está en marcha!")
```

Al ejecutar main_app.py, el resultado es inesperado:

```
La función calcular_area_circulo está siendo ejecutada.
```

Área de un círculo con radio 5: 78.5

¡El programa principal está en marcha!

El código de prueba de funciones_utiles.py se ejecutó automáticamente al ser importado, lo cual es un efecto secundario no deseado.

La Solución: El Bloque if __name__ == "__main__"

Python asigna una variable especial llamada __name__ a cada archivo. Su valor depende de cómo se use el archivo:

- Si el archivo es el programa principal que se está ejecutando, __name__ es igual a la cadena "__main__".
- Si el archivo es importado como un módulo, __name__ toma el nombre del archivo (ej. "funciones_utiles").

Al envolver el código de prueba dentro de la condición if __name__ == "__main__", el programa sabe cuándo debe ejecutar ese bloque y cuándo no.

```
# Archivo: funciones_utiles.py (VERSIÓN MEJORADA)


def calcular_area_circulo(radio):
    return 3.14 * radio * radio


if __name__ == "__main__":
    print("Este código solo se ejecuta si corres funciones_utiles.py directamente.")
    print(f"Área de un círculo con radio 5: {calcular_area_circulo(5)}")
```

Ahora, al ejecutar main_app.py de nuevo:

Archivo: main_app.py

import funciones_utiles

print("¡El programa principal está en marcha!")

El resultado es el esperado:

¡El programa principal está en marcha!

El código de prueba de funciones_utiles.py ya no se ejecuta. El programa puede acceder a la función calcular_area_circulo para usarla, pero sin ejecutar el resto del código que está diseñado solo para pruebas.

Beneficios Clave

- **Modularidad:** Permite que los archivos actúen como módulos reutilizables, separando el código de las pruebas de ejecución.
- **Reutilización:** Hace que tu código sea fácil de importar y usar en proyectos más grandes sin efectos secundarios no deseados.
- **Profesionalismo:** Es un estándar de la industria que demuestra una comprensión sólida de la arquitectura de aplicaciones en Python.