Nombre del Proyecto: Gestor Inteligente de Clientes (GIC).

Estudiante Benjamín Alonso Carmona Vega
Trabajo módulo 4
Nombre del Proyecto: Gestor Inteligente de Clientes (GIC).

Introducción

En este documento se realizará el resumen del trabajo por el desarrollo módulo 4, para el programa escrito en lenguaje Python, nombrado “Gestor Inteligente de Clientes (GIC).”

Se incluirán explicaciones acotadas del desarrollo, funcionalidades, capturas de log gestionando datos en el sistema, también el diagrama en UML.

Se registra el enlace https://github.com/Benj11ii/proyecto-modulo-4/tree/main respaldo realizado en plataforma github.

Contexto de la situación (basado en documento pdf)

Situación inicial

Unidad solicitante: Empresa SolutionTech (Startup de tecnología)
La empresa SolutionTech ha detectado la necesidad de mejorar su sistema de gestión de clientes mediante una solución escalable e integrable con otros servicios. Actualmente, los datos de los clientes se administran manualmente, lo que ha derivado en ineficiencias, duplicación de datos y problemas de seguridad.
La propuesta consiste en desarrollar una plataforma integral de gestión de clientes en Python, basada en POO y capaz de manejar distintos tipos de clientes, validaciones avanzadas, persistencia de datos y una interfaz de usuario básica. Además, la solución permitirá integración con APIs externas para validación de datos y envío de notificaciones automatizadas.

Nuestro objetivo 📋
Desarrollar una plataforma en Python basada en POO, que permita gestionar clientes, realizar validaciones en línea e integrarse con servicios externos, asegurando una estructura modular, escalable y segura.

Archivos python parte del proyecto.

Para lograr este objetivo considerando puntos a evaluar, he creado una carpeta llamada proyecto-modulo-4, ahí contengo los siguientes archivos:
main.py : archivo inicial y principal para comenzar a utilizar el sistema.
menu_inicial.py : Archivo que contiene funciones y lógica además del menú para navegar dentro del sistema.
modelos.py : En este archivo contengo la clase padre y las que heredan funciones, detalles de setter y getter, validadores y registro de actividad, además del funcionamiento para el esqueleto que permite crear el archivo json.
gestor_archivos.py : En este archivo guardo el funcionamiento del proceso para guardar, cargar, además de tener algunos clientes de ejemplo (proceso de pruebas que estuve realizando mientras desarrollaba el sistema).
conexion_api_externa.py : En este archivo, he implementado la gestión de validación básica y envío de mail, para cuando se agrega un cliente por ejemplo.

Paradigma de Orientación a Objetos

● Documento explicativo sobre POO y su importancia en sistemas escalables.

La Programación Orientada a Objetos es como organizar un taller de herramientas. En lugar de tener tornillos, martillos y planos tirados por todos lados (programación desordenada), guardamos cada elemento en su propia caja etiquetada.

En nuestro sistema GIC, estas "cajas" se llaman Clases como Cliente (clase padre) o RegistroActividad. Cada clase tiene:
Datos (Atributos): Como el nombre, rut o email.
Acciones (Métodos): Como mostrar_perfil() o generar_factura().

¿Por qué es importante para sistemas escalables?
La escalabilidad significa que el sistema puede crecer sin romperse. Gracias a la POO, el código es escalable porque:
Está en orden: Si falla el cálculo de un descuento, se sabe exactamente dónde buscar (en la clase ClientePremium), sin afectar al resto del programa.
Reutilización: Existe la validación del email una sola vez en la clase padre Cliente, y automáticamente ClienteRegular, Premium y Corporativo la usan.
Facilidad de crecimiento: Si mañana la empresa quiere un "Cliente VIP", solo se crea una nueva clase hija sin tocar lo que ya funciona.

● Implementación de la clase Cliente con atributos básicos.

La clase Cliente funciona como el plano maestro o molde principal. Define lo que todo cliente debe tener, sin importar su tipo.
En el archivo (modelos.py), se define atributos básicos y se protege los datos importantes:

```
class Cliente:
    def __init__(self, id_cliente, nombre, email, telefono, direccion):
        self.id_cliente = id_cliente
        self.nombre = nombre
        self._email = None  ##Definir inicialmente, para luego usar con setter de forma correcta
        self._telefono = None ##Definir privado como email
        self._direccion = None ##Definir privado igual que email y telefono
        self.activo = True
        self.logs: List[RegistroActividad] = []
        self.email = email ## llamar al setter
        self.telefono = telefono ## llamar al setter
        self.direccion = direccion ## llamar al setter
```

El método __init__ es el constructor: se ejecuta automáticamente al crear un cliente nuevo para "llenar" sus datos iniciales.
Se usa getters y setters (como @property) para que nadie pueda guardar un email sin @ o un teléfono con letras.

● Ejemplo práctico de instanciación y uso de métodos.

Instanciación:
"Instanciar" es simplemente la acción de crear un objeto real usando el molde (la clase). Es cuando pasas del plano a la casa construida.
En el archivo menu_inicial.py, esto ocurre cuando el usuario elige una opción:

 # Ejemplo de Instanciación (Creación del objeto)
 # Aquí nace "nuevo", que es un objeto real de tipo ClienteRegular
nuevo = ClienteRegular("aaa_111", "Benjamin", "benja@mail.com", "56912345678", "Calle Uno 123", 15)

Uso de métodos:
Una vez que el objeto nuevo existe, podemos pedirle que haga cosas usando sus métodos. Por ejemplo, mostrar sus datos o guardar un historial:

 # 1. Usar un método para mostrar información
print(nuevo.mostrar_perfil())
 # Resultado: "Tipo: Regular | ID: 123_abc | Nombre: Benjamin... | Puntos: 15"

 # 2. Usar un método para registrar una acción interna
nuevo.agregar_log("CREACIÓN", "Cliente creado desde el menú")

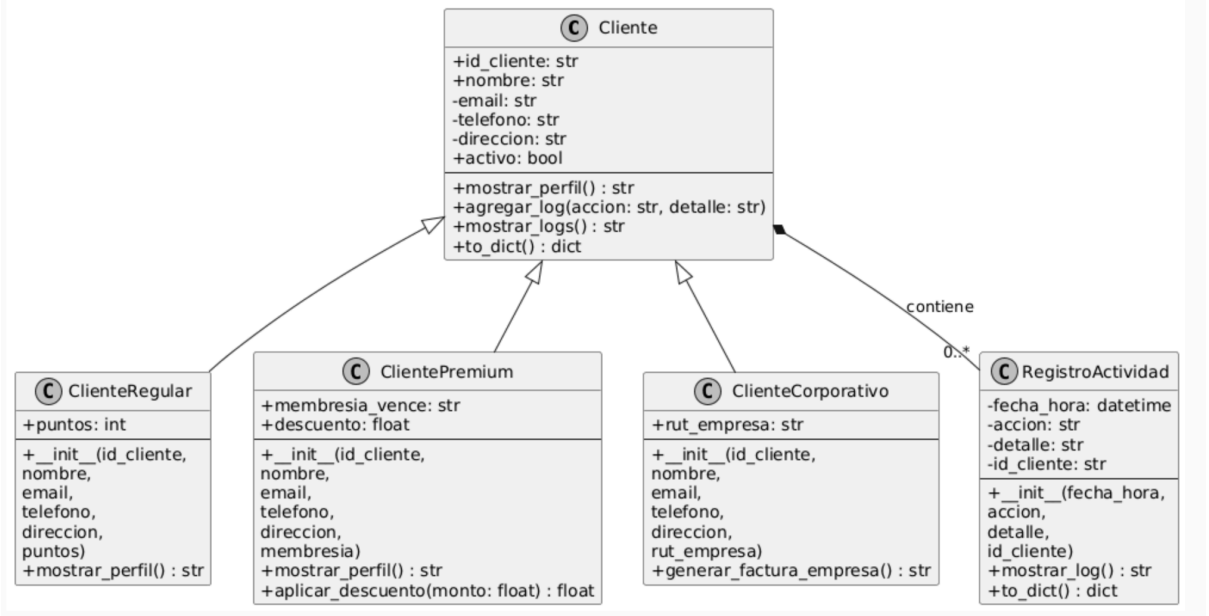
● Creación de métodos especiales (__str__, __eq__).

__str__ (String/Texto):
Sin esto: Si se hace print(cliente), Python muestra algo no decorado mensaje estándar como <modelos.Cliente object at 0x00....
Con esto: Si se hace print(cliente), Python mostrará automáticamente: Benjamin (ID: 123).
Utilidad: Facilita la depuración y los logs rápidos.

__eq__ (Equality/Igualdad):
Sin esto: Python piensa que dos clientes son distintos aunque tengan el mismo ID, porque ocupan lugares distintos en la memoria.
Con esto: Se enseña a Python que si el id_cliente es el mismo, entonces son la misma persona.
Utilidad: Sirve para evitar duplicados. Si intentas agregar un cliente a una lista y se usa if nuevo_cliente in lista_clientes, este método es el que trabaja por detrás.

Diagramas de Clase

● Creación del modelo UML detallado con relaciones entre clases.
(contenido como archivo para una mejor visualización)



● Representación de composición, herencia y agregación.

```
@startuml
skinparam classAttributeIconSize 0
skinparam monochrome true
skinparam shadowing false

class Cliente {
   +id_cliente: str
   +nombre: str
   -email: str
   +telefono: str
   +direccion: str
   +activo: bool
   +mostrar_perfil(): str
   +agregar_log(accion: str, detalle: str)
   +mostrar_logs(): str
   +to_dict(): dict
}

class ClienteRegular {
   +puntos: int
   +__init__(id_cliente, nombre, email, telefono, direccion, puntos)
   +mostrar_perfil(): str
}

class ClientePremium {
   +membresia_vence: str
   +descuento: float
```

```
    +__init__(id_cliente, nombre, email, telefono, direccion, membresia)
    +mostrar_perfil(): str
    +aplicar_descuento(monto: float): float
}

class ClienteCorporativo {
    +rut_empresa: str
    +__init__(id_cliente, nombre, email, telefono, direccion, rut_empresa)
    +generar_factura_empresa(): str
}

class RegistroActividad {
    -fecha_hora: datetime
    -accion: str
    -detalle: str
    -id_cliente: str
    +__init__(fecha_hora, accion, detalle, id_cliente)
    +mostrar_log(): str
    +to_dict(): dict
}

' Relaciones
Cliente <|-- ClienteRegular
Cliente <|-- ClientePremium
Cliente <|-- ClienteCorporativo
Cliente *-- "0..*" RegistroActividad : contiene

@enduml
```

● Uso de herramientas UML para documentación.

Emplee la herramienta web llamada PlantUML para e esquema

Descripción de Clases
Cliente (Clase Base): Actúa como la matriz del sistema. Contiene los datos comunes y la lógica de gestión de estados (activo/inactivo) y logs.

Subclases (Regular, Premium, Corporativo): Especializaciones del cliente que añaden comportamiento único (puntos, descuentos o facturación industrial).

RegistroActividad: Clase auxiliar encargada de la inmutabilidad de los eventos realizados por cada cliente.

B. Relaciones Técnicas
Herencia (Generalización): Representada por la flecha con triángulo blanco (<|--). Indica que los tres tipos de clientes "son" un Cliente y heredan todos sus métodos y atributos.

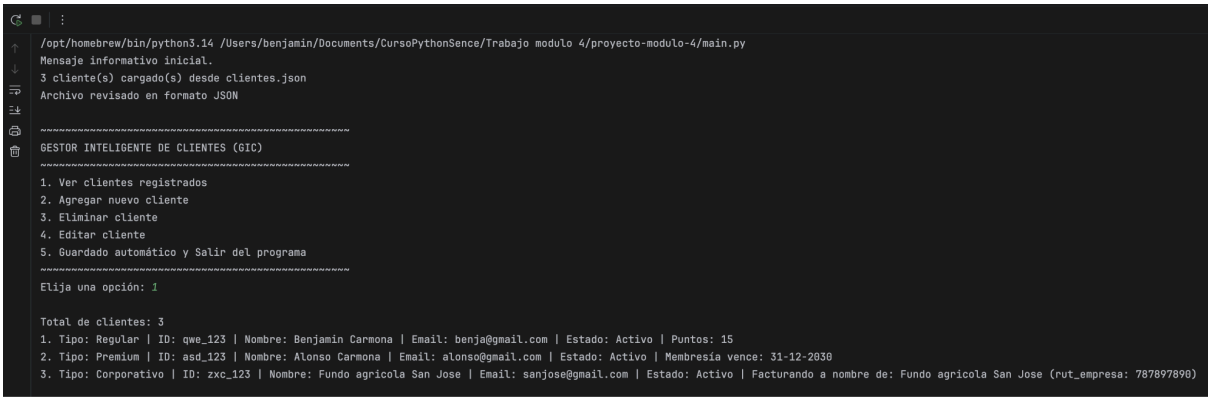
Composición: Representada por el rombo negro (*--). Indica una relación fuerte: un RegistroActividad pertenece a un Cliente. Si el cliente se elimina, sus registros de actividad pierden sentido en este contexto.

Encapsulamiento: Se utilizan atributos privados (con el signo - como -email o -fecha_hora) para asegurar que la información sensible sólo se modifique a través de métodos controlados (Setters/Getters).
3. Tips de Notación
+: Atributo/Método público (accesible desde cualquier parte).

-: Atributo/Método privado (solo la clase misma puede verlo).

0..*: Multiplicidad. Indica que un cliente puede tener desde cero hasta infinitos registros de actividad.
-

Capturas de código y ejecución del sistema.



Pruebas_programa_GIC_Log_005 copy.txt

```
49    Email: dificil@.
50
51    --- Validando email con API externa ---
52    Conectando con servicio de validación...
53    Formato de email inválido
54    Debe ingresar un email válido. Intente nuevamente.
55
56    Email: email@gmail.com
57
58    --- Validando email con API externa ---
59    Conectando con servicio de validación...
60    Email validado exitosamente por API externa
61    Teléfono (ej: 56912345678): 56939393939
62    Dirección (mínimo 10 caracteres): calle gente buena 123, Santiago
63    Puntos iniciales (0 por defecto): 0
64    Cliente Regular agregado
```

Conclusiones

He podido desarrollar un sistema funcional que cumple CRUD, dado que se puede agregar, editar (actualizar) y eliminar clientes, contener datos para respaldo y mantener coherencia que implica función normal en un sistema real.
Si bien no he implementado directamente API externa, he desarrollado una simulación suficiente para dar cuenta con la conexión y uso habitual en validación y envío de correo electrónico. Pero también agregué una API externa, el el archivo dedicado con la línea de código:
API_URL = "https://rapid-email-verifier.fly.dev/api/validate".

Manejo de excepciones y validadores de correo, teléfono y dirección también cumplen con la norma básica y estándar.
He gestionado encapsulamiento, polimorfismo y herencia de clases, en el código, para su escalabilidad y mantenimiento,

He podido aplicar lo aprendido en clases para programar este sistema de registro de clientes, con integridad de datos y manejo de archivo json además de las solicitudes explicadas en documentos.