ACTIVIDADES

UD01.03: Primer Programa y Fundamentos del Lenguaje

Objetivo General: Aplicar las buenas prácticas de programación en Python, incluyendo la guía de estilo **PEP 8**, la indentación y la estructura básica de un script.

Contenido

Actividad 1: La máquina de formato

Se te ha entregado un script con errores de estilo, indentación y comentarios. Tu misión es convertir este código en un ejemplo de buenas prácticas.

- **Detecta** todos los problemas de acuerdo con las normas de **PEP 8**.
- **Corrige** el código aplicando las reglas adecuadas de indentación, espaciado y comentarios para que sea legible y claro.

```
def suma(a,b):
 return a+b #esta función suma
```

Actividad 2: Mi primer programa funcional

Crea tu primer script en Python siguiendo estas instrucciones, asegurándote de organizar el código de manera profesional.

- Crea una función llamada **saludar_usuario** que contenga la lógica principal.
- Dentro de esa función, pide al usuario su **nombre** usando input().
- Pide al usuario su **edad** usando input().
- Muestra un saludo personalizado usando una **f-string**. El saludo debe incluir el nombre del usuario y un mensaje calculando su edad en 10 años.
- Organiza tu código para que la función principal se ejecute solo cuando el archivo se ejecute directamente, usando la estructura if __name__ == "__main__":.

Aclaración para la Actividad 2

Para esta actividad, necesitarás usar dos conceptos clave que nos permiten interactuar con el usuario: **funciones integradas** y la **definición de funciones propias**.

Funciones Integradas

Python incluye una serie de funciones que puedes usar directamente, sin necesidad de definirlas. Son como herramientas listas para usar. Para esta actividad, usarás:

- print(): Muestra texto en la consola. Lo que escribas dentro de los paréntesis aparecerá en la pantalla.
    - **Ejemplo:**

```
print("¡Hola, mundo!")
```

- input(): Pide al usuario que introduzca un texto a través de la consola. El programa se detendrá hasta que el usuario escriba algo y presione Enter. El texto que el usuario introduzca se guardará como un valor.
    - **Ejemplo:**

```
nombre = input("¿Cómo te llamas? ")
```

Definición de Funciones Propias

Además de las funciones integradas, tú puedes crear tus propias funciones para organizar tu código. Una función es un bloque de código que realiza una tarea específica. Esto nos ayuda a mantener el código limpio y ordenado.

Para definir una función, usamos la palabra clave def seguida del nombre de la función, paréntesis y dos puntos (:). El código que forma parte de la función debe estar indentado con **4 espacios**.

- **Ejemplo:**

```
def saludar_usuario():
    # El código dentro de esta función se ejecutará
    # solo cuando la llamemos más tarde.
    print("Este código está dentro de una función.")

# Para que se ejecute, debes "llamarla" por su nombre:
saludar_usuario()
```

Actividad 3: El debate del estilo

En esta actividad, trabajarás en grupos para reflexionar sobre la importancia de las normas de estilo en un proyecto de desarrollo de software.

- **Reflexiona** y responde a la pregunta: ¿Qué problemas pueden surgir en un proyecto si los programadores no siguen las normas de estilo **PEP 8**?
- **Elabora un listado** con al menos tres consecuencias negativas y prepárate para debatirlas con el resto de la clase.

Actividad 4: Clasifica y corrige identificadores

Analiza la siguiente lista de identificadores de variables, clases y constantes. Tu tarea es identificar qué reglas de **PEP 8** no cumplen y corregirlos.

Mi_funcion

edadPersona

PIredondeado

usuario-Datos

Clase_alumno

- **Indica** qué reglas de **PEP 8** incumplen.
- **Corrige** cada caso para que cumpla con las normas de estilo.

Actividad 5: Refactorización final

Se proporciona un script con problemas de estilo y organización. Deberás aplicar todo lo aprendido para convertirlo en un código de calidad.

- Refactoriza el código aplicando las normas aprendidas (PEP 8, indentación y la estructura del programa principal).
- Documenta la función utilizando **docstrings** para explicar qué hace.
- Modifica la función para que reciba el nombre del usuario y un segundo dato (ej. su ciudad), y utiliza una **f-string** para mostrar un mensaje más elaborado.

```
def saludo(nombre):
 print("hola",nombre)
print("programa principal")
n= input("tu nombre?")
saludo(n)
```

Actividad 6: El formateador de datos

En esta actividad, te enfocarás en dominar las distintas formas de dar formato a la salida.

- Crea un programa que pida al usuario su **nombre**, un **producto** que haya comprado y el **precio** de ese producto.
- Muestra el resultado usando una **f-string**. La salida debe ser: [Nombre], gracias por tu compra. El [producto] tiene un precio de [precio]€.
- Muestra el resultado usando el método **.format()**.
- Muestra el resultado usando la **separación por comas** en la función print(), pero modifica el separador (sep) para que los elementos se unan con un guion.
- Organiza tu código en una función principal y usa el bloque if __name__ == "__main__":.

Actividad 7: Verificación y formateo guiado (usando extensiones)

En esta actividad, utilizarás tu editor de código (por ejemplo, Visual Studio Code) para corregir un script de forma asistida.

Pasos a seguir:

1. **Instala las extensiones:** Busca e instala las extensiones de **Pylint** y **Black** en tu editor de código. Estas herramientas se integrarán automáticamente.
2. **Crea el archivo:** Crea un nuevo archivo llamado codigo_sucio.py y copia el siguiente código en él:

```
def mi_funcion_ejemplo(a,b,c):
```

```
  resultado = a + b+ c
  return resultado


x = 10
y=20
z=30


print("la suma es: ", mi_funcion_ejemplo(x,y,z))
```

3. **Detecta los errores:** Observa el código. Deberías ver advertencias y errores de estilo subrayados. Pasa el cursor sobre ellos o revisa el panel de "Problemas" para ver qué te dice **Pylint**.
4. **Formatea el código:** Con el archivo abierto y guardado, utiliza el comando para formatear el documento (generalmente Shift + Alt + F en VS Code). Observa cómo **Black** corrige la mayoría de los errores de espaciado y estructura de forma automática.
5. **Corrige lo que queda:** Si **Pylint** aún muestra advertencias (por ejemplo, nombres de variables), corrígelas manualmente hasta que no queden errores.

Actividad 8: Verificación y formateo autónomo (desde la terminal)

En esta actividad, serás tu propio asistente. Deberás instalar y usar herramientas de línea de comandos para realizar la misma tarea que en el ejercicio anterior.

Pasos a seguir:

1. **Instala las herramientas:** Abre tu terminal y ejecuta los siguientes comandos para instalar **Flake8** y **autopep8**:

```
pip install flake8 autopep8
```

2. **Crea el archivo:** Crea un nuevo archivo llamado codigo_manual.py y copia el siguiente código, que tiene los mismos problemas:

```
def mi_funcion_ejemplo(a,b,c):
  resultado = a + b+ c
  return resultado
```

```
x = 10

y=20

z=30


print("la suma es: ", mi_funcion_ejemplo(x,y,z))
```

3. **Verifica los errores:** Desde la terminal, navega hasta la carpeta donde guardaste el archivo y ejecuta **Flake8** para que analice el código:

```
flake8 codigo_manual.py
```

Analiza la salida de la terminal. Te mostrará el tipo de error (por ejemplo, E225), la línea y la columna donde se encuentra.

4. **Corrige los errores:** Usa **autopep8** para corregir automáticamente los errores de estilo.

```
autopep8 --in-place codigo_manual.py
```

5. **Verifica la corrección:** Vuelve a ejecutar flake8 para asegurarte de que todos los errores de estilo se han corregido. Si quedan errores (como nombres de variables mal elegidos), deberás corregirlos manualmente.