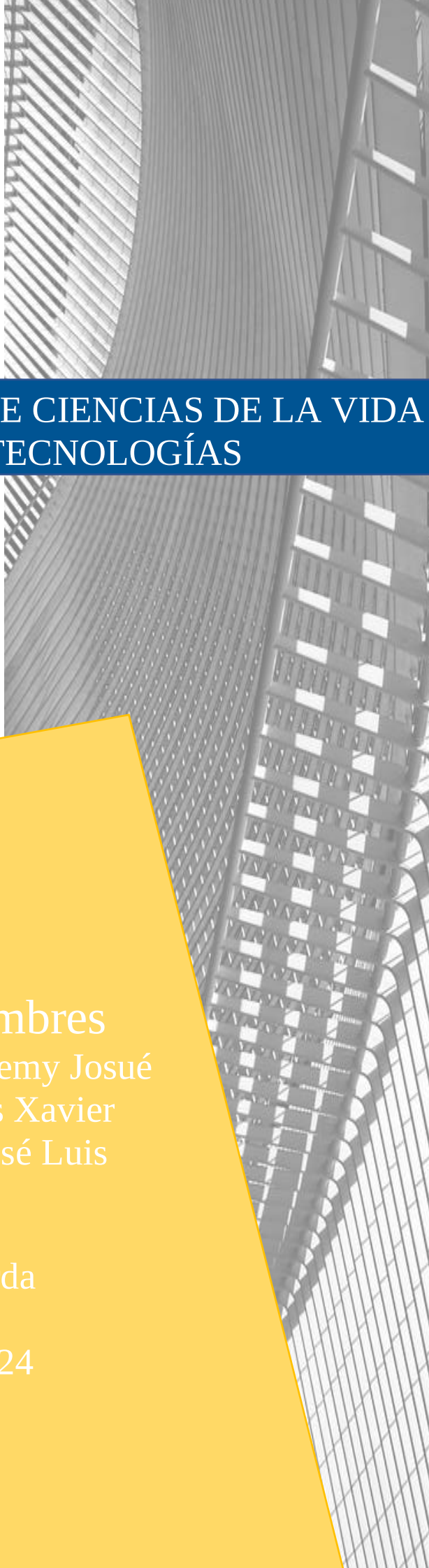


Uleam

FACULTAD DE CIENCIAS DE LA VIDA Y TECNOLOGÍAS

MODELADO ORIENTADO A OBJETOS

Apellidos y nombres

Delgado Solorzano Jeremy Josué

López Macias Derlis Xavier

Sarabia Calderón José Luis

Docente:

Anthony Legarda

2024-I

Octubre del 2024

Composición entre Usuario y Cuenta

Sí, los archivos que has compartido implementan una **relación de composición** entre las clases Usuario y Cuenta. A continuación, te explico por qué:

1. **Composición**: La composición implica que un objeto (en este caso, Usuario) es responsable de la existencia del otro objeto (Cuenta). Si el objeto contenedor (usuario) se elimina, los objetos contenidos (cuentas) también dejan de existir.

2. **Creación de las instancias de Cuenta dentro de Usuario**:

    - En el archivo user_controller.py, cada vez que se agrega una cuenta mediante el método agregar_cuenta, se crea una nueva instancia de Cuenta dentro del método:

Python
Copiar código
```
nueva_cuenta = Cuenta(id_cuenta, tipo_cuenta, nombre_cuenta, saldo_inicial)
self.__cuentas.append(nueva_cuenta)
```

3. Esto implica que las cuentas solo existen en el contexto del usuario y son manejadas a través de él.

4. **Control del ciclo de vida de las cuentas**:

    - El usuario también tiene métodos para **eliminar** cuentas, como se observa en:

Python
Copiar código
```
def eliminar_cuenta(self, nombre_cuenta):
    for cuenta in self.get_cuentas():
        if cuenta.get_nombre_cuenta() == nombre_cuenta:
            self.__cuentas.remove(cuenta)
            print(f"Cuenta {nombre_cuenta} eliminada exitosamente.")
            return
```

5. Esto indica que el usuario controla la creación, mantenimiento y eliminación de las cuentas, reforzando la composición.

6. **Acceso y encapsulamiento**:

    - El atributo __cuentas está definido como privado dentro de la clase Usuario, lo que asegura que solo se pueda modificar a través de los métodos del usuario:

Python
Copiar código
```
self.__cuentas = []
```

Esta implementación es un ejemplo claro de **composición**, ya que las cuentas no existen de manera independiente fuera de los objetos Usuario y su ciclo de vida está completamente ligado al del usuario que las contiene.

Agregación entre cuenta y transacción

1. Definición de Agregación

La agregación implica que un objeto (en este caso, Cuenta) contiene referencias a otros objetos (Transaccion), pero estos objetos pueden existir de forma independiente del contenedor. La destrucción del contenedor no implica necesariamente la destrucción de los contenidos.

2. Implementación de Agregación en tus Archivos

En el archivo de la clase Cuenta (account_controller.py), las transacciones se almacenan en una lista dentro de cada cuenta:

python
Copiar código
self.__transacciones = []

Las transacciones se agregan a la lista utilizando el método agregar_transaccion:

python
Copiar código
```
def agregar_transaccion(self, transaccion):
    self.__transacciones.append(transaccion)
    if transaccion.tipo_transaccion == "Ingreso":
        self.__saldo += transaccion.get_monto()  # Aumenta el saldo
    elif transaccion.tipo_transaccion == "Gasto":
        self.__saldo -= transaccion.get_monto()  # Disminuye el saldo
    print(f"Transacción '{transaccion.get_descripcion()}' añadida a la cuenta
{self.__nombre_cuenta}.")
```

Esto muestra que la cuenta guarda una referencia a cada transacción sin ser dueña de su ciclo de vida.

3. Independencia de las Transacciones

En el archivo transaction_controller.py, las instancias de Transaccion no dependen de la cuenta para existir. Pueden crearse de forma independiente y registrarse en varias cuentas si es necesario:

python
Copiar código
```
transaccion = TransaccionIngreso(100, "Sueldo", "2024-10-19", "Pago mensual")
cuenta.agregar_transaccion(transaccion)
```

Además, las transacciones se pueden eliminar o consultar sin que eso afecte a la instancia de la cuenta:

python
Copiar código
```
cuenta.eliminar_transaccion(0)
```

4. Conclusión

Esta estructura cumple con los principios de agregación porque las transacciones se gestionan como entidades separadas que solo están referenciadas por las cuentas, pero no dependen de ellas para su existencia. Esto es una distinción clara de la composición, donde los objetos contenidos no pueden existir sin su contenedor.

Por lo tanto, la relación entre Cuenta y Transaccion es de agregación.

Composición entre Transaccion y Informe

Comportamiento de la composición:

1. **Creación del Informe**:
    - Cuando creas una instancia de Transaccion, aún no existe el informe. Pero cuando registras una transacción (usando registrar_transaccion), dentro de ese método se crea un **nuevo informe** específico para esa transacción.
    - Ejemplo: Si registras una transacción de "compra", la transacción crea un informe que contiene los detalles del gasto relacionado con esa compra.

python
Copiar código

```
def registrar_transaccion(self, lista_transacciones):
    lista_transacciones.append(self)
    # Aquí se crea el informe para esta transacción
    self.__informe = Informe('gastos' if self.__categoria == 'gasto' else 'ingreso')
    self.__informe.generar_informe([self], self.__fecha, self.__fecha)
```

    - El Informe generado se crea **dentro** de la transacción y está relacionado directamente con los datos de la misma (como el monto, la categoría, etc.).

2. **Destrucción del Informe**:
    - Como el Informe está contenido dentro de la clase Transaccion (en el atributo __informe), su ciclo de vida está completamente ligado a la transacción. Esto significa que **cuando una instancia de Transaccion se destruye**, el Informe también se destruye automáticamente, ya que es un atributo de la transacción y no puede existir por separado.
    - No necesitas gestionar la destrucción manualmente; en Python, el recolector de basura destruirá tanto la transacción como su informe cuando ya no sean referenciados.

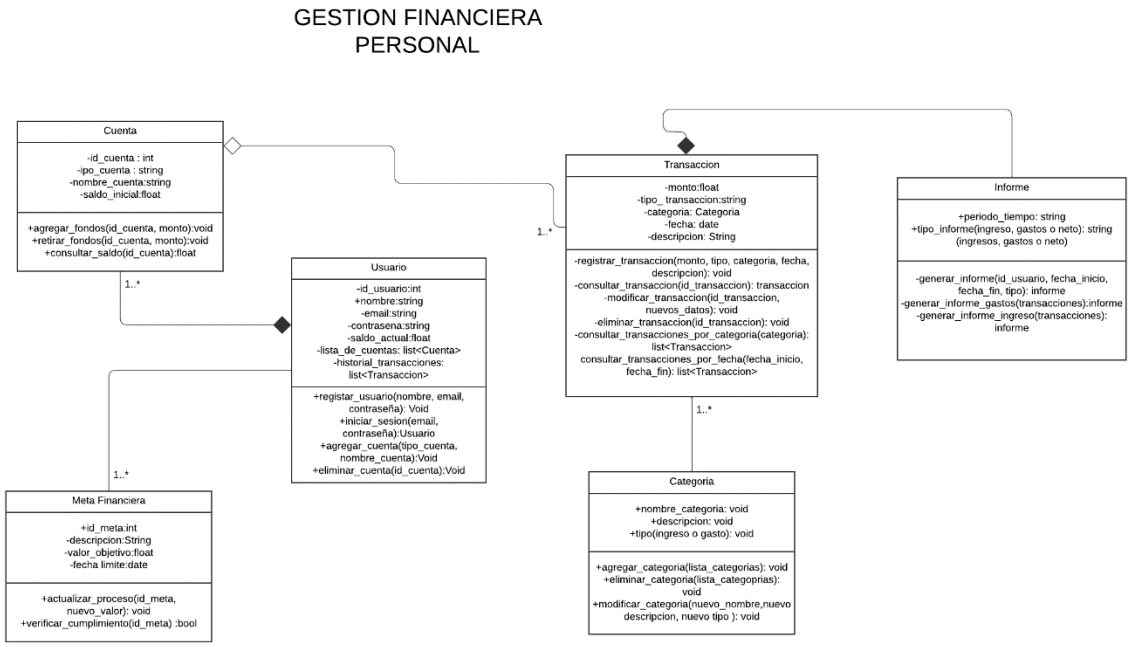
3. **Dependencia total**:
    - En este diseño, **el Informe depende completamente de la Transaccion**. Un Informe no puede existir por sí mismo; siempre necesita una transacción que lo cree y lo gestione. Si una transacción no se registra, tampoco se genera un informe.
    - Esta dependencia es lo que caracteriza a la **composición**: la relación es tan fuerte que la existencia de la clase contenida (Informe) depende de la clase contenedora (Transaccion).

¿Por qué es composición y no agregación?

La **composición** implica una relación más fuerte que la **agregación**, ya que en la composición:

- El ciclo de vida de la clase contenida está totalmente atado al ciclo de vida de la clase contenedora.
- La clase contenida (Informe) no puede existir sin la clase contenedora (Transaccion), lo cual es lo que está ocurriendo aquí.

Diagrama de clase UML



[Link LucidChart]

Documentación y Explicación UML

Clase Usuario

- **Atributos:**
  - id_usuario: int: Identificador único del usuario.
  - nombre: string: Nombre del usuario.
  - email: string: Correo electrónico del usuario.
  - contrasena: string: Contraseña del usuario.
  - saldo_actual: float: Saldo actual del usuario.
  - lista_de_cuentas: list<Cuenta>: Lista de cuentas del usuario.
  - historial_transacciones: list<Transaccion>: Historial de transacciones del usuario.
- **Métodos:**
  - +registrar_usuario(nombre, email, contraseña): void: Registra un nuevo usuario.

- +iniciar_sesion(email, contraseña): Usuario: Inicia sesión del usuario.
- +agregar_cuenta(tipo_cuenta, nombre_cuenta): void: Agrega una nueva cuenta.
- +eliminar_cuenta(id_cuenta): void: Elimina una cuenta existente.

Clase Transaccion

- **Atributos:**
  - monto: float: Monto de la transacción.
  - tipo_transaccion: string: Tipo de transacción (ingreso o gasto).
  - categoria: Categoria: Categoría de la transacción.
  - fecha: date: Fecha de la transacción.
  - descripcion: string: Descripción de la transacción.
- **Métodos:**
  - +registrar_transaccion(monto, tipo, categoria, fecha, descripcion): void: Registra una nueva transacción.
  - +consultar_transaccion(id_transaccion): Transaccion: Consulta una transacción específica.
  - +modificar_transaccion(id_transaccion, nuevos_datos): void: Modifica una transacción existente.
  - +eliminar_transaccion(id_transaccion): void: Elimina una transacción.
  - +consultar_transacciones_por_categoria(categoria): list<Transaccion>: Consulta transacciones por categoría.
  - +consultar_transacciones_por_fecha(fecha_inicio, fecha_fin): list<Transaccion>: Consulta transacciones por rango de fechas.

Clase Cuenta

- **Atributos:**
  - id_cuenta: int: Identificador único de la cuenta.
  - tipo_cuenta: string: Tipo de cuenta.
  - nombre_cuenta: string: Nombre de la cuenta.
  - saldo_inicial: float: Saldo inicial de la cuenta.
- **Métodos:**
  - +agregar_fondos(id_cuenta, monto): void: Agrega fondos a la cuenta.
  - +retirar_fondos(id_cuenta, monto): void: Retira fondos de la cuenta.
  - +consultar_saldo(id_cuenta): float: Consulta el saldo de la cuenta.

Clase Meta Financiera

- **Atributos:**
  - id_meta: int: Identificador único de la meta.
  - descripcion: string: Descripción de la meta.
  - valor_objetivo: float: Valor objetivo de la meta.
  - fecha_limite: date: Fecha límite para alcanzar la meta.
- **Métodos:**
  - +actualizar_proceso(id_meta, nuevo_valor): void: Actualiza el progreso de la meta.
  - +verificar_cumplimiento(id_meta): bool: Verifica si la meta ha sido cumplida.

Clase Categoria

- **Atributos:**
  - nombre_categoria: string: Nombre de la categoría.
  - descripcion: string: Descripción de la categoría.
- **Métodos:**
  - +tipo(ingreso o gasto): void: Define el tipo de la categoría.
  - +agregar_categoria(lista_categorias): void: Agrega una nueva categoría.
  - +eliminar_categoria(lista_categorias): void: Elimina una categoría existente.
  - +modificar_categoria(nuevo_nombre, nueva_descripcion, nuevo_tipo): void: Modifica una categoría.

Clase Informe

- **Atributos:**
  - periodo_tiempo: string: Periodo de tiempo del informe.
  - tipo_informe(ingreso, gastos o neto): string: Tipo de informe.
- **Métodos:**
  - +generar_informe(id_usuario, fecha_inicio, fecha_fin, tipo): informe: Genera un informe para un usuario en un rango de fechas específico.
  - +generar_informe_gastos(transacciones): informe: Genera un informe de gastos.
  - +generar_informe_ingreso(transacciones): informe: Genera un informe de ingresos.

Relaciones UML

- Un Usuario tiene una lista de Cuenta y un historial de Transaccion.
- Una Transaccion está asociada a una Categoria.
- Una Cuenta puede tener múltiples Transaccion.
- Una Meta Financiera está asociada a un Usuario.
- Un Informe puede generarse basado en las transacciones de un Usuario.

Documentación del Código

Clase Cuenta

Constructor __init__

- def __init__(self, id_cuenta, tipo_cuenta, nombre_cuenta, saldo_inicial=0):
  - Este es el método constructor que se ejecuta cuando se crea una nueva instancia de la clase Cuenta.
  - **Parámetros:**
    - id_cuenta: Identificador único para la cuenta.
    - tipo_cuenta: Tipo de la cuenta (por ejemplo, ahorro, corriente).
    - nombre_cuenta: Nombre de la cuenta.
    - saldo_inicial: Saldo inicial de la cuenta (opcional, por defecto es 0).
  - **Atributos:**
    - self.__id_cuenta: Almacena el ID de la cuenta.
    - self.__tipo_cuenta: Almacena el tipo de la cuenta.
    - self.__nombre_cuenta: Almacena el nombre de la cuenta.

        - self.__saldo: Almacena el saldo de la cuenta.
        - self.__transacciones: Almacena una lista de transacciones asociadas a la cuenta.

Getters y Setters

Estos métodos permiten acceder y modificar los atributos privados de la clase.

- **Métodos Getters:**
    - def get_id_cuenta(self): Retorna el ID de la cuenta.
    - def get_tipo_cuenta(self): Retorna el tipo de la cuenta.
    - def get_nombre_cuenta(self): Retorna el nombre de la cuenta.
    - def get_saldo(self): Retorna el saldo de la cuenta.
    - def get_transacciones(self): Retorna la lista de transacciones.
- **Métodos Setters:**
    - def set_tipo_cuenta(self, nuevo_tipo): Modifica el tipo de la cuenta.
    - def set_nombre_cuenta(self, nuevo_nombre): Modifica el nombre de la cuenta.
    - def set_saldo(self, nuevo_saldo): Modifica el saldo de la cuenta.

Métodos de Operación

- **agregar_fondos:**
    - def agregar_fondos(self, monto):
        - Este método añade fondos a la cuenta y actualiza el saldo.
        - **Parámetros:**
            - monto: Cantidad a agregar al saldo.
        - **Acciones:**
            - Incrementa el saldo con la cantidad especificada.
            - Imprime un mensaje con el monto agregado y el saldo actual.
- **retirar_fondos:**
    - def retirar_fondos(self, monto):
        - Este método retira fondos de la cuenta, si hay saldo suficiente.
        - **Parámetros:**
            - monto: Cantidad a retirar del saldo.
        - **Acciones:**
            - Si el saldo es suficiente, decrementa el saldo con la cantidad especificada.
            - Imprime un mensaje con el monto retirado y el saldo actual.
            - Si el saldo es insuficiente, imprime un mensaje de error.
- **consultar_saldo:**
    - def consultar_saldo(self):
        - Este método imprime y retorna el saldo actual de la cuenta.
        - **Acciones:**
            - Imprime un mensaje con el saldo actual de la cuenta.
            - Retorna el saldo actual.
- **agregar_transaccion:**
    - def agregar_transaccion(self, transaccion):
        - Este método añade una transacción a la cuenta.
        - **Parámetros:**

        - transaccion: Objeto de tipo Transaccion a agregar.
    - **Acciones:**
        - Añade la transacción a la lista de transacciones.
        - Si la transacción es de tipo "Ingreso", incrementa el saldo con el monto de la transacción.
        - Si la transacción es de tipo "Gasto", decrementa el saldo con el monto de la transacción.
        - Imprime un mensaje indicando que la transacción fue añadida.
- **mostrar_transacciones:**
    - def mostrar_transacciones(self):
        - Este método imprime todas las transacciones de la cuenta.
        - **Acciones:**
            - Si no hay transacciones, imprime un mensaje indicando que no hay transacciones.
            - Si hay transacciones, imprime cada transacción con su descripción, monto y tipo.

Clase Categoria

Constructor __init__

- def __init__(self, nombre_categoria, descripcion, tipo):
    - Este es el método constructor que se ejecuta cuando se crea una nueva instancia de la clase Categoria.
    - **Parámetros:**
        - nombre_categoria: Nombre de la categoría.
        - descripcion: Descripción de la categoría.
        - tipo: Tipo de la categoría (por ejemplo, ingreso o gasto).
    - **Atributos:**
        - self.nombre_categoria: Almacena el nombre de la categoría.
        - self.descripcion: Almacena la descripción de la categoría.
        - self.tipo: Almacena el tipo de la categoría.

Métodos

- **agregar_categoria:**
    - def agregar_categoria(self, lista_categorias):
        - Este método añade la categoría actual a una lista de categorías.
        - **Parámetros:**
            - lista_categorias: Lista de categorías a la que se añadirá la nueva categoría.
        - **Acciones:**
            - Añade la instancia actual (self) a la lista de categorías proporcionada.
            - Imprime un mensaje confirmando que la categoría fue agregada exitosamente.
- **eliminar_categoria:**
    - def eliminar_categoria(self, lista_categorias):
        - Este método elimina la categoría actual de una lista de categorías.
        - **Parámetros:**

            - lista_categorias: Lista de categorías de la cual se eliminará la categoría.
        - **Acciones:**
            - Recorre la lista de categorías y busca una categoría cuyo nombre coincida con el de la instancia actual (self).
            - Si encuentra la categoría, la elimina de la lista y imprime un mensaje de confirmación.
            - Si no encuentra la categoría, imprime un mensaje indicando que no se encontró la categoría.
- **modificar_categoria:**
    - def modificar_categoria(self, nuevo_nombre, nueva_descripcion, nuevo_tipo):
        - Este método modifica los atributos de la categoría actual.
        - **Parámetros:**
            - nuevo_nombre: Nuevo nombre para la categoría.
            - nueva_descripcion: Nueva descripción para la categoría.
            - nuevo_tipo: Nuevo tipo para la categoría.
        - **Acciones:**
            - Actualiza los atributos nombre_categoria, descripcion y tipo de la instancia actual con los nuevos valores proporcionados.
            - Imprime un mensaje confirmando que la categoría fue modificada exitosamente.

Clase **MetaFinanciera**

Importación

- from datetime import date: Importa la clase date del módulo datetime, que se utiliza para trabajar con fechas.

Constructor **__init__**

- def __init__(self, id_meta, valor_objetivo, fecha_limite, descripcion):
    - Este es el método constructor que se ejecuta cuando se crea una nueva instancia de la clase MetaFinanciera.
    - **Parámetros:**
        - id_meta: Identificador único de la meta.
        - valor_objetivo: Valor objetivo que se desea alcanzar con la meta.
        - fecha_limite: Fecha límite para alcanzar la meta.
        - descripcion: Descripción de la meta.
    - **Atributos:**
        - self.id_meta: Almacena el identificador único de la meta.
        - self._descripcion: Almacena la descripción de la meta.
        - self._valor_objetivo: Almacena el valor objetivo de la meta.
        - self._fecha_limite: Almacena la fecha límite para alcanzar la meta.
        - self._valor_actual: Almacena el progreso actual de la meta (inicialmente es 0).

Métodos

- **_actualizar_progreso:**

    - def _actualizar_progreso(self, cantidad):
        - Este método privado actualiza el progreso actual de la meta.
        - **Parámetros:**
            - cantidad: La cantidad a añadir al progreso actual.
        - **Acciones:**
            - Incrementa el atributo _valor_actual con la cantidad especificada.
            - Imprime un mensaje con el progreso actualizado.
- **verificar_cumplimiento:**
    - def verificar_cumplimiento(self):
        - Este método verifica si la meta ha sido cumplida o si ha pasado la fecha límite.
        - **Acciones:**
            - Compara _valor_actual con _valor_objetivo y la fecha actual con _fecha_limite.
            - Imprime un mensaje indicando si la meta ha sido cumplida, si la fecha límite ha pasado o si aún no se ha cumplido.
- **calcular_progreso:**
    - def calcular_progreso(self, gestion_transacciones, fecha_inicio, fecha_fin):
        - Este método calcula el progreso de la meta en función de las transacciones realizadas en un rango de fechas.
        - **Parámetros:**
            - gestion_transacciones: Objeto que gestiona las transacciones.
            - fecha_inicio: Fecha de inicio del rango para consultar las transacciones.
            - fecha_fin: Fecha de fin del rango para consultar las transacciones.
        - **Acciones:**
            - Consulta las transacciones en el rango de fechas especificado utilizando gestion_transacciones.
            - Itera sobre las transacciones y, si la categoría de la transacción es "ingreso", actualiza el progreso de la meta con el monto de la transacción.
- **registrar_meta:**
    - def registrar_meta(self, lista_metas):
        - Este método registra la meta en una lista de metas.
        - **Parámetros:**
            - lista_metas: Lista en la que se añadirá la meta.
        - **Acciones:**
            - Añade la instancia actual (self) a la lista de metas proporcionada.
            - Imprime un mensaje confirmando que la meta fue registrada con éxito.

Importaciones

- from controllers.transaction_controller import TransaccionIngreso, TransaccionGasto:
    - Importa las clases TransaccionIngreso y TransaccionGasto desde el módulo transaction_controller.

Clase Informe

Constructor __init__

- def __init__(self, tipo_informe):
  - Este es el método constructor que se ejecuta cuando se crea una nueva instancia de la clase Informe.
  - **Parámetros:**
    - tipo_informe: Tipo de informe a generar. Debe ser uno de los siguientes valores: "ingreso", "gastos", "neto".
  - **Atributos:**
    - self.tipo_informe: Almacena el tipo de informe.
  - **Acciones:**
    - Verifica que tipo_informe sea uno de los valores permitidos. Si no es así, lanza un ValueError.

Métodos

- **generar_informe:**
  - def generar_informe(self, transacciones, fecha_inicio, fecha_fin):
    - Este método genera un informe basado en las transacciones y el tipo de informe especificado.
    - **Parámetros:**
      - transacciones: Lista de transacciones a considerar.
      - fecha_inicio: Fecha de inicio del rango para filtrar las transacciones.
      - fecha_fin: Fecha de fin del rango para filtrar las transacciones.
    - **Acciones:**
      - Filtra las transacciones para incluir solo aquellas que están dentro del rango de fechas especificado.
      - Llama al método correspondiente (_generar_informe_ingresos, _generar_informe_gastos, _generar_informe_neto) basado en self.tipo_informe.
- **_generar_informe_ingresos:**
  - def _generar_informe_ingresos(self, transacciones):
    - Este método privado genera un informe de ingresos.
    - **Parámetros:**
      - transacciones: Lista de transacciones filtradas.
    - **Acciones:**
      - Filtra las transacciones para incluir solo aquellas de tipo TransaccionIngreso.
      - Calcula el total de ingresos sumando los montos de las transacciones filtradas.
      - Retorna una cadena con el total de ingresos.
- **_generar_informe_gastos:**
  - def _generar_informe_gastos(self, transacciones):
    - Este método privado genera un informe de gastos.
    - **Parámetros:**
      - transacciones: Lista de transacciones filtradas.
    - **Acciones:**

            - Filtra las transacciones para incluir solo aquellas de tipo TransaccionGasto.
            - Calcula el total de gastos sumando los montos de las transacciones filtradas.
            - Retorna una cadena con el total de gastos.
- **_generar_informe_neto:**
    - def _generar_informe_neto(self, transacciones):
        - Este método privado genera un informe neto (ingresos menos gastos).
        - **Parámetros:**
            - transacciones: Lista de transacciones filtradas.
        - **Acciones:**
            - Calcula el total de ingresos sumando los montos de las transacciones de tipo TransaccionIngreso.
            - Calcula el total de gastos sumando los montos de las transacciones de tipo TransaccionGasto.
            - Calcula el neto restando el total de gastos del total de ingresos.
            - Retorna una cadena con el total neto.

Clase Transaccion

Constructor __init__

- def __init__(self, monto, categoria, fecha, descripcion):
    - Este es el método constructor que se ejecuta cuando se crea una nueva instancia de la clase Transaccion.
    - **Parámetros:**
        - monto: Monto de la transacción.
        - categoria: Categoría de la transacción.
        - fecha: Fecha de la transacción.
        - descripcion: Descripción de la transacción.
    - **Atributos:**
        - self.__monto: Almacena el monto de la transacción.
        - self.__categoria: Almacena la categoría de la transacción.
        - self.__fecha: Almacena la fecha de la transacción.
        - self.__descripcion: Almacena la descripción de la transacción.

Métodos

- **get_monto:**
    - def get_monto(self):
        - Este método retorna el monto de la transacción.
        - **Retorno:** El monto de la transacción (self.__monto).
- **get_categoria:**
    - def get_categoria(self):
        - Este método retorna la categoría de la transacción.
        - **Retorno:** La categoría de la transacción (self.__categoria).
- **get_fecha:**
    - def get_fecha(self):
        - Este método retorna la fecha de la transacción.

        - **Retorno:** La fecha de la transacción (self.__fecha).
- **get_descripcion:**
    - def get_descripcion(self):
        - Este método retorna la descripción de la transacción.
        - **Retorno:** La descripción de la transacción (self.__descripcion).
- **registrar_transaccion:**
    - def registrar_transaccion(self, lista_transacciones):
        - Este método registra la transacción en una lista de transacciones.
        - **Parámetros:**
            - lista_transacciones: Lista de transacciones donde se registrará la transacción.
        - **Acciones:**
            - Añade la instancia actual (self) a la lista de transacciones.
            - Imprime un mensaje confirmando que la transacción fue registrada.
- **consultar_transaccion:**
    - def consultar_transaccion(self, id_transaccion, lista_transacciones):
        - Este método busca y devuelve una transacción por su ID en la lista de transacciones.
        - **Parámetros:**
            - id_transaccion: ID de la transacción a consultar.
            - lista_transacciones: Lista de transacciones donde se buscará la transacción.
        - **Acciones:**
            - Verifica si el ID está dentro del rango válido de la lista de transacciones.
            - Si el ID es válido, retorna la transacción correspondiente.
            - Si el ID no es válido, imprime un mensaje de error y retorna None.
- **modificar_transaccion:**
    - def modificar_transaccion(self, id_transaccion, nuevos_datos, lista_transacciones):
        - Este método modifica una transacción con nuevos datos.
        - **Parámetros:**
            - id_transaccion: ID de la transacción a modificar.
            - nuevos_datos: Diccionario con los nuevos datos para modificar la transacción.
            - lista_transacciones: Lista de transacciones donde se encuentra la transacción a modificar.
        - **Acciones:**
            - Verifica si el ID está dentro del rango válido de la lista de transacciones.
            - Si el ID es válido, actualiza los atributos de la transacción con los nuevos datos proporcionados.
            - Imprime un mensaje confirmando que la transacción fue modificada.
            - Si el ID no es válido, imprime un mensaje de error.

Subclases de **Transaccion**

Clase **TransaccionIngreso**

- **Constructor __init__:**
  - def __init__(self, monto, categoria, fecha, descripcion):
    - Este es el método constructor que se ejecuta cuando se crea una nueva instancia de la clase TransaccionIngreso.
    - **Parámetros:**
      - monto: Monto de la transacción.
      - categoria: Categoría de la transacción.
      - fecha: Fecha de la transacción.
      - descripcion: Descripción de la transacción.
    - **Acciones:**
      - Llama al constructor de la clase base Transaccion utilizando super().
      - Define el atributo tipo_transaccion como "Ingreso".

Clase TransaccionGasto

- **Constructor __init__:**
  - def __init__(self, monto, categoria, fecha, descripcion):
    - Este es el método constructor que se ejecuta cuando se crea una nueva instancia de la clase TransaccionGasto.
    - **Parámetros:**
      - monto: Monto de la transacción.
      - categoria: Categoría de la transacción.
      - fecha: Fecha de la transacción.
      - descripcion: Descripción de la transacción.
    - **Acciones:**
      - Llama al constructor de la clase base Transaccion utilizando super().
      - Define el atributo tipo_transaccion como "Gasto".

Clase Usuario

Constructor __init__

- def init(self, nombre, email, password, saldo_actual=0, historial_transacciones=None):
  - Este es el método constructor que se ejecuta cuando se crea una nueva instancia de la clase Usuario.
  - **Parámetros:**
    - nombre: Nombre del usuario.
    - email: Correo electrónico del usuario.
    - password: Contraseña del usuario (almacenada de forma privada).
    - saldo_actual: Saldo actual del usuario, por defecto 0.
    - historial_transacciones: Historial de transacciones del usuario, por defecto None (vacío).
  - **Atributos:**
    - self.nombre: Almacena el nombre del usuario.
    - self.email: Almacena el correo electrónico del usuario.
    - self.__password: Almacena la contraseña del usuario (privada).
    - self.__saldo_actual: Almacena el saldo actual del usuario (privada).

        - self.__historial_transacciones: Almacena el historial de transacciones (privada).
        - self.__cuentas: Almacena la lista de cuentas del usuario (privada).
        - self.lista_categorias: Almacena la lista de categorías asociadas al usuario.
        - self.lista_metas: Almacena la lista de metas asociadas al usuario (no utilizada en este fragmento).
        - self.lista_transacciones: Almacena la lista de transacciones (no utilizada en este fragmento).

Métodos

- get_password
    - def get_password(self):
        - Este método retorna la contraseña del usuario.
        - Retorno: La contraseña del usuario (self.__password).
- set_password
    - def set_password(self, new_password):
        - Este método establece una nueva contraseña para el usuario.
        - Parámetros:
            - new_password: Nueva contraseña del usuario.
- get_saldo_actual
    - def get_saldo_actual(self):
        - Este método retorna el saldo actual del usuario.
        - Retorno: El saldo actual del usuario (self.__saldo_actual).
- set_saldo_actual
    - def set_saldo_actual(self, new_saldo):
        - Este método establece un nuevo saldo para el usuario.
        - **Parámetros:**
            - new_saldo: Nuevo saldo del usuario.
- get_historial_transacciones
    - def get_historial_transacciones(self):
        - Este método retorna el historial de transacciones del usuario.
        - **Retorno:** El historial de transacciones del usuario (self.__historial_transacciones).
- set_historial_transacciones
    - def set_historial_transacciones(self, nuevo_historial):
        - Este método establece un nuevo historial de transacciones para el usuario.
        - **Parámetros:**
            - nuevo_historial: Nuevo historial de transacciones del usuario.
- get_cuentas
    - def get_cuentas(self):
        - Este método retorna la lista de cuentas asociadas al usuario.
        - **Retorno:** Lista de cuentas del usuario (self.__cuentas).
- agregar_cuenta
    - def agregar_cuenta(self, id_cuenta, tipo_cuenta, nombre_cuenta, saldo_inicial=0):
        - Este método agrega una nueva cuenta al usuario.
        - **Parámetros:**
            - **id_cuenta:** ID de la nueva cuenta.
            - **tipo_cuenta:** Tipo de la nueva cuenta (por ejemplo, ahorro, corriente).

            - **nombre_cuenta:** Nombre de la nueva cuenta.
            - **saldo_inicial:** Saldo inicial de la nueva cuenta, por defecto 0.
        - **Acciones:**
            - Crea una instancia de la clase Cuenta y la añade a la lista de cuentas del usuario.
            - Imprime un mensaje confirmando que la cuenta fue agregada exitosamente.
- eliminar_cuenta
    - def eliminar_cuenta(self, nombre_cuenta):
        - Este método elimina una cuenta del usuario por su nombre.
        - **Parámetros:**
            - **nombre_cuenta:** Nombre de la cuenta a eliminar.
        - **Acciones:**
            - Busca la cuenta en la lista de cuentas y la elimina si se encuentra.
            - Imprime un mensaje confirmando que la cuenta fue eliminada o que no se encontró.
- mostrar_cuentas
    - def mostrar_cuentas(self):
        - Este método muestra todas las cuentas asociadas al usuario.
        - **Acciones:**
            - Verifica si el usuario tiene cuentas y las imprime; si no tiene, imprime un mensaje indicando que no hay cuentas.
- agregar_categoria
    - def agregar_categoria(self, nombre_categoria, descripcion, tipo):
        - Este método agrega una nueva categoría a la lista de categorías del usuario.
        - **Parámetros:**
            - **nombre_categoria:** Nombre de la nueva categoría.
            - **descripcion:** Descripción de la nueva categoría.
            - **tipo:** Tipo de la nueva categoría (por ejemplo, ingresos, gastos).
        - **Acciones:**
            - Crea una instancia de la clase Categoria y la añade a la lista de categorías del usuario.
- eliminar_categoria
    - def eliminar_categoria(self, nombre_categoria):
        - Este método elimina una categoría de la lista de categorías del usuario.
        - **Parámetros:**
            - **nombre_categoria:** Nombre de la categoría a eliminar.
        - **Acciones:**
            - Crea una instancia de la clase Categoria y llama a su método para eliminar la categoría de la lista del usuario.
- mostrar_categorias
    - def mostrar_categorias(self):
        - Este método muestra todas las categorías registradas del usuario.
        - **Acciones:**
            - Verifica si hay categorías y las imprime; si no hay, imprime un mensaje indicando que no hay categorías.
- modificar_categoria

- def modificar_categoria(self, nombre_actual, nuevo_nombre, nueva_descripcion, nuevo_tipo):
  - Este método modifica una categoría existente por su nombre.
  - **Parámetros:**
    - **nombre_actual:** Nombre actual de la categoría a modificar.
    - **nuevo_nombre:** Nuevo nombre de la categoría.
    - **nueva_descripcion:** Nueva descripción de la categoría.
    - **nuevo_tipo:** Nuevo tipo de la categoría.
  - **Acciones:**
    - Busca la categoría en la lista de categorías y la modifica si se encuentra.
    - Imprime un mensaje indicando si la categoría fue modificada o no se encontró.

Función validar_email

Definición

- **def validar_email(email):**
  - Esta función se encarga de validar si un correo electrónico (email) tiene un formato correcto.

Docstring

- **"""Valida si el formato del email es correcto"""**
  - Proporciona una breve descripción de la función, indicando que su propósito es validar el formato del correo electrónico.

Condicional

- **if "@" in email and "." in email:**
  - Se utiliza una condición que verifica si el símbolo "@" y el punto "." están presentes en el correo electrónico.
  - **Lógica:** Para que un correo electrónico sea considerado válido, generalmente debe contener ambos caracteres; el "@" separa el nombre de usuario del dominio, y el "." generalmente precede a una extensión (por ejemplo, .com, .org).

Retorno

- **return True**
  - Si ambos caracteres están presentes, la función devuelve True, indicando que el formato del correo electrónico es válido.

Caso contrario

- **return False**

    - Si la condición no se cumple (es decir, falta "@" o "."), la función devuelve False, indicando que el formato del correo electrónico no es válido.

Librerías centrales

Kivy

- **Descripción:**
    - Una plataforma para el desarrollo de aplicaciones multiplataforma (móvil, escritorio, web) con una interfaz de usuario natural (NUI).
    - Es ideal para crear aplicaciones con elementos gráficos interactivos como juegos, simulaciones y aplicaciones de realidad aumentada.

Pillow (PIL Fork)

- **Descripción:**
    - La biblioteca de imágenes de Python, utilizada para abrir, manipular y guardar imágenes en diversos formatos.
    - Es fundamental para tareas como el procesamiento de imágenes, creación de miniaturas y conversión de formatos.

certifi

- **Descripción:**
    - Proporciona un conjunto de certificados SSL/TLS de confianza.
    - Esencial para realizar solicitudes seguras a través de HTTPS.

Dependencias de Kivy

kivy_deps

- **Descripción:**
    - Conjunto de dependencias necesarias para que Kivy funcione correctamente en diferentes plataformas.

angle, glew, sdl2

- **Descripción:**
    - Estas son bibliotecas gráficas y de sistema que Kivy utiliza para renderizar gráficos y gestionar la interacción con el hardware.

Otras librerías

adodbapi

- **Descripción:**

    - Probablemente utilizada para la conexión a bases de datos, aunque se necesita más contexto para determinar el tipo de base de datos específico.

charset_normalizer

- **Descripción:**
    - Ayuda a detectar y normalizar codificaciones de caracteres en texto.

docutils

- **Descripción:**
    - Utilizado para procesar y estructurar documentos, especialmente en formatos como reStructuredText.

idna

- **Descripción:**
    - Maneja la conversión entre nombres de dominio internacionales y las etiquetas ASCII utilizadas en el sistema de nombres de dominio (DNS).

isapi

- **Descripción:**
    - Probablemente relacionada con el desarrollo de extensiones para servidores web IIS de Microsoft.

pip

- **Descripción:**
    - El gestor de paquetes de Python, utilizado para instalar y administrar otras librerías.

Librerías importadas

from controllers.user_controller import Usuario

- **Descripción:**
    - Importa la clase Usuario desde el módulo user_controller, que se encarga de gestionar las operaciones relacionadas con los usuarios.

from controllers.transaction_controller import Transaccion

- **Descripción:**
    - Importa la clase Transaccion desde el módulo transaction_controller, que maneja las transacciones financieras.

from controllers.account_controller import Cuenta

- **Descripción:**
  - Importa la clase Cuenta desde el módulo account_controller, que gestiona las cuentas de los usuarios.

from controllers.transaction_controller import TransaccionIngreso, TransaccionGasto

- **Descripción:**
  - Importa las clases TransaccionIngreso y TransaccionGasto desde el módulo transaction_controller, que representan transacciones de ingreso y gasto, respectivamente.

from controllers.category_controller import Categoria

- **Descripción:**
  - Importa la clase Categoria desde el módulo category_controller, que se encarga de gestionar las categorías de las transacciones.

from controllers.inform_controller import Informe

- **Descripción:**
  - Importa la clase Informe desde el módulo inform_controller, que se encarga de generar informes financieros.

from controllers.goals_finance_controller import MetaFinanciera

- **Descripción:**
  - Importa la clase MetaFinanciera desde el módulo goals_finance_controller, que gestiona las metas financieras de los usuarios.

from datetime import date

- **Descripción:**
  - Importa la clase date del módulo datetime, que se utiliza para manejar fechas.

Variables Globales

lista_categorias

- **Descripción:**
  - Lista vacía que almacenará las categorías de las transacciones.

lista_transacciones

- **Descripción:**
  - Lista vacía que almacenará las transacciones financieras registradas.

Funciones

mostrar_menu

```
def mostrar_menu():
    print("1. Registrarse")
    print("2. Iniciar sesión")
    print("3. Salir")
```

- **Descripción:**
    - Muestra el menú principal de la aplicación.

mostrar_crear_cuenta

```
def mostrar_crear_cuenta():
    print("\nMenú de Cuentas:")
    print("1. Agregar cuenta")
    print("2. Eliminar cuenta")
    print("3. Mostrar cuentas")
    print("4. Salir")
```

- **Descripción:**
    - Muestra el menú de gestión de cuentas.

mostrar_menu_categorias

```
def mostrar_menu_categorias():
    print("\n menú de categorías")
    print("1. Agregar categoría")
    print("2. Eliminar categoría")
    print("3. Modificar categoría")
    print("4. Mostrar categorías")
    print("5. Salir")
```

- **Descripción:**
    - Muestra el menú de gestión de categorías.

registrar_usuario

```
def registrar_usuario():
    nombre = input("Ingrese su nombre: ")
    email = input("Ingrese su email: ")
    password = input("Ingrese su contraseña: ")
    nuevo_usuario = Usuario(nombre, email, password)
    Usuario.usuarios_registrados.append(nuevo_usuario)
    print(f"Usuario {nombre} registrado exitosamente.")
```

- **Descripción:**
    - Registra un nuevo usuario en el sistema.
- **Acciones:**
    - Solicita al usuario su nombre, email y contraseña.

- Crea una nueva instancia de Usuario y la agrega a la lista de usuarios registrados.

iniciar_sesion

```
def iniciar_sesion():
    email = input("Ingrese su email: ")
    password = input("Ingrese su contraseña: ")

    for usuario in Usuario.usuarios_registrados:
        if usuario.email == email and usuario.get_password() == password:
            print(f"Bienvenido, {usuario.nombre}!")
            return usuario

    print("Credenciales incorrectas. Intente nuevamente.")
    return None
```

- **Descripción:**
  - Inicia sesión para un usuario registrado.
- **Acciones:**
  - Verifica las credenciales ingresadas y, si son correctas, devuelve el objeto Usuario.

agregar_cuenta

```
def agregar_cuenta(usuario):
    id_cuenta = input("Ingrese el ID de la cuenta: ")
    tipo_cuenta = input("Ingrese el tipo de cuenta: ")
    nombre_cuenta = input("Ingrese el nombre de la cuenta: ")
    saldo_inicial = float(input("Ingrese el saldo inicial: "))

    usuario.agregar_cuenta(id_cuenta, tipo_cuenta, nombre_cuenta, saldo_inicial)
```

- **Descripción:**
  - Agrega una nueva cuenta a un usuario.
- **Parámetros:**
  - usuario: El objeto Usuario al que se le agregará la cuenta.

eliminar_cuenta

```
def eliminar_cuenta(usuario):
    nombre_cuenta = input("Ingrese el nombre de la cuenta a eliminar: ")
    usuario.eliminar_cuenta(nombre_cuenta)
```

- **Descripción:**
  - Elimina una cuenta del usuario.
- **Parámetros:**
  - usuario: El objeto Usuario del que se eliminará la cuenta.

mostrar_cuentas

```
def mostrar_cuentas(usuario):
    usuario.mostrar_cuentas()
```

- **Descripción:**
  - Muestra todas las cuentas del usuario.
- **Parámetros:**
  - usuario: El objeto Usuario cuyas cuentas se mostrarán.

menu_cuenta

```
def menu_cuenta(usuario):
    while True:
        mostrar_crear_cuenta()
        opcion_menu = input("Seleccione una opción: ")

        if opcion_menu == '1':
            agregar_cuenta(usuario)
        elif opcion_menu == '2':
            eliminar_cuenta(usuario)
        elif opcion_menu == '3':
            mostrar_cuentas(usuario)
        elif opcion_menu == '4':
            break
        else:
            print("Opción inválida. Intente nuevamente.")
```

- **Descripción:**
  - Muestra el menú de gestión de cuentas y ejecuta las acciones correspondientes según la opción seleccionada.
- **Parámetros:**
  - usuario: El objeto Usuario para gestionar las cuentas.

ingresar_gastos

```
def ingresar_gastos(usuario, lista_transacciones):
    descripcion = input("Ingrese la descripción del gasto: ")
    monto = float(input("Ingrese el monto del gasto: "))

    categoria = mostrar_categorias(usuario)
    if categoria is None:
        print("Categoría no válida. Gasto no registrado.")
        return

    fecha = input("Ingrese la fecha del gasto (YYYY-MM-DD): ")
    id_cuenta = input("Ingrese el ID de la cuenta: ")

    cuenta_encontrada = None
    for cuenta in usuario.get_cuentas():
        if cuenta.get_id_cuenta() == id_cuenta:
            cuenta_encontrada = cuenta
            break

    if cuenta_encontrada:
        transaccion = TransaccionGasto(monto, categoria.nombre_categoria, fecha, descripcion)
        cuenta_encontrada.agregar_transaccion(transaccion)
        usuario.lista_transacciones.append(transaccion)

        print("Gasto registrado exitosamente.")
    else:
        print("La cuenta no existe.")
```

- **Descripción:**
  - Permite al usuario registrar un gasto.
- **Parámetros:**
  - usuario: El objeto Usuario que registra el gasto.
  - lista_transacciones: La lista de transacciones donde se almacenará el gasto.

ingresar_ingresos

```
def ingresar_ingresos(usuario, lista_transacciones):
    descripcion = input("Ingrese la descripción del ingreso: ")
    monto = float(input("Ingrese el monto del ingreso: "))

    categoria = mostrar_categorias(usuario)
    if categoria is None:
        print("Categoría no válida. Ingreso no registrado.")
        return

    fecha = input("Ingrese la fecha del ingreso (YYYY-MM-DD): ")
    id_cuenta = input("Ingrese el ID de la cuenta: ")

    cuenta_encontrada = None
    for cuenta in usuario.get_cuentas():
        if cuenta.get_id_cuenta() == id_cuenta:
            cuenta_encontrada = cuenta
            break

    if cuenta_encontrada:
        transaccion = TransaccionIngreso(monto, categoria.nombre_categoria, fecha, descripcion)
        cuenta_encontrada.agregar_transaccion(transaccion)
        usuario.lista_transacciones.append(transaccion)

        print("Ingreso registrado exitosamente.")
    else:
        print("La cuenta no existe.")
```

- **Descripción:**
  - Permite al usuario registrar un ingreso.
- **Parámetros:**
  - usuario: El objeto Usuario que registra el ingreso.
  - lista_transacciones: La lista de transacciones donde se almacenará el ingreso.

mostrar_categorias

```
def mostrar_categorias(usuario):
    print("Categorías disponibles:")
    for idx, categoria in enumerate(usuario.lista_categorias):
        print(f"{idx + 1}. {categoria.nombre_categoria} - {categoria.descripcion} ({categoria.tipo})")

    seleccion = int(input("Seleccione el número de la categoría: ")) - 1
    if 0 <= seleccion < len(usuario.lista_categorias):
        return usuario.lista_categorias[seleccion]
    else:
        print("Selección inválida.")
        return None
```

- **Descripción:**
  - Muestra las categorías disponibles y permite al usuario seleccionar una.
- **Parámetros:**
  - usuario: El objeto Usuario cuyas categorías se mostrarán.

menu_categoria

```
def menu_categoria(usuario):
    while True:
        mostrar_menu_categorias()
        opcion = input("Seleccione una opción: ")

        if opcion == '1':
            nombre_categoria = input("Ingrese el nombre de la categoría: ")
            descripcion = input("Ingrese una descripción para la categoría: ")
            tipo = input("Ingrese el tipo de la categoría (Ingreso o Gasto): ")
            usuario.agregar_categoria(nombre_categoria, descripcion, tipo)
        elif opcion == '2':
            nombre_categoria = input("Ingrese el nombre de la categoría a eliminar: ")
            usuario.eliminar_categoria(nombre_categoria)
        elif opcion == '3':
            nombre_categoria = input("Ingrese el nombre de la categoría a modificar: ")
            nombre_categorianuevo = input("Ingrese el nombre de la categoría a modificar: ")
            descripcion = input("Ingrese nueva descripción para la categoría: ")
            tipo = input("Ingrese  tipo de la categoría a modificar (Ingreso o Gasto): ")
            usuario.modificar_categoria(nombre_categoria,nombre_categorianuevo, descripcion, tipo)
        elif opcion == '4':
            usuario.mostrar_categorias()
        elif opcion == '5':
            print("Saliendo del menú de categorías...")
            break
        else:
            print("Opción inválida. Intente nuevamente.")
```

- **Descripción:**
  - Muestra el menú de gestión de categorías y ejecuta las acciones correspondientes según la opción seleccionada.
- **Parámetros:**
  - usuario: El objeto Usuario para gestionar las categorías.

menu_informes

```
def menu_informes(lista_transacciones):
    print("\n=== GENERAR INFORMES ===")
    print("1. Informe de ingresos")
    print("2. Informe de gastos")
    print("3. Informe neto")
    print("4. Volver al menú principal")
    opcion = input("Seleccione una opción: ")

    if opcion in ['1', '2', '3']:
        fecha_inicio = input("Ingrese la fecha de inicio (YYYY-MM-DD): ")
        fecha_fin = input("Ingrese la fecha de fin (YYYY-MM-DD): ")

        tipo_informe = ""
        if opcion == '1':
            tipo_informe = "ingreso"
```

```
        elif opcion == '2':
            tipo_informe = "gastos"
        elif opcion == '3':
            tipo_informe = "neto"

        informe = Informe(tipo_informe)
        resultado = informe.generar_informe(lista_transacciones, fecha_inicio, fecha_fin)

        print(resultado)
    elif opcion == '4':
        return
    else:
        print("Opción no válida.")
```

- **Descripción:**
  - Muestra el menú para generar informes y ejecuta la acción correspondiente según la opción seleccionada.
- **Parámetros:**
  - lista_transacciones: La lista de transacciones para generar los informes.

menu_cuentaPRI

```
python
def menu_cuentaPRI(usuario, lista_metas, lista_transacciones):
    while True:
        print("\n=== MENÚ DE CUENTAS ===")
        print("1. Ingresar gastos")
        print("2. Ingresar ingresos")
        print("3. Cuenta")
        print("4. Categorías")
        print("5. Gestionar Metas")
        print("6. Generar Informe")
        print("7. Logout")
        print("========================")

        opcion_menu = input("Seleccione una opción: ")

        if opcion_menu == '1':
            ingresar_gastos(usuario, lista_transacciones)
        elif opcion_menu == '2':
            ingresar_ingresos(usuario, lista_transacciones)
        elif opcion_menu == '3':
            menu_cuenta(usuario)
        elif opcion_menu == '4':
            menu_categoria(usuario)
        elif opcion_menu == '5':
            menu_metas(lista_metas)
        elif opcion_menu == '6':
            menu_informes(usuario.lista_transacciones)
        elif opcion_menu == '7':
            print("Cerrando sesión...")
            break
        else:
            print("Opción invalida. Intente nuevamente.")
```

- **Descripción:**

    - Muestra el menú principal de cuentas y permite al usuario realizar acciones relacionadas con ingresos, gastos, cuentas, categorías, metas e informes.
- **Parámetros:**
    - `usuario`: El objeto `Usuario` que realiza las acciones.
    - `lista_metas`: La lista de metas financieras del usuario.
    - `lista_transacciones`: La lista de transacciones registradas.

menu_metas

```
python
def menu_metas(lista_metas):
    while True:
        print("\n=== GESTIÓN DE METAS ===")
        print("1. Registrar nueva meta")
        print("2. Verificar cumplimiento de metas")
        print("3. Volver al menú principal")
        print("========================")

        opcion = input("Seleccione una opción: ")

        if opcion == '1':
            id_meta = input("Ingrese el ID de la meta: ")
            descripcion = input("Ingrese la descripción de la meta: ")
            valor_objetivo = float(input("Ingrese el valor objetivo: "))

            try:
                fecha_limite = date.fromisoformat(input("Ingrese la fecha límite (YYYY-MM-DD): "))
            except ValueError:
                print("Fecha inválida. Intente nuevamente.")
                continue

            nueva_meta = MetaFinanciera(id_meta, valor_objetivo, fecha_limite, descripcion)
            nueva_meta.registrar_meta(lista_metas)
            print("Meta registrada exitosamente.")

        elif opcion == '2':
            if not lista_metas:
                print("No hay metas registradas.")
            else:
                for meta in lista_metas:
                    print(f"\nMeta: {meta._descripcion}")
                    meta.verificar_cumplimiento()

        elif opcion == '3':
            break
        else:
            print("Opción inválida. Intente nuevamente.")
```

- **Descripción:**
    - Muestra el menú de gestión de metas y permite al usuario registrar nuevas metas o verificar su cumplimiento.
- **Parámetros:**
    - `lista_metas`: La lista de metas financieras del usuario.

main

```python
def main():
    lista_metas = []
    lista_transacciones = []

    while True:
        mostrar_menu()
        opcion = input("Seleccione una opción: ")

        if opcion == '1':
            registrar_usuario()
        elif opcion == '2':
            usuario = iniciar_sesion()
            if usuario:
                menu_cuentaPRI(usuario, lista_metas, lista_transacciones)
        elif opcion == '3':
            print("Saliendo...")
            break
        else:
            print("Opción inválida. Intente nuevamente.")
```

- **Descripción:**
    - Función principal que ejecuta la aplicación y gestiona el flujo entre el menú principal y las distintas opciones disponibles.

name == "main"

```python
if __name__ == "__main__":
    main()
```

- **Descripción:**
    - Verifica si el script se está ejecutando como programa principal y llama a la función `main` para iniciar la aplicación.