**A blue logo with a globe and text

Description automatically generatedUNIVERSIDAD GERARDO BARRIOSA logo with a person in the center

Description automatically generated**

SAN MIGUEL

FACULTAD DE CIENCIA Y TECNOLOGIA

Técnico en ingeniería en sistemas y Redes Informáticas

Ciclo II

Programación III

Ingeniero. Willian Alexis Montes Girón

PRIMER PARCIAL

Presentado Por:

Katerin Michelle Campos Aparicio SMTR036323

Dinora Verenice Funes Lemus SMTR091923

AGOSTO 2024

**Ejercicio 1**

**Clase Producto** Para poder manejar diferentes productos en la tienda, cada producto necesita tener un nombre y un precio.

**Clase Cliente** Para almacenar su nombre y una lista de los productos que han comprado.

**Método agregar\_compra:** Añade un producto a la lista de compras del cliente.

**Método calcular\_total:** Calcula el total de todos los productos que el cliente ha comprado.

**Método calcular\_vuelto:** Calcula cuánto cambio (vuelto) debe recibir el cliente después de pagar, asegurando que solo se devuelva el vuelto si el pago es suficiente.

**Clase Proveedor** Se guarda su nombre, el producto que ofrecen, y el precio sugerido para venderlo.

**Función ingresar\_producto** se pide al usuario que ingrese el nombre y el precio.

**Función ingresar\_cliente** Se pide al usuario que ingrese los datos del cliente y los productos que ha comprado.

**Función ingresar\_proveedor:** registrar a un proveedor en el sistema.

Se usa Bucle (while): Para pedir al usuario que siga agregando productos hasta que decida detenerse.

Clases (class):

Se usan para definir estructuras que representan entidades del sistema, como Producto, Cliente, y Proveedor. Las clases agrupan datos (atributos) y comportamientos (métodos) relacionados.

Métodos (def):

Son funciones definidas dentro de las clases que permiten realizar acciones sobre los objetos, como agregar compras (agregar\_compra), calcular totales (calcular\_total), o calcular el vuelto (calcular\_vuelto).

**Ejercicio 2**

Razonamiento y Justificación del Código

1. Planteamiento del Problema

El problema presentado es registrar la asistencia de estudiantes en un colegio privado de manera eficiente. Cada docente tiene una lista de estudiantes y debe registrar si cada estudiante asistió a clase, tuvo permiso o se ausentó. Además, si un estudiante tiene un permiso, se debe registrar la razón. El sistema también debe permitir al director revisar las asistencias de los docentes.

2. Enfoque Utilizado

Para abordar esta solución, se utilizó la Programación Orientada a Objetos (POO) en Python, junto con algunas funcionalidades básicas del lenguaje. La decisión de usar POO y otras características de Python se justifica de la siguiente manera:

a. Programación Orientada a Objetos (POO)

Razón:

Modelado Natural de Entidades: La POO permite modelar entidades del mundo real como clases en el código. En este caso, se crearon las clases Estudiante, Asistencia, Docente y Director para representar los conceptos de estudiantes, asistencia, docentes y directores, respectivamente. Esto facilita la gestión de datos y el comportamiento asociado a cada entidad.

Justificación de Clases:

Estudiante: Representa a un estudiante con atributos como nombre y una lista de asistencias. Cada estudiante tiene múltiples registros de asistencia, que se gestionan en esta clase.

Asistencia: Representa el registro de una asistencia con atributos para la fecha, el estado (Asistió, Permiso, Inasistencia) y una posible razón para el permiso.

Docente: Representa al docente que maneja una lista de estudiantes y registra sus asistencias. Incluye métodos para agregar estudiantes, registrar asistencias y mostrar registros.

Director: Representa al director que revisa las asistencias de los docentes. Tiene el método para revisar y mostrar las asistencias registradas.

Ventajas:

Encapsulamiento: Permite agrupar datos y métodos relacionados en una misma estructura, haciendo el código más modular y fácil de mantener.

Reusabilidad: Las clases permiten crear instancias (objetos) y reutilizar código para gestionar diferentes entidades de manera consistente.

b. Funcionalidades de Python Utilizadas

Razón:

Métodos de Clase: Los métodos dentro de las clases permiten encapsular la lógica relacionada con cada entidad. Por ejemplo, el método registrar\_asistencia en la clase Docente maneja el proceso de registro de asistencia para los estudiantes del docente.

Listas y Enumeraciones: Se utilizaron listas para almacenar múltiples objetos (estudiantes, asistencias) y se empleó la función enumerate para iterar sobre listas con índices, lo que facilita la presentación de opciones.

Entrada del Usuario: Se utilizaron funciones de entrada (input()) para recibir datos del usuario, lo que permite la interacción en tiempo real y la personalización del registro de asistencia.

Justificación de Funcionalidades:

Listas: Permiten manejar dinámicamente múltiples estudiantes y asistencias sin necesidad de definir un número fijo.

Enumerate: Facilita la visualización y selección de opciones de manera ordenada.

Entrada del Usuario: Permite la flexibilidad de ingresar datos en tiempo real y registrar la información de manera interactiva.

c. Proceso de Solución

Modelado de Entidades:

Se identificaron las entidades principales (Estudiantes, Asistencia, Docente, Director) y se modelaron como clases.

Definición de Métodos:

Cada clase tiene métodos que encapsulan la lógica específica para manejar sus datos. Por ejemplo, registrar\_asistencia en Docente maneja el registro de asistencia para cada estudiante.

Inicialización y Ejecución:

Se crean instancias de las clases y se utilizan métodos para realizar operaciones como agregar estudiantes, registrar asistencias y revisar registros.

Interacción con el Usuario:

El código permite al usuario ingresar datos (fecha, nombre del estudiante, estado de asistencia) y visualizar la información registrada de manera clara y organizada.

Conclusión

La decisión de utilizar POO y las funcionalidades básicas de Python se basa en la necesidad de representar entidades del mundo real de manera estructurada y eficiente, manejar datos dinámicamente y permitir una interacción fácil y flexible con el usuario. La POO proporciona un enfoque natural y modular para gestionar las entidades del sistema, mientras que las funcionalidades de Python permiten una implementación práctica y accesible.

**Ejercicio 3**

El ejercicio plantea la necesidad de desarrollar un sistema para un hotel de playa que permita a un recepcionista gestionar el proceso de registro de clientes, la selección de habitaciones, la elección de servicios extras, y la generación de facturas detalladas para cada cliente. Este sistema debe ser capaz de manejar múltiples habitaciones y servicios, así como registrar los datos personales de los clientes.

Razonamiento y Solución

Uso de Programación Orientada a Objetos (POO)

La POO fue la elección principal para este ejercicio debido a varias razones:

1. Modelado Natural del Problema: El problema se presta perfectamente para ser modelado con objetos. Cada entidad clave del sistema, como una Habitacion, un Cliente, y un ServicioExtra, tiene propiedades y comportamientos claramente definidos. Por ejemplo, una habitación tiene un tipo y un precio, y un cliente puede seleccionar una habitación, agregar servicios extras, y solicitar una factura.

2. Reusabilidad y Modularidad: Al crear clases para cada entidad, el código se vuelve más modular y reutilizable. Esto significa que las clases pueden ser extendidas o modificadas en el futuro sin necesidad de reescribir grandes partes del código. Por ejemplo, si se agregan nuevos tipos de servicios o habitaciones, el código puede ser fácilmente actualizado.

3. Facilidad de Mantenimiento: La POO permite encapsular el comportamiento dentro de las clases, lo que facilita la comprensión y el mantenimiento del código. Cada clase maneja su propia lógica, lo que hace que el sistema sea más fácil de entender y modificar.

Estructura del Código

1. Clase Habitacion:

o Planteamiento: Modela las habitaciones del hotel, cada una con un tipo y un precio.

o Razonamiento: Esta clase encapsula la información relevante de una habitación, lo que simplifica la gestión de las opciones disponibles para los clientes.

2. Clase Cliente:

o Planteamiento: Representa a los clientes del hotel, manejando sus datos personales, la habitación seleccionada, y los servicios extras.

o Razonamiento: Al agrupar la información y comportamientos relacionados con el cliente en una sola clase, se facilita la gestión de los datos y la interacción con otras partes del sistema, como las habitaciones y servicios extras.

3. Clase ServicioExtra:

o Planteamiento: Modela los servicios adicionales que los clientes pueden solicitar.

o Razonamiento: Esta clase permite agregar fácilmente nuevos servicios sin alterar la estructura del cliente o del hotel, promoviendo la extensibilidad del sistema.

4. Funciones seleccionar\_habitacion y seleccionar\_servicios\_extras:

o Planteamiento: Facilitan la interacción del usuario con el sistema, permitiéndole seleccionar una habitación y agregar servicios extras.

o Razonamiento: Estas funciones separan la lógica de interacción con el usuario del resto del sistema, haciendo que el código sea más modular y fácil de probar.

5. Generación de Factura:

o Planteamiento: El cliente puede solicitar una factura detallada que incluye la habitación seleccionada, el número de noches, los servicios extra, y el costo total.

o Razonamiento: Este enfoque permite que todos los cálculos y la presentación de la información se realicen en un solo lugar, mejorando la claridad y la organización del código.

Justificación de Decisiones

• POO: Se utilizó la POO para crear un sistema más estructurado, claro, y fácil de mantener. La naturaleza del problema se prestaba a un diseño basado en clases, donde cada clase representa una entidad con sus propios datos y comportamientos.

• Funciones para Interacción: Las funciones seleccionar\_habitacion y seleccionar\_servicios\_extras se crearon para separar la lógica de negocio de la lógica de interacción, haciendo que el código sea más modular y fácil de modificar.

• Cálculos y Facturación: Centralizar los cálculos en el método calcular\_total y la generación de la factura en generar\_factura asegura que todos los detalles financieros se manejen de manera coherente y en un solo lugar, facilitando la verificación y la modificación si es necesario.

Conclusión

Este enfoque estructurado y orientado a objetos no solo facilita el desarrollo inicial, sino que también permite la fácil expansión y mantenimiento del sistema a medida que se requieran nuevas características o modificaciones. La separación clara entre las entidades, la lógica de negocio, y la interacción con el usuario asegura que el sistema sea robusto, flexible, y sostenible en el tiempo.

**Ejercicio 4**

Una empresa cuenta con dos tipos de empleados: aquellos con plaza fija y aquellos que trabajan por horas. Se han registrado los datos de ambos tipos y, al generar la planilla de pago, se realizan dos cálculos diferentes. A los empleados de plaza fija se les paga el salario base más comisiones, mientras que a los empleados por horas se les paga en función de la cantidad de horas trabajadas. Adicionalmente, si un empleado ha laborado más de 5 años, sin importar su tipo de contrato, se le otorga un bono adicional.

Solución Implementada

1. Clase Base Empleado:

Planteamiento:

La clase Empleado sirve como base para los diferentes tipos de empleados en la empresa. Esta clase almacena la información común a todos los empleados, como su nombre, edad y años trabajados en la empresa. Además, incluye un método para calcular un bono adicional si el empleado ha trabajado más de cinco años.

Razonamiento:

La elección de usar una clase base permite centralizar la lógica común a todos los empleados, evitando la duplicación de código y facilitando el mantenimiento. La POO se utilizó aquí para aprovechar la herencia, de modo que las subclases (empleados con plaza fija y empleados por horas) puedan heredar atributos y métodos comunes y solo implementar los detalles específicos.

Funcionalidades:

• Encapsulación: Permite que los atributos como nombre, edad, y anios\_trabajados estén encapsulados dentro de la clase, protegiendo y organizando la información de manera clara.

• Herencia: La herencia de la clase Empleado en subclases como EmpleadoPlazaFija y EmpleadoPorHoras permite la reutilización del código y la fácil extensión del sistema.

2. Subclase EmpleadoPlazaFija:

Planteamiento:

Esta subclase se deriva de la clase Empleado y está diseñada específicamente para empleados con plaza fija. Además de los atributos heredados, esta clase introduce los atributos salario\_base y comisiones, que son únicos para este tipo de empleados. El método calcular\_pago suma el salario base, las comisiones y el bono por antigüedad para determinar el pago total.

Razonamiento:

El uso de una subclase específica para los empleados con plaza fija permite que la lógica de cálculo del pago se mantenga clara y específica para este tipo de empleado, evitando mezclarla con la lógica de otros tipos de empleados. Esto mejora la organización del código y facilita futuras modificaciones o expansiones.

Funcionalidades:

• Polimorfismo: Aunque la clase Empleado define un método abstracto calcular\_pago, es en la subclase EmpleadoPlazaFija donde este método se implementa concretamente, mostrando cómo cada tipo de empleado tiene su propia forma de calcular el pago.

3. Subclase EmpleadoPorHoras:

Planteamiento:

Esta subclase, también derivada de Empleado, representa a los empleados que trabajan por horas. Aparte de los atributos heredados, introduce horas\_trabajadas y tarifa\_por\_hora para calcular el pago. El método calcular\_pago multiplica las horas trabajadas por la tarifa por hora, añadiendo el bono por antigüedad si corresponde.

Razonamiento:

La creación de esta subclase permite manejar la lógica específica de los empleados por horas, manteniendo el código modular y fácil de entender. Se evita así la complicación de gestionar diferentes tipos de empleados en una sola clase, lo que podría conducir a un código menos claro y más propenso a errores.

Funcionalidades:

• Herencia: Aprovecha la herencia de Empleado para mantener un código limpio y organizado.

• Polimorfismo: El método calcular\_pago es implementado aquí de manera específica para el cálculo basado en horas trabajadas, mostrando la flexibilidad de la POO.

4. Funciones de Registro de Empleados:

Planteamiento:

Se han creado dos funciones (registrar\_empleado\_fijo y registrar\_empleado\_por\_horas) para ingresar los datos de los empleados, permitiendo así interactuar con el usuario para obtener la información necesaria para cada tipo de empleado.

Razonamiento:

Estas funciones se utilizaron para mantener el código principal limpio y modular. Al separar la lógica de ingreso de datos, se asegura que el código sea más fácil de leer y mantener. Además, esto permite modificar o extender el proceso de ingreso de datos sin afectar otras partes del sistema.

Funcionalidades:

• Modularidad: La división del proceso de registro en funciones específicas mejora la claridad del código y facilita su mantenimiento y expansión.

• Interactividad: Permite que el programa sea dinámico y adaptable, solicitando información al usuario en lugar de manejar datos estáticos.

Justificación del Uso de POO

La Programación Orientada a Objetos (POO) fue elegida para esta solución debido a varias razones:

1. Organización: La POO facilita la organización del código, permitiendo que diferentes tipos de empleados se gestionen mediante subclases que heredan de una clase base. Esto hace que el código sea más legible y fácil de mantener.

2. Reusabilidad: La herencia y el polimorfismo permiten reutilizar y extender el código sin duplicación, lo que reduce errores y facilita futuras expansiones del sistema.

3. Flexibilidad: La POO permite definir métodos específicos en cada subclase (calcular\_pago en este caso), lo que ofrece una gran flexibilidad para manejar diferentes tipos de cálculos de manera clara y estructurada.

Conclusión

La solución implementada utiliza la POO para gestionar diferentes tipos de empleados en una empresa, aprovechando la herencia, la encapsulación, y el polimorfismo para mantener el código modular, extensible y fácil de mantener. La interactividad del programa se logra mediante funciones específicas para la entrada de datos, lo que también contribuye a un código organizado y adaptable.