**Tarea dada**

*"Genera un script en Python que recorra una lista de números, filtre los números pares y los almacene en un archivo llamado pares.txt."*

**Proceso de generación de código**

**1. Comprensión del problema**

El agente primero analiza los requisitos:

* Recorrer una lista de números.
* Filtrar los números pares.
* Guardar los resultados en un archivo de texto llamado pares.txt.

**2. Identificación de patrones y reglas**

* Uso de listas y estructuras de control (for, if).
* Aplicación del operador módulo % para filtrar números pares.
* Escritura en un archivo usando open() con el modo "w".

**3. Generación del código**

Aquí está el código generado en respuesta a la tarea:

python

CopiarEditar

# Lista de números de ejemplo

numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Filtrar los números pares

pares = [num for num in numeros if num % 2 == 0]

# Guardar en un archivo

with open("pares.txt", "w") as archivo:

for num in pares:

archivo.write(f"{num}\n")

print("Los números pares han sido guardados en pares.txt")

**4. Explicación del enfoque**

* **Uso de listas y comprensión de listas:** Se usa una lista por comprensión para filtrar los números pares (pares = [num for num in numeros if num % 2 == 0]).
* **Estructura de control:** Se usa if num % 2 == 0 para seleccionar los números pares.
* **Uso de archivos:** Se utiliza with open("pares.txt", "w") as archivo: para escribir los números filtrados en un archivo de texto, asegurando que el archivo se cierre correctamente después de la operación.

**Conclusión**

Este ejemplo muestra cómo el agente sigue un proceso estructurado:

1. **Analiza la tarea.**
2. **Identifica patrones y reglas aplicables.**
3. **Genera un código.**

**Ejemplo: Generación de código en respuesta a la tarea dada**

**Consigna dada al agente:**

*"Genera código --- creando una función que permita al usuario ingresar descripciones de tareas pendientes y que se agreguen a una lista de tareas."*

**Análisis del agente**

1. Se necesita una función que reciba una descripción y la agregue a una lista.
2. Se debe implementar un bucle que permita al usuario ingresar múltiples tareas.
3. Es conveniente ofrecer la opción de salir del bucle.

**Código generado por el agente**

python

CopiarEditar

# Lista para almacenar las tareas pendientes

tareas\_pendientes = []

def agregar\_tarea(descripcion):

"""Agrega una tarea a la lista de tareas pendientes."""

if descripcion.strip(): # Evita agregar tareas vacías

tareas\_pendientes.append(descripcion)

print(f"Tarea agregada: {descripcion}")

else:

print("La descripción no puede estar vacía.")

def ingresar\_tareas():

"""Permite al usuario ingresar tareas en un bucle hasta que escriba 'salir'."""

while True:

tarea = input("Ingresa una tarea (o escribe 'salir' para terminar): ")

if tarea.lower() == "salir":

break

agregar\_tarea(tarea)

print("\nLista de tareas pendientes:")

for i, tarea in enumerate(tareas\_pendientes, 1):

print(f"{i}. {tarea}")

# Ejecución del programa

ingresar\_tareas()

**Análisis de patrones y enfoques utilizados**

**🔹 Patrones utilizados**

1. **Funciones modulares:**
   * agregar\_tarea(descripcion): Encapsula la lógica para agregar tareas.
   * ingresar\_tareas(): Gestiona la interacción con el usuario en un bucle.
2. **Estructuras de control:**
   * Un while True permite la entrada continua de tareas hasta que el usuario escriba "salir".
   * Un if filtra descripciones vacías para evitar errores.
3. **Listas para almacenamiento:**
   * Se utiliza tareas\_pendientes = [] para guardar las tareas de manera ordenada.
4. **Feedback inmediato al usuario:**
   * Se imprimen mensajes de confirmación cuando una tarea es agregada.
   * Se muestra la lista completa al finalizar la entrada de tareas.

**Reflexión sobre eficiencia y calidad del código**

✅ **Eficiencia:**

* El código es **modular**, lo que facilita la reutilización y mantenimiento.
* Evita agregar elementos vacíos, previniendo errores.

✅ **Calidad del código:**

* Es claro, legible y sigue buenas prácticas de programación.
* Se asegura de que el usuario reciba retroalimentación en cada paso.
* Es fácilmente extensible: se podría integrar con bases de datos o archivos en el futuro.

**Discusión abierta**

🔍 **Preguntas para reflexionar:**

1. ¿El enfoque del agente fue el que esperaban o hubo alguna sorpresa?
2. ¿Qué mejoras o funcionalidades adicionales agregarían?
3. ¿Cómo podría mejorarse la interacción con el usuario?

**1️Cómo el agente aborda la tarea**

El agente sigue un enfoque estructurado:  
✅ **Uso de with open()**: Asegura que el archivo se cierre automáticamente, evitando fugas de recursos.  
✅ **División de palabras con split()**: Utiliza una solución sencilla y efectiva para contar palabras.  
✅ **Manejo de errores**: Incluye excepciones para archivos inexistentes (FileNotFoundError) y otros errores inesperados.

**Patrones seguidos:**

🔹 *Simplicidad*: Usa funciones pequeñas y directas.  
🔹 *Buenas prácticas*: Manejo adecuado de archivos y excepciones.  
🔹 *Legibilidad*: Código claro y fácil de entender.

**2️⃣ Reflexión sobre las capacidades del agente**

📌 **Fortalezas en la generación de código:**  
🔹 *Estructura limpia y lógica* 🧹: La solución es directa y eficiente.  
🔹 *Aplicación de reglas estándar* 📏: Usa convenciones comunes en Python.  
🔹 *Adaptabilidad* ⚡: Puede extenderse fácilmente para más funciones (como contar líneas o caracteres).

💡 **Casos en los que el agente destaca:**  
✅ Resolviendo tareas bien definidas, como procesamiento de archivos.  
✅ Implementando soluciones con errores mínimos.  
✅ Escribiendo código estructurado y modular.

**3️⃣ Limitaciones del agente**

🚧 **Casos en los que podría fallar o generar código subóptimo:**  
❌ *No maneja archivos grandes eficientemente*: Leer archivos grandes de una sola vez (file.read()) puede consumir mucha memoria.  
✅ **Solución alternativa:** Usar lectura por líneas para mayor eficiencia:

python

CopiarEditar

def contar\_palabras\_archivo(archivo):

try:

with open(archivo, 'r') as file:

cantidad\_palabras = sum(len(line.split()) for line in file)

return cantidad\_palabras

except FileNotFoundError:

return "El archivo no fue encontrado"

except Exception as e:

return f"Error inesperado: {str(e)}"

💡 Esto reduce el consumo de memoria, útil para archivos grandes.

❌ *No considera puntuación ni caracteres especiales*: "Hola," y "Hola" se cuentan como palabras separadas.  
✅ **Solución alternativa:** Usar re.findall() para contar solo palabras alfabéticas:

python

CopiarEditar

import re

def contar\_palabras\_archivo(archivo):

try:

with open(archivo, 'r') as file:

contenido = file.read()

palabras = re.findall(r'\b\w+\b', contenido) # Extrae solo palabras

return len(palabras)

except FileNotFoundError:

return "El archivo no fue encontrado"

except Exception as e:

return f"Error inesperado: {str(e)}"

📌 Esto mejora la precisión al ignorar puntuaciones.

**🔍 Conclusión**

El agente **es bueno generando código funcional y bien estructurado**, pero **podría no ser óptimo en eficiencia y precisión sin ajustes adicionales**.  
📢 **Discusión:** ¿Qué mejoras agregarían? ¿Qué situaciones podrían hacer que el agente falle? 🚀