

Instituto de formación técnica superior N° 18

Paradigmas de Programación - apuntes

Curso: 1° Año

Profesor: Bonini Juan Ignacio

Ciclo lectivo: 2023

Carga horaria: 5 hs cátedra/semana

Régimen: Cuatrimestral

Introducción a la programación orientada a objetos (POO)

El paradigma de programación orientada a objetos se centra en la creación y manipulación de objetos como bloques fundamentales para construir software. Con este enfoque, los programas se diseñan modelando entidades del mundo real o abstracto como objetos, que combinan datos (atributos) y comportamientos (métodos) que están estrechamente vinculados y forman un conjunto lógico y cohesivo de funcionalidades.

Cuando se diseñan clases y objetos en la programación orientada a objetos, se busca agrupar características y comportamientos relacionados en una misma entidad. Esto se hace para que los datos y las acciones que pertenecen juntos estén organizados de manera ordenada y se puedan manipular de manera conjunta.

La coherencia en el diseño de clases y objetos es importante para crear un código más legible, mantenible y comprensible. Cuando los elementos relacionados están agrupados de manera lógica, es más fácil entender cómo interactúan entre sí y qué responsabilidades tiene cada parte del código.

Clase

Una clase es una plantilla o un plano para crear objetos. Define las propiedades (atributos) y comportamientos (métodos) que los objetos creados a partir de esa clase tendrán. En otras palabras, una clase es una representación abstracta de un concepto, entidad o tipo de datos. Una clase y un objeto son conceptos fundamentales que permiten organizar y estructurar el código de manera más eficiente y modular. Las clases agrupan los atributos y métodos relacionados en una sola unidad, lo que fomenta la reutilización de código y la organización eficiente de la lógica del programa.

Objeto

Un objeto es una instancia concreta de una clase. Se crea utilizando la plantilla proporcionada por la clase y contiene valores específicos para los atributos definidos en la clase. Los objetos son las unidades básicas con las que trabajamos en la POO y pueden interactuar entre sí a través de métodos y propiedades.

Imaginemos que estamos construyendo una ciudad virtual en nuestra computadora. Una clase es como un plano o diseño para crear edificios similares. Tiene todas las instrucciones sobre cómo deben ser los edificios: cuántos pisos, qué colores, qué puertas y ventanas.

Ahora, un objeto es un edificio real hecho basándose en ese plano. Cada edificio puede ser único, pero sigue las reglas del plano de la clase. Por ejemplo, si la clase es "Edificio", los objetos podrían ser "Casa", "Oficina" o "Escuela", todos contruidos a partir de la misma plantilla pero con detalles diferentes.

Atributos

Los atributos son las características o propiedades que definen el estado de un objeto. Representan datos asociados a un objeto y pueden ser variables que almacenan información específica. Los atributos son esenciales para describir las características únicas de cada instancia de una clase.

Supongamos que tenemos una clase "Auto". Los atributos de esta clase podrían ser "marca", "color" y "rodado". Estos atributos definen las características de un auto en el contexto del programa.

Propiedades

Las propiedades son métodos especiales que se utilizan para controlar el acceso y la modificación de los atributos de una clase. Permiten definir un comportamiento personalizado al obtener o establecer el valor de un atributo, en lugar de acceder directamente a él. Las propiedades son útiles para implementar lógica adicional al interactuar con los datos de un objeto.

Siguiendo el ejemplo previo, si deseamos transformar el atributo "color" de la clase Auto en una propiedad, el proceso se inicia renombrando la variable a "_color". Este enfoque de nomenclatura con un guión bajo indica que la variable es privada y no debe accederse directamente.

Posteriormente, creamos la propiedad "color", que está diseñada para recuperar el color del auto. Sin embargo, esta propiedad no permitirá que el valor del color sea modificado directamente. En cambio, se proporciona una interfaz controlada para obtener información sobre el color del automóvil mientras se mantiene la integridad de los datos internos.

Métodos

Un método es una función definida dentro de una clase que opera en instancias de esa clase. Los métodos son acciones o comportamientos asociados a un objeto y se utilizan para realizar diversas operaciones y manipulaciones en los datos contenidos en ese objeto.

Un método en Python se declara dentro de una clase y generalmente toma self como su primer parámetro, que hace referencia a la instancia del objeto en sí mismo. Los métodos pueden acceder a los atributos de la instancia y realizar tareas específicas relacionadas con la clase.

Si creáramos el método "cambiar_color" en la clase "Auto" sería una función definida dentro de la clase que permitiría modificar el color del automóvil. Este método tomaría un nuevo color como parámetro y, después de validar si el color es válido, cambiaría el color actual del auto al nuevo color proporcionado.

El método "cambiar_color" en la clase "Auto" sería responsable de cambiar el color del automóvil a otro color válido en función de los criterios de validación establecidos.

Métodos de clase

Un método de clase en Python es una función definida dentro de una clase que opera en la misma clase, en lugar de en instancias individuales de la clase. Se denota como un método que lleva el decorador `@classmethod` justo encima de su definición. A diferencia de los métodos de instancia, los métodos de clase toman la clase como su primer argumento, generalmente llamado `cls`, en lugar de una instancia (`self`).

Los métodos de clase son útiles cuando se necesita realizar una operación que involucra a la clase en sí misma, en lugar de las instancias específicas de la clase. Un caso común de uso es cuando se necesita mantener datos o realizar operaciones que sean relevantes para la clase en general y no para instancias individuales.

Métodos estáticos

Un método estático en Python es una función definida dentro de una clase que no tiene acceso a las instancias de la clase ni a sus atributos. Se denota como un método que lleva el decorador `@staticmethod` justo encima de su definición. A diferencia de los métodos de instancia y de clase, los métodos estáticos no reciben automáticamente ningún argumento especial relacionado con la instancia o la clase.

Los métodos estáticos son útiles cuando se necesita encapsular una funcionalidad que está relacionada con la clase, pero no depende de los atributos de instancia o de la clase. No tienen acceso a los atributos de instancia ni pueden modificar el estado de la instancia. Se utilizan principalmente para agrupar funciones que están relacionadas con la clase, pero que no necesitan acceder a información específica de instancia.

Decoradores

Los decoradores en Python son funciones especiales que se utilizan para modificar o extender el comportamiento de otras funciones o métodos. Los decoradores permiten agregar funcionalidades adicionales a una función sin cambiar su código interno. Se aplican utilizando la sintaxis `@nombre_del_decorador` justo antes de la definición de una función.

Un decorador es esencialmente una función que toma otra función como argumento, realiza alguna operación y devuelve una nueva función con el comportamiento modificado. Esto permite encapsular lógica repetitiva o tareas comunes que se deben realizar antes o después de la ejecución de una función.

```

class Auto:
    """Esta es una clase que representa un automóvil."""
    COLORES_VALIDOS: tuple = ("rojo", "verde", "azul")

    def __init__(self, marca: str, color: str, rodado: int):
        """Inicializa un objeto de la clase Auto."""
        self.marca = marca
        self._color = color
        self.rodado = rodado

    def cambiar_color(self, color: str) -> bool:
        """Cambia el color del automóvil si es un color válido."""
        se_cambio_el_color = False
        if self.validar_color(color):
            self._color = color
            se_cambio_el_color = True
        return se_cambio_el_color

    @classmethod
    def validar_color(cls, color: str) -> bool:
        """ si un color dado es válido.
        return color in cls.COLORES_VALIDOS

    @property
    def color(self) -> str:
        """Obtiene el color actual del automóvil."""
        return self._color

    @staticmethod
    def metodo_estatico():
        """Este es un método estático que no realiza ninguna acción."""
        pass

```

El código anterior define una clase llamada "Auto" que representa un automóvil. La clase tiene atributos como "marca", "color" y "rodado". Además, incluye métodos para cambiar el color del automóvil, validar si un color es válido, obtener el color actual y un método estático a modo ejemplo que no tiene implementación.

Ejercicios prácticos de clases

1. Crea una clase llamada "Estudiante" con los siguientes atributos: nombre, edad y `_promedio`. Implementa tres métodos: `mostrar_informacion()` para mostrar los detalles del estudiante, `actualizar_promedio(nuevo_promedio)` para cambiar el promedio del estudiante y `incrementar_edad()` para aumentar en 1 la edad del estudiante cada vez que se llama. Además, crea una propiedad llamada `promedio` para acceder y modificar el promedio de manera controlada.
2. Crea una clase llamada "Persona" con los atributos nombre, edad y altura. Implementa métodos para mostrar información, incrementar la edad y cambiar la altura.
3. Crea una clase llamada "Perro" con los atributos nombre, raza y edad. Crea métodos para mostrar información, cambiar la raza y aumentar la edad.
4. Crea una clase llamada "Cuenta" con los atributos titular, saldo y `_numero_cuenta`. Crea métodos para mostrar información, depositar y retirar dinero.
5. Crea una clase llamada "Libro" con los atributos título, autor y `_paginas`. Implementa métodos para mostrar información, cambiar el autor y agregar páginas.

Getters y Setters

En Python, los getters y setters son métodos utilizados para acceder y modificar atributos de una clase.

```
class Estudiante:
    def __init__(self, promedio):
        self._promedio = promedio

    def actualizar_promedio(self, nuevo_promedio):
        self._promedio = nuevo_promedio

    @property
    def promedio(self):
        return self._promedio
```

Cómo podemos ver en el ejemplo anterior, hicimos una clase `Estudiante` con una propiedad para obtener el promedio (getter) y un método para actualizarlo (setter).

Ahora, vamos a explicar el código paso a paso. La clase Estudiante tiene un atributo privado `_promedio` que almacena el promedio del estudiante.

El método `actualizar_promedio` es un setter personalizado. Permite actualizar el valor del promedio de un estudiante asignando un nuevo valor al atributo `_promedio`.

El decorador `@property` se utiliza para crear un getter personalizado llamado `promedio`. Esto permite que puedan obtener el valor del promedio del estudiante como si fuera un atributo directo, sin necesidad de usar paréntesis para llamar a un método. Es decir, en lugar de `estudiante.promedio()`, pueden simplemente hacer `estudiante.promedio`.

En Python, por convención, los atributos que comienzan con un guión bajo se consideran "privados", lo que significa que deberían ser tratados como si fueran de acceso restringido desde fuera de la clase. Sin embargo, aún pueden ser accedidos directamente si es necesario.

Ejemplo del uso de los getters y setters según la convención que estamos utilizando

```
# Crear un estudiante
estudiante = Estudiante("Ana", 20, 85.5)

# Obtener el promedio utilizando la propiedad como si fuera el "getter"
print(estudiante.promedio) # Imprimirá 85.5

# Actualizar el promedio utilizando un método que hace de setter
estudiante.actualizar_promedio(90.0)

# Obtener el nuevo promedio utilizando la propiedad (getter)
print(estudiante.promedio) # Imprimirá 90.0
```

Uso de métodos y propiedades en lugar de getters y setters

En Python, el uso de métodos con nombres como `cambiar_valor` (o `set_atributo`) en lugar de `@atributo.setter` es una convención que se basa en los principios de legibilidad, simplicidad y facilidad de mantenimiento del código. La idea es evitar la complejidad innecesaria y favorecer la claridad al proporcionar lógica adicional al establecer u obtener valores de atributos.

La razón detrás de esta convención es la legibilidad y simplicidad, flexibilidad y las futuras modificaciones

Legibilidad y simplicidad

Los métodos como `cambiar_valor` son más explícitos y fáciles de entender para los desarrolladores que puedan estar trabajando en el código. Al leer un método con nombre claro, es evidente lo que hace y qué operación se realiza en él.

Flexibilidad

Al utilizar métodos en lugar de `@atributo.setter`, se puede realizar cualquier lógica adicional que sea necesaria al establecer un valor. Esto podría incluir validaciones, transformaciones de datos o actualizaciones en otros atributos relacionados.

Futuras modificaciones

Si en el futuro se necesita realizar cambios en la lógica al establecer un atributo, los métodos como `cambiar_valor` permiten hacerlo sin afectar la interfaz pública de la clase. En cambio, si se utiliza `@atributo.setter`, cualquier cambio en la lógica podría afectar el comportamiento esperado de otras partes del código que utilizan la propiedad directamente.

La convención de utilizar métodos explícitos en lugar de `@atributo.setter` en Python se basa en la idea de que el código debe ser fácil de entender, mantener y modificar. Aunque `@property` y `@atributo.setter` son herramientas útiles en ciertos contextos, la simplicidad y la claridad suelen ser preferidas en el diseño de código en Python.

Pilares de la Programación Orientada a Objetos

Los pilares fundamentales de la Programación Orientada a Objetos (POO) son conceptos fundamentales que guían la forma en que se estructuran y diseñan los programas. La **abstracción** simplifica objetos, la **encapsulación** oculta detalles internos, la **herencia** permite la creación de nuevas clases basadas en las existentes y el **polimorfismo** permite que objetos de diferentes tipos respondan de manera única a las mismas llamadas de método. Estos pilares combinados promueven la modularidad, la reutilización de código y la flexibilidad en el desarrollo de software orientado a objetos.

Abstracción

La abstracción en programación implica simplificar objetos complejos al modelarlos a través de clases que representan sus características esenciales, centrándose en los aspectos relevantes y descartando los detalles innecesarios. Por ejemplo, al desarrollar un sistema de gestión de recursos humanos, podríamos abstraer el concepto de un "Empleado" en una clase. Esta clase podría contener atributos esenciales como el nombre, el número de empleado y el salario, mientras que aspectos menos relevantes, como la marca de los zapatos que usa, se omiten para mantener el enfoque en la información crítica para el sistema.

Esta práctica de abstracción no sólo simplifica la representación de objetos, sino que también permite a los programadores concentrarse en los aspectos esenciales de un sistema, mejorando así la comprensión y mantenibilidad del código.

```
class Empleado:
    def __init__(self, nombre, salario, beneficio=None):
        self.nombre = nombre
        self.salario = salario
        self.beneficio = beneficio

    def obtener_informacion(self):
        info = f"Nombre: {self.nombre}, Salario: {self.salario}"
        if self.beneficio:
            info += f", Beneficio: {self.beneficio}"
        return info

class EmpleadoTiempoCompleto(Empleado):
    def adquirir_beneficio(self, nuevo_beneficio):
        self.beneficio = nuevo_beneficio
```

Encapsulación

La encapsulación trata de restringir el acceso directo a los componentes internos de un objeto y proporcionar métodos específicos que actúan como llaves para interactuar con dicho objeto. De esta manera, los detalles internos y las implementaciones se mantienen ocultos y protegidos fuera del alcance directo.

Imaginemos que estamos desarrollando un sistema de gestión de recursos humanos para una empresa. En este sistema, la clase "Empleado" representa a los trabajadores de la empresa. Los detalles internos de un empleado, como su salario o su historial de trabajo, están encapsulados, lo que significa que no se pueden acceder directamente desde fuera de la clase "Empleado". En su lugar, se proporcionan métodos públicos, como "obtener_salario()" y "actualizar_historial()", que actúan como las llaves para acceder y modificar esta información.

Esta encapsulación asegura que la integridad de los datos de los empleados se mantenga protegida, evitando que se realicen cambios no autorizados desde fuera de la clase. Además, permite realizar actualizaciones internas en la implementación de la clase "Empleado" sin afectar el código externo que interactúa con los empleados. Así, la encapsulación mejora la seguridad y la robustez del sistema de gestión de recursos humanos.

```
class Empleado:
    def __init__(self, nombre, salario, departamento):
        self.nombre = nombre
        self._salario = salario
        self.departamento = departamento

    def obtener_informacion(self):
        return f"Nombre: {self.nombre}. " \
            f"Salario: {self._salario}. " \
            f"Departamento: {self.departamento}. " \

    def aumentar_salario(self, porcentaje):
        self._salario += (self._salario * porcentaje / 100)

    @property
    def salario(self):
        return self._salario
```

Herencia

La herencia es como construir una nueva herramienta a partir de una ya existente. En este contexto, se trata de crear una nueva clase, llamada subclase o clase derivada, que toma prestados los atributos y métodos de una clase original, llamada clase base o clase padre. Esto fomenta la reutilización de código al evitar la necesidad de escribir nuevamente lo que ya existe y permite la organización jerárquica de las clases.

Imaginemos un sistema de gestión de recursos humanos de una empresa, donde tenemos una clase base llamada "Empleado". Esta clase base contiene atributos y métodos comunes a todos los empleados, como el nombre y la fecha de inicio. Ahora, si queremos representar a diferentes tipos de empleados, como "EmpleadoTiempoCompleto" y "EmpleadoTiempoParcial", podemos crear subclases que hereden de la clase base "Empleado". Estas subclases heredarán los atributos y métodos de la clase base y luego podrán agregar sus propios atributos y métodos específicos, como el salario para un "EmpleadoTiempoCompleto" o las horas trabajadas para un "EmpleadoTiempoParcial".

La herencia facilita la creación de una jerarquía de clases donde las subclases heredan características comunes de la clase base y, al mismo tiempo, pueden tener atributos y métodos únicos. Esto mejora la reutilización del código y permite una organización más clara y estructurada en sistemas como el de gestión de recursos humanos.

```
class Empleado:
    def __init__(self, nombre, salario, departamento):
        pass

class EmpleadoTiempoCompleto(Empleado):
    def __init__(self, nombre, salario, departamento):
        super().__init__(nombre, salario, departamento)
```

Polimorfismo

El polimorfismo permite que objetos de diferentes clases respondan a un mismo método con la misma firma (nombre y argumentos) de maneras específicas y adaptadas a su naturaleza.

En el contexto de un sistema de gestión de recursos humanos, consideremos la clase "Empleado". Todos los empleados, ya sean a tiempo completo o a tiempo parcial, tienen una función llamada "calcular_salario()" que calcula su salario mensual. Sin embargo, la forma en que se calcula este salario puede variar según el tipo de empleado.

El polimorfismo nos permite tratar a todos los empleados de manera uniforme, independientemente de si son empleados a tiempo completo o a tiempo parcial. Aunque ambos tipos de empleados implementan el método "calcular_salario()", cada uno lo hace de acuerdo con sus propias reglas y políticas internas. Esto promueve la flexibilidad en el sistema de gestión de recursos humanos, ya que podemos trabajar con diferentes tipos de empleados sin preocuparnos por los detalles específicos de cómo se calcula su salario.

```
class Persona:
    def __init__(self, nombre):
        self.nombre = nombre


    def obtener_descripcion(self):
        return f"Persona: {self.nombre}"

class Estudiante(Persona):
    def __init__(self, nombre, promedio):
        super().__init__(nombre)
        self.promedio = promedio

    def obtener_descripcion(self):
        return f"Estudiante: {self.nombre}, Promedio: {self.promedio}"

class Profesor(Persona):
    def __init__(self, nombre, especialidad):
        super().__init__(nombre)
        self.especialidad = especialidad

    def obtener_descripcion(self):
        return f"Prof.: {self.nombre}, Especialidad:{self.especialidad}"
```



Diseñar clases en programación orientada a objetos (POO) es una parte fundamental del proceso de desarrollo de software.

Principio de Responsabilidad Única (SRP - Single Responsibility Principle):

Cada clase debe tener una única razón para cambiar. Esto significa que una clase debe tener una única responsabilidad o tarea. Si una clase realiza múltiples tareas o tiene múltiples motivos para cambiar, es más difícil de mantener y entender. Diseña clases que sean cohesivas y centradas en una tarea específica.

Simplificación de conceptos, resumiendo las ideas

Uno de los principios clave en el diseño de clases es la abstracción. La abstracción implica la identificación y representación de los aspectos más importantes y relevantes de un objeto del mundo real en una clase. En lugar de modelar cada detalle minuciosamente, se capturan solo los atributos y comportamientos esenciales.

Este proceso de abstracción simplifica conceptos complejos, lo que es crucial para la comprensión y gestión de datos y procesos en un sistema de software. Al centrarse en lo esencial, el código se vuelve más claro y manejable.

En pocas palabras, extrae los conceptos relevantes de la vida real en tus clases. Identifica las propiedades y comportamientos esenciales que deben ser representados y encapsulados en la clase. Esto te ayudará a crear modelos más claros y precisos.

Control de acceso a los datos, protege los datos

Utiliza la encapsulación para ocultar los detalles internos de una clase y proporciona una interfaz clara para interactuar con ella. Esto se logra utilizando atributos privados y métodos públicos o propiedades para acceder y modificar esos atributos. En Python, puedes usar guiones bajos (`_atributo`) para indicar que un atributo es "privado", aunque aún puede ser accedido desde fuera de la clase.

La encapsulación es esencial para proteger los datos y garantizar la integridad del sistema. Al restringir el acceso directo a los atributos internos, se evita la modificación no autorizada de los datos, lo que puede causar comportamientos inesperados.

Extiende tus clases para compartir comportamientos

Utiliza la herencia cuando tengas una relación "es un" entre clases y la composición cuando tengas una relación "tiene un". La herencia se utiliza para crear nuevas clases basadas en clases existentes, mientras que la composición implica que una clase contiene una instancia de otra. La composición generalmente es preferible a la herencia, ya que es más flexible y evita problemas potenciales de herencia múltiple.

La herencia es uno de los mecanismos más potentes en POO. Permite crear nuevas clases basadas en clases existentes, lo que facilita la reutilización de código y la construcción de relaciones entre las clases. Una clase derivada hereda propiedades y métodos de su clase base, lo que promueve la coherencia y reduce la duplicación de código.

Sin embargo, es importante tener cuidado al usar la herencia. Demasiada herencia puede llevar a una jerarquía de clases compleja y difícil de mantener. En lugar de heredar por capricho, se debe usar cuando existe una relación "es un" lógica y coherente entre las clases.

¿Qué es la relación "es un"?


Esto significa que una clase derivada (o subclase) debe ser un tipo más específico de la clase base (o superclase). Por esto decimos que en la programación orientada a objetos, la herencia se utiliza para establecer una relación "es un" entre clases.

Por ejemplo, si tienes una clase base llamada "Animal", tiene sentido crear subclases como "Perro" o "Gato" porque un perro y un gato son tipos específicos de animales. En este caso, la herencia es lógica y coherente porque refleja la relación "es un".

Crear subclases que no tienen una relación "es un" clara con la clase base. Por ejemplo, si tuvieras una clase base "Animal" y creas una subclase llamada "Coche". No se debe heredar de manera arbitraria, sino que debe haber una razón lógica y coherente para la herencia entre las clases.

Composición y flexibilidad de diseño

Además de la herencia, la composición es otra técnica importante para diseñar clases. En lugar de heredar comportamientos, la composición implica que una clase contiene una instancia de otra clase. Esto permite construir objetos complejos al combinar múltiples componentes independientes.



La composición es especialmente útil cuando no hay una relación "es un" clara entre las clases, pero aún se necesita combinar funcionalidades. Promueve una mayor flexibilidad de diseño, ya que las partes pueden modificarse o reemplazarse sin afectar la estructura principal.

Reducir el acoplamiento al máximo

El acoplamiento se refiere a la dependencia entre clases. Intenta reducir el acoplamiento haciendo que las clases sean independientes entre sí. Esto facilita la modificación y el mantenimiento del código.

Evita la lógica excesiva en los inicializadores

Los inicializadores deben ser simples y eficientes. Evita poner lógica compleja en ellos. Si una clase requiere configuración adicional después de la creación, proporciona métodos de inicialización separados.

Sigue Convenciones de Nomenclatura

Usa nombres descriptivos y significativos para tus clases, métodos y atributos. Sigue las convenciones de nomenclatura de tu lenguaje de programación (por ejemplo, PEP 8 para Python) para que el código sea legible y coherente.

Documenta tu código

Proporciona documentación clara para tus clases, métodos y atributos. Usa comentarios y docstrings para explicar cómo funciona tu código y cómo se deben utilizar tus clases.

Proba tus clases

Escribe pruebas unitarias para tus clases para garantizar que funcionen correctamente y cumplan con sus responsabilidades. Las pruebas automatizadas son esenciales para mantener la calidad del código.

Métodos especiales y personalización de comportamiento

Python, como lenguaje orientado a objetos, ofrece una serie de métodos especiales, también conocidos como "métodos mágicos" o "dunders" (double underscores). Estos métodos permiten personalizar el comportamiento de las clases en situaciones específicas.

El método `__init__` se utiliza para inicializar objetos cuando se crean. El método `__str__` define cómo se debe representar el objeto como cadena. El método `__eq__` se utiliza para comparar objetos. Estos métodos especiales permiten que las clases se comporten de manera personalizada y adecuada a sus necesidades.

¡Ahora veamos código!

Pongamos en práctica los conceptos que hemos discutido anteriormente. A través de ejemplos concretos, exploraremos la abstracción al simplificar ideas complejas en código claro y eficiente. Utilizaremos la encapsulación para proteger datos y garantizar la integridad del sistema. Además, demostraremos cómo la herencia y la composición permiten crear jerarquías de clases y combinar componentes de manera flexible. No olvidaremos el polimorfismo, que permitirá que objetos de diferentes clases compartan un comportamiento común. Así, veremos cómo estas herramientas fundamentales en la programación orientada a objetos cobran vida en la práctica.


```
class Libro:
    def __init__(self, titulo, autor):
        # Usamos _ como convención para atributos "protegidos"
        self._titulo = titulo
        self._autor = autor

    def obtener_info(self):
        return f"{self._titulo} ({self._autor})"

class Novela(Libro): # Ejemplo de herencia: Clase derivada Novela
    def __init__(self, titulo, autor, genero):
        super().__init__(titulo, autor)
        self._genero = genero

    def obtener_info(self):
        return f"{self._titulo} ({self._autor}) - {self._genero}"

class Libreria: # Ejemplo de composición: Clase Libreria
    def __init__(self, nombre):
        self._nombre = nombre
        self._libros = [] # Usamos una lista para almacenar libros

    def agregar_libro(self, libro):
        self._libros.append(libro)

    def listar_libros(self):
        listado = ""
        for libro in self._libros:
            listado += f"{libro.obtener_info()}\n"
        return listado
```



En este código de ejemplo

- Hemos definido una clase base Libro que representa un libro con atributos de título y autor.
- Hemos creado una clase derivada Novela que hereda de Libro e introduce un atributo adicional, el género de la novela.
- La clase Librería utiliza la composición al mantener una lista de libros como parte de su estado interno.
- Se han agregado libros a la librería y se han listado para demostrar la relación "tiene un" en la composición.
- Hemos utilizado el polimorfismo al sobrescribir el método obtener_info en la clase Novela para proporcionar información específica de las novelas.