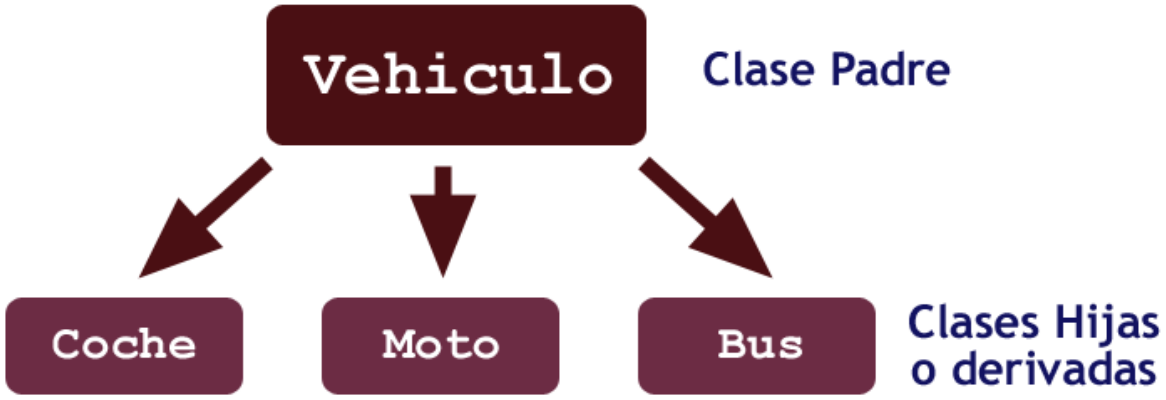
Programación Orientada a Objeto

Herencia

La herencia es uno de los conceptos fundamentales de la Programación Orientada a Objetos (POO), permitiendo que una clase derive propiedades y comportamientos de otra clase. En Python, esto se utiliza para crear nuevas clases basadas en clases existentes, facilitando la reutilización de código y la creación de jerarquías de clases más comprensibles y mantenibles.



Conceptos Básicos de Herencia

Clase Base (o Superclase): La clase de la cual se derivan otras clases.
Clase Derivada (o Subclase): Una clase que hereda de una clase base. Puede añadir o modificar atributos y métodos de la clase base.

Programación Orientada a Objeto

Ejemplo Básico de Herencia en Python

Vamos a crear un ejemplo básico con una clase base llamada `Vehiculo` y dos clases derivadas, `Coche` y `Bicicleta`.

```
class Vehiculo:
 def __init__(self, marca, modelo):
 self.marca = marca
 self.modelo = modelo

 def mostrar_informacion(self):
 return f"Marca: {self.marca}, Modelo: {self.modelo}"

# Clase derivada
class Coche(Vehiculo):
 def __init__(self, marca, modelo, caballos_de_fuerza):
 super().__init__(marca, modelo) # Llama al constructor de la clase base
 self.caballos_de_fuerza = caballos_de_fuerza

 def mostrar_informacion(self):
 info_base = super().mostrar_informacion() # Obtiene información de la clase
base
 return f"{info_base}, Caballos de fuerza: {self.caballos_de_fuerza}"

# Otra clase derivada
class Bicicleta(Vehiculo):
 def __init__(self, marca, modelo, tipo):
 super().__init__(marca, modelo)
 self.tipo = tipo

 def mostrar_informacion(self):
 info_base = super().mostrar_informacion()
 return f"{info_base}, Tipo: {self.tipo}"
```

Uso de las Clases con Herencia

Aquí se muestra cómo crear instancias de estas clases y cómo se utilizan:

```
# Crear instancias de Coche y Bicicleta
mi_coche = Coche("Toyota", "Corolla", 132)
mi_bicicleta = Bicicleta("Trek", "Marlin 5", "Montaña")

# Mostrar información de cada vehículo
print(mi_coche.mostrar_informacion()) # Salida: Marca: Toyota, Modelo: Corolla,
Caballos de fuerza: 132
print(mi_bicicleta.mostrar_informacion()) # Salida: Marca: Trek, Modelo: Marlin
5, Tipo: Montaña
```

Programación Orientada a Objeto

¿Qué Permite la Herencia?

1. **Reutilización de Código**: Evita duplicación al permitir que las subclases utilicen métodos y atributos de la clase base.
2. **Extensibilidad**: Facilita la adición de nuevas características a programas existentes.
3. **Mantenibilidad**: Simplifica los cambios en el programa afectando a una menor cantidad de código.

La herencia hace que el código sea más organizado y modular, facilitando tanto su comprensión como su mantenimiento. Esto es especialmente útil en proyectos grandes donde las jerarquías de clases bien definidas pueden significar una gran diferencia en la eficiencia del desarrollo.

En otras palabras

imagina que la herencia en programación es como en las familias donde los hijos pueden heredar características de sus padres, como el color de ojos o el cabello. En programación, las "clases" (que son como plantillas para crear objetos) pueden actuar como padres e hijos también.

Por ejemplo, si tenemos una clase llamada "Animal" que sabe cómo comer y dormir, podríamos tener una clase "Perro" que hereda de "Animal". Esto significa que "Perro" automáticamente sabe cómo comer y dormir sin que tengamos que escribir el código nuevamente para eso. Además, "Perro" puede aprender trucos nuevos, como ladrar o traer la pelota, que otros animales quizás no hagan.

Así que, en resumen, herencia es una manera en que una clase de programación obtiene (o hereda) capacidades de otra clase, de la misma manera que tú podrías heredar el color de pelo de tus padres. ¡Es una forma de compartir cosas automáticamente!

Programación Orientada a Objeto

ED Ejercicio Dirigido

Vamos a crear un ejercicio práctico dirigido sobre herencia en Python utilizando el concepto de una jerarquía de clases para vehículos. Este ejercicio ilustrará cómo diferentes tipos de vehículos pueden heredar propiedades y métodos de una clase base común, mientras agregan o modifican algunos aspectos específicos para cada tipo de vehículo.

Paso 1: Definir la Clase Base

Primero, definiremos una clase base llamada `Vehiculo`, que incluirá propiedades y métodos comunes que todos los vehículos deben tener.

```
class Vehiculo:
 def __init__(self, marca, modelo, año):
 self.marca = marca
 self.modelo = modelo
 self.año = año

 def mostrar_informacion(self):
 return f"Marca: {self.marca}, Modelo: {self.modelo}, Año: {self.año}"
```

Paso 2: Crear Clases Derivadas

A continuación, crearemos algunas clases derivadas que representen diferentes tipos de vehículos, como `Coche` y `Motocicleta`. Cada una de estas clases heredará de `Vehiculo` y podrá añadir sus propios atributos específicos o modificar los métodos existentes.

```
class Coche(Vehiculo):
 def __init__(self, marca, modelo, año, puertas):
 super().__init__(marca, modelo, año)
 self.puertas = puertas

 def mostrar_informacion(self):
 return f"{super().mostrar_informacion()}, Puertas: {self.puertas}"

class Motocicleta(Vehiculo):
 def __init__(self, marca, modelo, año, cc):
 super().__init__(marca, modelo, año)
 self.cc = cc # Cilindrada

 def mostrar_informacion(self):
 return f"{super().mostrar_informacion()}, Cilindrada: {self.cc} cc"
```

Programación Orientada a Objeto

Paso 3: Utilizar las Clases

Ahora, vamos a crear instancias de cada clase derivada y utilizar los métodos definidos para mostrar cómo funcionan.

```
# Crear instancias de cada tipo de vehículo
mi_coche = Coche("Toyota", "Corolla", 2021, 4)
mi_moto = Motocicleta("Harley Davidson", "Street 750", 2019, 750)

# Mostrar la información de cada vehículo
print(mi_coche.mostrar_informacion()) # Output: Marca: Toyota, Modelo: Corolla,
Año: 2021, Puertas: 4
print(mi_moto.mostrar_informacion()) # Output: Marca: Harley Davidson, Modelo:
Street 750, Año: 2019, Cilindrada: 750 cc
```

Conclusiones del Ejercicio

Este ejercicio muestra cómo la herencia permite reutilizar y extender el código de manera eficiente. Las clases derivadas `Coche` y `Motocicleta` utilizan todos los atributos y métodos de la clase base `Vehiculo`, mientras añaden o personalizan algunos detalles específicos para cada tipo de vehículo. Esto hace que el código sea más manejable y reduce la duplicación, facilitando el mantenimiento y la expansión futura del sistema.

Programación Orientada a Objeto

TA Ejercicio

Enunciado del Ejercicio

Contexto: Eres un desarrollador de software en una empresa de software educativo que está creando una aplicación de simulación de zoológico. Esta aplicación debe manejar diferentes tipos de animales, cada uno con comportamientos específicos para comer y moverse.

Objetivo: Implementar una jerarquía de clases que modele un zoológico. La jerarquía debe tener una clase base `Animal` que defina interfaces comunes para todos los animales. Luego, debes crear clases derivadas para diferentes tipos de animales como `León`, `Ave` y `Pez`, que implementen sus comportamientos específicos.

Requisitos:

1. La clase base `Animal` debe tener métodos `comer()` y `moverse()` que cada subclase implementará de manera específica.
2. La clase `León` debe moverse corriendo y comer carne.
3. La clase `Ave` debe moverse volando y comer semillas.
4. La clase `Pez` debe moverse nadando y comer algas.

Programación Orientada a Objeto

Implementación del Ejercicio

Primero, definiremos la clase base y luego las clases derivadas con sus comportamientos específicos.

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

 def comer(self):
 raise NotImplementedError("Este método debe ser implementado por subclases.")

 def moverse(self):
 raise NotImplementedError("Este método debe ser implementado por subclases.")

class León(Animal):
 def comer(self):
 return f"{self.nombre} está comiendo carne."

 def moverse(self):
 return f"{self.nombre} está corriendo."

class Ave(Animal):
 def comer(self):
 return f"{self.nombre} está comiendo semillas."

 def moverse(self):
 return f"{self.nombre} está volando."

class Pez(Animal):
 def comer(self):
 return f"{self.nombre} está comiendo algas."

 def moverse(self):
 return f"{self.nombre} está nadando."

# Creación de instancias de cada animal
leon = León("Simba")
ave = Ave("Piolín")
pez = Pez("Nemo")

# Demostración de los métodos
print(leon.comer()) # Simba está comiendo carne.
print(leon.moverse()) # Simba está corriendo.
print(ave.comer()) # Piolín está comiendo semillas.
print(ave.moverse()) # Piolín está volando.
print(pez.comer()) # Nemo está comiendo algas.
print(pez.moverse()) # Nemo está nadando.
```

Programación Orientada a Objeto

Conclusión del Ejercicio

Este ejercicio demuestra cómo la herencia puede ser utilizada para crear una estructura clara y organizada de clases en un sistema que requiere manejar múltiples tipos con comportamientos comunes pero implementaciones específicas. La herencia facilita la expansión y el mantenimiento del código, permitiendo añadir nuevos tipos de animales con facilidad y sin modificar el código existente significativamente.

QZ Cuestionario

1. **¿Qué describe mejor la herencia en la programación orientada a objetos?**
    - A) La capacidad de un objeto para cambiar su tipo durante la ejecución del programa.
    - B) Un método por el cual una clase puede adquirir las propiedades y métodos de otra clase.
    - C) La práctica de encapsular todos los datos dentro de una clase.
    - D) La técnica de crear métodos que pueden ser reescritos en cualquier clase.

2. **¿Qué se consigue principalmente a través de la herencia en POO?**
    - A) Mayor velocidad de ejecución del programa.
    - B) Reducción en el uso de la memoria del sistema.
    - C) Reutilización de código, extendiendo la funcionalidad de clases existentes en nuevas clases.
    - D) Aumento de la seguridad del software al restringir el acceso a métodos específicos.

Programación Orientada a Objeto

3. **En el contexto de la herencia, ¿qué es una clase base?**
    - A) Una clase que sólo puede instanciarse, no heredarse.
    - B) La clase de la que otras clases derivan funcionalidad.
    - C) Una clase que no contiene ninguna implementación, sólo definiciones.
    - D) La clase más compleja en un sistema de clases.

4. **¿Qué permite la herencia múltiple en la programación orientada a objetos?**
    - A) Que una clase derive de más de una clase base.
    - B) Que una clase tenga múltiples instancias al mismo tiempo.
    - C) Que una clase implemente múltiples interfaces simultáneamente.
    - D) Que una clase sea parte de múltiples aplicaciones.

5. **¿Cuál de los siguientes escenarios es un ejemplo de uso apropiado de la herencia?**
    - A) Una clase `Forma` que es base para las clases `Círculo`, `Cuadrado` y `Triángulo`.
    - B) Una clase `Usuario` que hereda de las clases `Producto` y `Compra`.
    - C) Una clase `BaseDatos` que deriva de una clase `Cliente`.
    - D) Una clase `Vehículo` que hereda de una clase `Persona`.

- Respuestas correctas: BCBAA

Programación Orientada a Objeto

Encapsulamiento

El encapsulamiento es otro principio fundamental de la Programación Orientada a Objetos (POO), esencial para proteger el estado interno de un objeto y esconder los detalles de implementación de las clases de otros objetos que interactúan con ellos. Esto se logra mediante el uso de métodos de acceso (getters y setters) y la restricción del acceso directo a las variables de instancia.

Objetivos del Encapsulamiento

1. **Seguridad**: Evita que el estado interno del objeto sea cambiado de manera accidental o maliciosa por partes externas.
2. **Simplicidad**: Oculta la complejidad y solo expone una interfaz con la que los otros objetos pueden interactuar.
3. **Flexibilidad**: Permite cambiar la implementación interna sin afectar a quienes usan la clase.

Implementación del Encapsulamiento en Python

Python no tiene modificadores de acceso tradicionales como `private` o `protected` como en otros lenguajes como Java o C++. En Python, el encapsulamiento se realiza mediante convenciones, principalmente utilizando guiones bajos (_) antes del nombre de un atributo para indicar que no se debe acceder directamente.

Programación Orientada a Objeto

Ejemplo de Encapsulamiento en Python

Vamos a ver cómo podríamos encapsular los detalles de una clase `CuentaBancaria`:

```
class CuentaBancaria:
 def __init__(self, titular, saldo_inicial=0):
 self.titular = titular
 self._saldo = saldo_inicial # saldo está "protegido" por convención

 def depositar(self, cantidad):
 if cantidad > 0:
 self._saldo += cantidad
 print(f"Se han depositado {cantidad}€. Saldo actual: {self._saldo}€")
 else:
 print("El monto a depositar debe ser positivo.")

 def retirar(self, cantidad):
 if cantidad > 0 and cantidad <= self._saldo:
 self._saldo -= cantidad
 print(f"Se han retirado {cantidad}€. Saldo actual: {self._saldo}€")
 else:
 print("Fondos insuficientes o monto inválido.")

 def ver_saldo(self):
 return f"Saldo actual: {self._saldo}€"

# Uso de la clase CuentaBancaria
cuenta = CuentaBancaria("Juan Pérez", 1000)
cuenta.depositar(500)
print(cuenta.ver_saldo())
cuenta.retirar(200)
print(cuenta.ver_saldo())
```

Observaciones Importantes

- En el ejemplo, `_saldo` utiliza un único guion bajo, que es una convención para indicar que la variable debe tratarse como protegida (accesible solo dentro de la clase o por subclases).
- No hay manera de forzar el acceso privado en Python; todo se basa en la convención y en la disciplina del desarrollador.
- Puedes usar doble guion bajo (__) para un nivel más fuerte de ocultamiento que cambia el nombre de la variable de manera que sea más difícil acceder desde fuera de la clase (name mangling).

Programación Orientada a Objeto

El encapsulamiento ayuda a mantener el código organizado y seguro, asegurando que los objetos se usen de la manera prevista y facilitando la detección de errores relacionados con el uso incorrecto de las variables internas de un objeto.

En otras palabras

Imagina que tienes una caja fuerte en casa donde guardas cosas importantes como dinero o un diario secreto. Solo tú y las personas a las que les das la combinación pueden abrir la caja fuerte y ver lo que hay dentro. Esto mantiene tus cosas seguras y privadas, lejos de quienes no deben verlas.

El encapsulamiento en programación es parecido a tener esa caja fuerte. Cuando creas un programa y usas encapsulamiento, estás poniendo algunas partes de tu código, como las variables y los métodos, en una "caja fuerte". Solo ciertas partes del programa pueden ver o cambiar lo que está dentro de esa caja fuerte.

Por ejemplo, si estás haciendo un juego y tienes un personaje que tiene una cantidad de vidas, podrías poner esa información en una especie de caja fuerte. Así, solo ciertos métodos en tu código pueden cambiar el número de vidas. Esto ayuda a evitar que algo extraño pase, como que un error en otra parte del código haga que tu personaje pierda todas sus vidas de repente.

En resumen, el encapsulamiento ayuda a mantener partes de tu programa seguras y privadas, permitiendo que solo las partes que realmente necesitan saber sobre ellas puedan acceder y modificar esa información. ¡Es como proteger tus secretos en una caja fuerte!

Programación Orientada a Objeto

ED Ejercicio Dirigido

Un caso práctico de encapsulamiento puede ser un sistema de gestión bancaria, donde se necesita proteger y controlar el acceso a la información sensible de los clientes y las operaciones que pueden realizar sobre sus cuentas. En este ejemplo, vamos a diseñar una clase `CuentaBancaria` en Python que demuestre cómo el encapsulamiento ayuda a mantener seguro el saldo de la cuenta y proporciona métodos para interactuar con este saldo de manera controlada.

Paso 1: Definir la Clase con Propiedades Privadas

Primero, definiremos la clase `CuentaBancaria` con una propiedad privada `_saldo`. El uso del guion bajo indica que esta propiedad no debe ser accesible directamente desde fuera de la clase, siguiendo la convención en Python para indicar un atributo protegido.

```
class CuentaBancaria:
 def __init__(self, titular, saldo_inicial=0):
 self.titular = titular
 self._saldo = saldo_inicial # Atributo protegido

 def depositar(self, monto):
 if monto > 0:
 self._saldo += monto
 print(f"{monto}€ depositados. Saldo actual: {self._saldo}€.")
 else:
 print("El monto a depositar debe ser positivo.")

 def retirar(self, monto):
 if 0 < monto <= self._saldo:
 self._saldo -= monto
 print(f"{monto}€ retirados. Saldo restante: {self._saldo}€.")
 else:
 print("Fondos insuficientes o monto inválido.")

 def ver_saldo(self):
 return f"Saldo actual: {self._saldo}€"
```

Programación Orientada a Objeto

Paso 2: Probar la Clase con Encapsulamiento

Ahora, podemos crear una instancia de `CuentaBancaria` y usar los métodos proporcionados para manipular el saldo, sin acceder directamente a la variable `_saldo`.

```
mi_cuenta = CuentaBancaria("Juan Pérez", 1000)
mi_cuenta.depositar(500)
print(mi_cuenta.ver_saldo())

mi_cuenta.retirar(200)
print(mi_cuenta.ver_saldo())

# Intento de acceso directo a la propiedad protegida (no recomendado y mal visto)
print(mi_cuenta._saldo) # Aunque esto funciona en Python, va en contra de las prácticas recomendadas de encapsulamiento.
```

Beneficios del Encapsulamiento en Este Ejemplo

1. Seguridad: Al encapsular el saldo, nos aseguramos de que solo pueda ser modificado por las operaciones definidas (depósito y retiro), protegiendo los datos contra modificaciones indebidas.
2. Integridad de Datos: Controlando cómo se accede y modifica el saldo, la clase puede garantizar que el estado del saldo siempre sea coherente (por ejemplo, evitando que se retire más dinero del disponible).
3. Simplicidad en la Interfaz: Los usuarios de la clase `CuentaBancaria` no necesitan entender cómo se implementa el manejo del saldo. Solo necesitan saber qué métodos están disponibles para interactuar con la cuenta.

Conclusión

Este ejemplo demuestra cómo el encapsulamiento permite diseñar clases con una interfaz clara y segura, mientras oculta los detalles de implementación y protege la integridad de los datos internos. Esto es esencial en aplicaciones como los sistemas bancarios, donde la seguridad y la precisión son críticos.

Programación Orientada a Objeto

TA Ejercicio

Enunciado del Ejercicio

Contexto: Eres un desarrollador en una compañía de desarrollo de software que está trabajando en una aplicación de gestión financiera personal. La aplicación debe permitir a los usuarios administrar sus cuentas bancarias, incluyendo operaciones como consultar saldo, depositar dinero y retirar dinero.

Objetivo: Implementar una clase `CuentaBancaria` que utilice el principio de encapsulamiento para asegurar que las operaciones sensibles sean manejadas de manera segura y que los detalles de implementación no sean accesibles desde fuera de la clase.

Requisitos:

1. La clase `CuentaBancaria` debe tener un atributo privado para almacenar el saldo de la cuenta.
2. Debe incluir métodos públicos para depositar dinero, retirar dinero y consultar el saldo.
3. Implementar validaciones dentro de los métodos para asegurar que las operaciones son seguras (por ejemplo, no permitir retirar más dinero del que hay en la cuenta).

Solución del Ejercicio

Implementaremos la clase `CuentaBancaria` con los métodos y validaciones necesarios.

```
class CuentaBancaria:
 def __init__(self, titular, saldo_inicial=0):
 self.titular = titular
 self._saldo = saldo_inicial # Atributo privado para el saldo

 def depositar(self, monto):
 if monto > 0:
 self._saldo += monto
 print(f"{monto}€ depositados correctamente. Saldo actual: {self._saldo}€.")
 else:
 print("El monto a depositar debe ser positivo.")

 def retirar(self, monto):
 if monto > 0 and monto <= self._saldo:
 self._saldo -= monto
 print(f"{monto}€ retirados correctamente. Saldo restante: {self._saldo}€.")
 else:
 print("Fondos insuficientes o monto inválido para retirar.")
```

Programación Orientada a Objeto

```
 def consultar_saldo(self):
 return f"El saldo actual de la cuenta es: {self._saldo}€"

# Creación de una instancia de CuentaBancaria
cuenta_de_ana = CuentaBancaria("Ana Pérez", 1000)

# Operaciones
cuenta_de_ana.depositar(500)
print(cuenta_de_ana.consultar_saldo()) # Muestra el saldo después del depósito

cuenta_de_ana.retirar(200)
print(cuenta_de_ana.consultar_saldo()) # Muestra el saldo después de retirar

cuenta_de_ana.retirar(1500) # Intento de retirar más de lo disponible
```

Conclusión del Ejercicio

Este ejercicio muestra cómo el encapsulamiento protege el estado interno de una clase (`CuentaBancaria`) y controla cómo los datos pueden ser modificados o accedidos. Usando métodos públicos que manejan el acceso a datos privados, podemos asegurar que el saldo de la cuenta no sea modificado inapropiadamente y mantener la integridad de los datos dentro de la aplicación. Este patrón es esencial para desarrollar aplicaciones seguras y confiables, especialmente en el ámbito financiero donde la precisión y la seguridad son críticas.

Programación Orientada a Objeto

QZ Cuestionario

1. **¿Qué es el encapsulamiento en la programación orientada a objetos?**
    - A) El proceso de transformar datos durante la ejecución del programa.
    - B) La técnica de ocultar los detalles internos de la implementación de un objeto, exponiendo solo lo que es necesario a través de una interfaz pública.
    - C) La capacidad de un objeto para realizar múltiples funciones al mismo tiempo.
    - D) El método por el cual un objeto puede heredar características de múltiples clases base.

2. **¿Cuál es el principal beneficio del encapsulamiento?**
    - A) Aumenta la velocidad de ejecución de los programas.
    - B) Permite que un programa se ejecute en múltiples plataformas.
    - C) Protege el estado interno de un objeto de accesos no autorizados y modificaciones indebidas.
    - D) Hace que el código sea más difícil de leer y mantener.

3. **¿Cómo se logra típicamente el encapsulamiento en la programación orientada a objetos?**
    - A) Usando bucles y condicionales.
    - B) Mediante la herencia de clases múltiples.
    - C) A través de la definición de métodos y variables privadas o protegidas dentro de una clase.
    - D) Implementando todas las funciones en una sola clase.

Programación Orientada a Objeto

4. **¿Qué permite hacer correctamente el encapsulamiento en el diseño de un software?**
    - A) Cambiar el código interno de un objeto sin afectar a aquellos que lo utilizan.
    - B) Permitir que todos los métodos de un objeto sean accesibles desde cualquier parte del programa.
    - C) Hacer que el software sea menos eficiente pero más seguro.
    - D) Evitar que los desarrolladores utilicen patrones de diseño.

5. **¿Cuál de las siguientes afirmaciones es verdadera respecto al encapsulamiento?**
    - A) Fomenta el acoplamiento fuerte entre clases.
    - B) Impide la reutilización de código entre diferentes programas.
    - C) Evita que los detalles de implementación sean accesibles desde fuera de la clase.
    - D) Reduce la flexibilidad en cómo se pueden usar los objetos.

- Respuesta correcta: BCCAC

Programación Orientada a Objeto

Polimorfismo

El polimorfismo es otro concepto esencial en la Programación Orientada a Objetos (POO) que se refiere a la capacidad de diferentes clases para responder a los mismos métodos o mensajes. Esto permite que se utilicen objetos de diferentes clases de manera intercambiable, siempre y cuando compartan la misma interfaz o método. Esto hace que el software sea más flexible y extensible.

Tipos de Polimorfismo en POO

1. Polimorfismo de Métodos: También conocido como sobrecarga de métodos, ocurre cuando diferentes métodos en la misma clase tienen el mismo nombre pero diferentes listas de parámetros.
2. Polimorfismo de Herencia: Ocurre cuando una operación es realizada de manera diferente en diferentes clases que están relacionadas por herencia. Esto se logra a través de la redefinición de métodos en clases derivadas.

Programación Orientada a Objeto

Ejemplo de Polimorfismo en Python

Vamos a definir una clase base `Animal` y varias clases derivadas que redefinen un método común para demostrar polimorfismo. Cada subclase tendrá una implementación específica del método `hacer_sonido`, lo cual permitirá tratar a todos los animales de manera uniforme en ciertos contextos, mostrando distintos comportamientos.

```
class Animal:
 def hacer_sonido(self):
 raise NotImplementedError("Este método debe ser definido por subclases.")

class Perro(Animal):
 def hacer_sonido(self):
 return "Guau!"

class Gato(Animal):
 def hacer_sonido(self):
 return "Miau!"

class Pato(Animal):
 def hacer_sonido(self):
 return "Cuac!"

# Función que llama al método hacer_sonido, sin necesidad de saber exactamente
qué animal es
def imprimir_sonido(animal):
 print(animal.hacer_sonido())

# Crear instancias de Perro, Gato, y Pato
perro = Perro()
gato = Gato()
pato = Pato()

# Llamar a imprimir_sonido para cada animal
imprimir_sonido(perro) # Output: Guau!
imprimir_sonido(gato) # Output: Miau!
imprimir_sonido(pato) # Output: Cuac!
```

Programación Orientada a Objeto

Beneficios del Polimorfismo

- **Flexibilidad y Reutilización**: Los programas pueden ser escritos de manera más general para trabajar con objetos de cualquier nueva clase que cumpla con la interfaz esperada.
- **Interfaz Unificada**: Permite que las funciones manejen objetos de diferentes clases a través de una interfaz común.
- **Mantenibilidad**: Facilita la adición de nuevas clases sin alterar funciones que utilizan objetos polimórficos.

El polimorfismo es, en esencia, una forma de implementar abstracciones en las que se pueden definir métodos en una clase base y luego adaptarse o sobrescribirse para cada clase derivada, proporcionando así un comportamiento específico mientras se mantiene una interfaz común.

En otras palabras

El polimorfismo en programación es similar. Es como tener un botón mágico que hace que cada figura de acción haga su cosa especial. No necesitas decirle exactamente qué hacer a cada figura; simplemente presionas el botón, y cada una hace lo que sabe hacer mejor.

Por ejemplo, si tienes un programa con diferentes clases como "Perro", "Gato" y "Pájaro", y cada una tiene un método para hacer un sonido. Cuando presionas el botón mágico (o sea, cuando usas el método), el perro ladra, el gato maúlla, y el pájaro canta. Aunque todos están haciendo un "sonido", cada uno lo hace a su manera.

Así que el polimorfismo es una forma de usar objetos de diferentes clases juntos y hacer que cada uno actúe de su manera única con solo decirles que "actúen" o que "hagan su truco". ¡Es como magia!

Programación Orientada a Objeto

ED Ejercicio Dirigido

Vamos a diseñar un caso práctico sobre polimorfismo usando Python para ilustrar cómo diferentes clases pueden implementar la misma interfaz de maneras distintas. En este ejemplo, crearemos un sistema que gestione diferentes tipos de dispositivos de impresión en una oficina. Cada dispositivo (como una impresora láser, una impresora de inyección de tinta y una impresora 3D) implementará un método común para imprimir, pero cada uno funcionará de manera ligeramente diferente.

Paso 1: Definir la Interfaz Común

Primero, definiremos una clase abstracta llamada `Impresora` que servirá como base para todos los tipos de impresoras. Esta clase tendrá un método abstracto `imprimir` que las subclases deben implementar.

```
from abc import ABC, abstractmethod

class Impresora(ABC):
 @abstractmethod
 def imprimir(self, documento):
 pass
```

Paso 2: Crear Subclases Específicas

A continuación, implementaremos varias subclases que representen diferentes tipos de impresoras en la oficina.

```
class ImpresoraLaser(Impresora):
 def imprimir(self, documento):
 return f"Impresión láser de {documento}"

class ImpresoraInyeccion(Impresora):
 def imprimir(self, documento):
 return f"Impresión de inyección de tinta de {documento}"

class Impresora3D(Impresora):
 def imprimir(self, documento):
 return f"Creando objeto 3D desde {documento}"
```

Programación Orientada a Objeto

Paso 3: Utilizar Polimorfismo

Ahora, crearemos una función que pueda recibir cualquier objeto impresora y realizar una tarea de impresión, sin necesidad de saber qué tipo de impresora está manejando. Esto es un ejemplo clásico de cómo el polimorfismo permite que el mismo método sea utilizado para objetos de diferentes clases.

```
def imprimir_documento(impresora, documento):
 print(impresora.imprimir(documento))

# Crear instancias de cada tipo de impresora
impresora_laser = ImpresoraLaser()
impresora_inyeccion = ImpresoraInyeccion()
impresora_3d = Impresora3D()

# Llamar a la función imprimir_documento con diferentes impresoras
imprimir_documento(impresora_laser, "Informe Financiero.pdf")
imprimir_documento(impresora_inyeccion, "Folleto Publicitario.pdf")
imprimir_documento(impresora_3d, "Modelo de Producto.stl")
```

Resultado

Cuando ejecutas este código, verás que cada impresora maneja el documento de manera adecuada a su tecnología, aunque todas comparten el mismo método `imprimir`. El resultado sería algo como esto:

```
Impresión láser de Informe Financiero.pdf
Impresión de inyección de tinta de Folleto Publicitario.pdf
Creando objeto 3D desde Modelo de Producto.stl
```

Conclusión

Este ejemplo muestra cómo el polimorfismo permite diseñar un sistema flexible donde diferentes objetos pueden ser tratados de manera uniforme a través de una interfaz común. Esta capacidad es fundamental en la programación orientada a objetos y ayuda a hacer el código más modular y fácil de extender.

Programación Orientada a Objeto

TA Ejercicio

Enunciado del Ejercicio

Contexto: Eres el jefe de desarrollo de software en una empresa de fabricación de electrodomésticos que produce varios tipos de dispositivos inteligentes para el hogar, como termostatos, refrigeradores y sistemas de iluminación. Cada dispositivo tiene una interfaz de usuario diferente y un conjunto de funcionalidades específicas, pero todos pueden ser controlados a través de una aplicación central.

Objetivo: Implementar un sistema de control centralizado que pueda interactuar con diferentes tipos de dispositivos inteligentes utilizando el principio de polimorfismo, asegurando que cada dispositivo responda de manera adecuada a un conjunto común de comandos, como "encender", "apagar" y "configurar".

Requisitos:

1. Crear una clase abstracta `DispositivoInteligente` que defina los métodos abstractos para los comandos mencionados.
2. Implementar clases derivadas específicas para cada tipo de dispositivo (Termostato, Refrigerador, Sistema de Iluminación) que realicen operaciones concretas al recibir estos comandos.
3. Demostrar cómo la misma llamada de método en diferentes instancias de dispositivos lleva a acciones específicas para cada dispositivo.

Programación Orientada a Objeto

Solución del Ejercicio

Primero, definiremos la clase abstracta y luego las clases derivadas para cada tipo de dispositivo.

```
from abc import ABC, abstractmethod

class DispositivoInteligente(ABC):
 def __init__(self, ubicacion):
 self.ubicacion = ubicacion

 @abstractmethod
 def encender(self):
 pass

 @abstractmethod
 def apagar(self):
 pass

 @abstractmethod
 def configurar(self, configuracion):
 pass

class Termostato(DispositivoInteligente):
 def encender(self):
 return f"Termostato en {self.ubicacion} encendido."

 def apagar(self):
 return f"Termostato en {self.ubicacion} apagado."

 def configurar(self, configuracion):
 return f"Termostato configurado a {configuracion}°C en {self.ubicacion}."

class Refrigerador(DispositivoInteligente):
 def encender(self):
 return f"Refrigerador en {self.ubicacion} encendido, enfriando..."

 def apagar(self):
 return f"Refrigerador en {self.ubicacion} apagado."

 def configurar(self, configuracion):
 return f"Refrigerador configurado a modo '{configuracion}' en
{self.ubicacion}."

class SistemaIluminacion(DispositivoInteligente):
 def encender(self):
 return f"Luces en {self.ubicacion} encendidas."
```

Programación Orientada a Objeto

```
 def apagar(self):
 return f"Luces en {self.ubicacion} apagadas."

 def configurar(self, configuracion):
 return f"Intensidad de luces ajustada a {configuracion}% en {self.ubicacion}."

# Uso de las clases
termostato = Termostato("Sala")
refrigerador = Refrigerador("Cocina")
iluminacion = SistemaIluminacion("Patio")

dispositivos = [termostato, refrigerador, iluminacion]

# Demostración de polimorfismo
for dispositivo in dispositivos:
 print(dispositivo.encender())
 print(dispositivo.configurar(20))
 print(dispositivo.apagar())
```

Conclusión del Ejercicio

Este ejercicio demuestra cómo el polimorfismo permite interactuar con diferentes tipos de objetos a través de una interfaz común, simplificando el manejo de múltiples tipos de dispositivos en un sistema integrado. Cada clase derivada implementa los métodos de la clase base de manera que refleje su funcionamiento particular, permitiendo que un solo código en el sistema central pueda controlar diversos dispositivos sin conocer los detalles específicos de cada uno. Esto es esencial para sistemas de automatización del hogar donde la variedad de dispositivos requiere un enfoque flexible y expansible.

Programación Orientada a Objeto

QZ Cuestionario

1. ¿Qué describe mejor el polimorfismo en la programación orientada a objetos?
    - A) La capacidad de una clase para derivar de múltiples clases base.
    - B) La capacidad de diferentes clases para responder de manera diferente al mismo mensaje o método.
    - C) La habilidad de cambiar el código fuente de un programa durante su ejecución.
    - D) La capacidad de una función para ejecutarse rápidamente.

2. ¿Qué permite el polimorfismo en el desarrollo de software?
    - A) Que un método puede tener diferentes nombres en diferentes clases.
    - B) Que un método pueda tener diferentes implementaciones en diferentes clases.
    - C) Que una clase pueda tener múltiples instancias al mismo tiempo.
    - D) Que un programa pueda correr en diferentes plataformas sin cambios en el código.

3. ¿Cuál es una ventaja clave del polimorfismo en la programación orientada a objetos?
    - A) Permite a las clases compartir el mismo código de implementación para todos los métodos.
    - B) Reduce la memoria utilizada por las aplicaciones.
    - C) Facilita la adición de nuevas clases que actúan de forma similar a las existentes sin modificar el código que utiliza las clases.
    - D) Permite a los métodos cambiar su retorno de tipo durante la ejecución.

Programación Orientada a Objeto

4. ¿Qué tipo de polimorfismo se muestra cuando se usan interfaces o clases abstractas en programación?
    - A) Polimorfismo de sobrecarga.
    - B) Polimorfismo de inclusión.
    - C) Polimorfismo dinámico.
    - D) Polimorfismo estático

5. ¿Cómo se relaciona el polimorfismo con la herencia en la programación orientada a objetos?
    - A) La herencia no está relacionada con el polimorfismo.
    - B) El polimorfismo permite que las clases derivadas implementen métodos de la clase base con comportamientos específicos.
    - C) El polimorfismo impide que las clases derivadas hereden características de la clase base.
    - D) La herencia permite que múltiples clases derivadas utilicen el mismo método sin cambios

.

- Respuesta correcta: BBCCB

Programación Orientada a Objeto

Abstracción

La abstracción es un principio clave de la programación orientada a objetos (POO) que se centra en separar la implementación de un objeto de su especificación o definición. En esencia, permite al desarrollador concentrarse en lo que debe hacer un objeto sin necesidad de conocer cómo se logran esos detalles internamente.

Propósito de la Abstracción

1. Simplificar el diseño: Al ocultar los detalles de implementación complicados y exponer solo los componentes necesarios, se facilita la comprensión y el uso de la clase.
2. Reducir la complejidad: Permite al desarrollador tratar con conceptos a un nivel más general sin preocuparse por los detalles.
3. Incrementar la reutilización: Proporciona una base sobre la cual se pueden construir otras abstracciones.

Implementación de la Abstracción en Python

En Python, la abstracción se puede implementar de varias maneras, incluyendo el uso de clases abstractas a través del módulo `abc` (Abstract Base Classes). Una clase abstracta es aquella que no se puede instanciar directamente y está destinada a ser una clase base para otras subclases. Define métodos que deben ser creados por sus clases derivadas.

Programación Orientada a Objeto

Ejemplo de Abstracción con Clases Abstractas

Vamos a crear una clase abstracta llamada `Vehiculo` que define un método abstracto `moverse`. Esta clase no se puede instanciar, pero se puede heredar y la implementación de `moverse` se puede definir en las subclases.

```
from abc import ABC, abstractmethod

class Vehiculo(ABC):
 @abstractmethod
 def moverse(self):
 pass

class Coche(Vehiculo):
 def moverse(self):
 return "El coche se está moviendo"

class Barco(Vehiculo):
 def moverse(self):
 return "El barco está navegando"

# Intentar crear una instancia de Vehiculo resultaría en un error
# vehiculo = Vehiculo() # Esto lanzaría TypeError

# Instanciación de subclases
coche = Coche()
barco = Barco()

print(coche.moverse()) # Output: El coche se está moviendo
print(barco.moverse()) # Output: El barco está navegando
```

Beneficios de la Abstracción

- Encapsulamiento: Abstracción y encapsulamiento van de la mano para ocultar los detalles internos y mostrar solo lo necesario.
- Flexibilidad: Las clases derivadas pueden ofrecer una implementación completamente diferente de los métodos abstractos.
- Mantenibilidad: Cambios en la implementación de una clase no afectan a aquellas que la utilizan, siempre y cuando la interfaz se mantenga.

En resumen, la abstracción es un pilar fundamental de la POO que ayuda a diseñar sistemas de software robustos, mantenibles y escalables al permitir que los desarrolladores se enfoquen en interacciones a alto nivel, minimizando las preocupaciones sobre las complejidades subyacentes.

Programación Orientada a Objeto

En otras palabras

Imagina que tienes una caja de legos. Con estos legos, puedes construir muchas cosas, como casas, carros, o naves espaciales. Aunque cada uno de estos juguetes se construye con piezas diferentes, todos comienzan con el mismo concepto básico: conectando legos.

La abstracción en programación es como empezar con una idea básica, como los legos. No necesitas saber cómo cada pieza de lego se fabrica o de qué está hecha, solo necesitas saber que puedes unirlas para construir algo más grande.

Por ejemplo, si estás creando un videojuego, podrías tener una idea básica de lo que es un personaje. No necesitas preocuparte por todos los detalles específicos sobre cada tipo de personaje desde el principio. Solo necesitas saber que cada personaje puede moverse, saltar, o recolectar objetos. Esta idea general de un personaje es como tu caja de legos.

Cuando programas, usas abstracción para ignorar los detalles complicados y concentrarte solo en lo importante. Eso hace que sea más fácil construir y manejar tu programa, porque como cuando juegas con legos, te enfocas en crear y divertirte, no en cómo se hacen los legos.

En resumen, la abstracción te ayuda a manejar y organizar tus ideas en programación, permitiéndote concentrarte en construir cosas increíbles paso a paso, sin preocuparte demasiado por todos los pequeños detalles desde el inicio.

Programación Orientada a Objeto

ED Ejercicio Dirigido

Un ejemplo práctico y real de abstracción en programación puede ser el diseño de una aplicación de banca en línea. En una aplicación así, los usuarios pueden realizar diversas operaciones como consultar saldos, hacer transferencias, pagar servicios, etc. Veamos cómo se aplica la abstracción:

Concepto General: Cuenta Bancaria

Para simplificar el desarrollo y la utilización de la aplicación, los detalles complicados sobre los diferentes tipos de cuentas y operaciones se manejan de manera abstracta.

1. Clase Abstracta: Cuenta Bancaria

Primero, se crea una clase abstracta llamada `CuentaBancaria`. Esta clase sirve como un modelo básico que define operaciones comunes que todas las cuentas bancarias deben poder realizar, como:

- Consultar saldo
- Depositar dinero
- Retirar dinero

Estas operaciones son definidas de manera general, sin detalles específicos sobre cómo se ejecutan. Aquí, nos enfocamos en qué se hace, no en cómo se hace exactamente.

```
from abc import ABC, abstractmethod

class CuentaBancaria(ABC):
 def __init__(self, propietario, saldo=0):
 self.propietario = propietario
 self.saldo = saldo

 @abstractmethod
 def consultar_saldo(self):
 pass

 @abstractmethod
 def depositar(self, monto):
 pass

 @abstractmethod
 def retirar(self, monto):
 pass
```

Programación Orientada a Objeto

2. Clases Concretas: Cuenta de Ahorro y Cuenta Corriente

Luego, se definen clases concretas como `CuentaDeAhorro` y `CuentaCorriente` que heredan de `CuentaBancaria`. Estas clases implementan los métodos abstractos según las reglas específicas de cada tipo de cuenta.

```
class CuentaDeAhorro(CuentaBancaria):
 def consultar_saldo(self):
 return f"Saldo de ahorros: {self.saldo}€"

 def depositar(self, monto):
 self.saldo += monto
 print(f"Depositado {monto}€ a cuenta de ahorros")

 def retirar(self, monto):
 if monto <= self.saldo:
 self.saldo -= monto
 print(f"Retirado {monto}€ de cuenta de ahorros")
 else:
 print("Fondos insuficientes")

class CuentaCorriente(CuentaBancaria):
 def consultar_saldo(self):
 return f"Saldo de corriente: {self.saldo}€"

 def depositar(self, monto):
 self.saldo += monto
 print(f"Depositado {monto}€ a cuenta corriente")

 def retirar(self, monto):
 if monto <= self.saldo:
 self.saldo -= monto
 print(f"Retirado {monto}€ de cuenta corriente")
 else:
 print("Fondos insuficientes o descubierto no permitido")
```

Programación Orientada a Objeto

3. Uso Práctico

En la aplicación, cuando un usuario desea realizar alguna operación, el sistema simplemente llama al método correspondiente sin necesidad de saber detalles específicos del tipo de cuenta:

```
mi_ahorro = CuentaDeAhorro("Juan Pérez", 1000)
mi_corriente = CuentaCorriente("Ana López", 500)

mi_ahorro.depositar(200)
print(mi_ahorro.consultar_saldo())

mi_corriente.retirar(100)
print(mi_corriente.consultar_saldo())
```

Conclusión

La abstracción permite a los desarrolladores de la aplicación concentrarse en implementar características generales para manejar cuentas bancarias, sin lidiar con los detalles específicos de cada tipo de cuenta directamente. Esto simplifica el diseño y mejora la mantenibilidad del sistema, permitiendo que sea más fácil agregar nuevas funciones o tipos de cuentas en el futuro.

TA Ejercicio

Enunciado del Ejercicio

Contexto: Eres un desarrollador de software en una empresa de tecnología médica que está trabajando en un sistema de monitorización de salud para pacientes en hospitales. Este sistema debe ser capaz de interactuar con diferentes tipos de dispositivos de monitoreo, como monitores cardíacos, monitores de glucosa y dispositivos de presión arterial.

Objetivo: Utilizar el principio de abstracción para diseñar una interfaz común que permita al sistema central interactuar con todos estos dispositivos de manera uniforme, facilitando la expansión y mantenimiento del sistema al añadir nuevos tipos de monitores.

Programación Orientada a Objeto

Requisitos:

1. Crear una clase abstracta `MonitorSalud` que defina métodos comunes para iniciar monitorización, detener monitorización y obtener datos de monitorización.
2. Implementar clases específicas para cada tipo de dispositivo (MonitorCardiaco, MonitorGlucosa, MonitorPresionArterial) que realicen sus operaciones específicas.
3. Demostrar cómo un único método puede ser utilizado para manejar diferentes dispositivos, promoviendo la flexibilidad y la reutilización del código.

Solución del Ejercicio

Primero, definiremos la clase abstracta y luego desarrollaremos clases específicas para cada tipo de monitor.

```
from abc import ABC, abstractmethod

class MonitorSalud(ABC):
 def __init__(self, ubicacion):
 self.ubicacion = ubicacion

 @abstractmethod
 def iniciar_monitorizacion(self):
 pass

 @abstractmethod
 def detener_monitorizacion(self):
 pass

 @abstractmethod
 def obtener_datos(self):
 pass

class MonitorCardiaco(MonitorSalud):
 def iniciar_monitorizacion(self):
 return "Monitorización cardíaca iniciada."

 def detener_monitorizacion(self):
 return "Monitorización cardíaca detenida."

 def obtener_datos(self):
 return "Datos cardíacos recogidos con éxito."
```

Programación Orientada a Objeto

```
class MonitorGlucosa(MonitorSalud):
 def iniciar_monitorizacion(self):
 return "Monitorización de glucosa iniciada."

 def detener_monitorizacion(self):
 return "Monitorización de glucosa detenida."

 def obtener_datos(self):
 return "Nivel de glucosa en la sangre registrado."

class MonitorPresionArterial(MonitorSalud):
 def iniciar_monitorizacion(self):
 return "Monitorización de presión arterial iniciada."

 def detener_monitorizacion(self):
 return "Monitorización de presión arterial detenida."

 def obtener_datos(self):
 return "Lectura de presión arterial completada."

# Uso de las clases
monitores = [
 MonitorCardiaco("Sala 101"),
 MonitorGlucosa("Sala 102"),
 MonitorPresionArterial("Sala 103")
]

for monitor in monitores:
 print(monitor.iniciar_monitorizacion())
 print(monitor.obtener_datos())
 print(monitor.detener_monitorizacion())
```

Conclusión del Ejercicio

Este ejercicio muestra cómo la abstracción permite diseñar un sistema modular y fácil de expandir. Al definir una interfaz común para todos los dispositivos de monitorización, el sistema puede manejar diferentes dispositivos de manera uniforme, lo que facilita la incorporación de nuevos tipos de monitores sin alterar la arquitectura principal. Esto reduce la complejidad y mejora la mantenibilidad del sistema, factores cruciales en el entorno de la tecnología médica.

Programación Orientada a Objeto

QZ Cuestionario.

1. ¿Qué es la abstracción en la programación orientada a objetos?
    - A) La capacidad de un programa para procesar datos rápidamente.
    - B) Una técnica que se usa para ocultar los detalles de implementación y mostrar solo las operaciones esenciales al usuario.
    - C) Un método para cambiar el comportamiento de una aplicación en tiempo de ejecución.
    - D) La capacidad de un programa para operar en diferentes sistemas operativos.

2. ¿Cuál es el propósito principal de utilizar la abstracción en el diseño de software?
    - A) Mejorar la velocidad de las aplicaciones.
    - B) Facilitar la manipulación directa de la memoria.
    - C) Simplificar el diseño de software al ocultar detalles complejos y exponer solo lo necesario.
    - D) Permitir que el software consuma menos recursos de hardware.

3. ¿Qué tipo de elemento en POO típicamente se utiliza para implementar abstracción?
    - A) Variables globales
    - B) Clases abstractas e interfaces
    - C) Funciones matemáticas
    - D) Bucles de iteración

Programación Orientada a Objeto

4. ¿Cuál de los siguientes es un beneficio de usar la abstracción en la programación orientada a objetos?
    - A) Aumenta la carga en el procesador.
    - B) Reduce la necesidad de pruebas de software.
    - C) Permite al desarrollador gestionar la complejidad reduciendo la interdependencia entre software.
    - D) Hace el código menos reutilizable.

5. ¿Cómo ayuda la abstracción a manejar los cambios en el desarrollo de software?
    - A) Obliga a los programadores a escribir más código.
    - B) Permite cambios en la implementación interna sin afectar a otros partes del programa que usan el código.
    - C) Hace que el software sea menos seguro.
    - D) Reduce la cantidad de datos que el software puede procesar.

- Respuesta correcta: BCBCB

Programación Orientada a Objeto

ED Ejercicio Dirigido

Enunciado del Ejercicio

Contexto: Eres el desarrollador principal en una compañía de seguros que está creando un sistema para manejar diferentes tipos de pólizas de seguros, como automóviles, vivienda y vida. Cada tipo de póliza tiene características y comportamientos únicos, pero también comparten ciertas funcionalidades.

Objetivo: Implementar un sistema de clases en Python que utilice los principios de la Programación Orientada a Objetos (POO) —herencia, encapsulamiento, polimorfismo y abstracción— para modelar esta variedad de pólizas de seguros.

Requisitos:

1. Crear una clase abstracta `PolizaSeguro` que sirva como base para los diferentes tipos de pólizas. Debe incluir atributos comunes y métodos abstractos.
2. Implementar clases derivadas específicas para `SeguroAuto`, `SeguroVivienda` y `SeguroVida`.
3. Utilizar encapsulamiento para proteger los atributos importantes y asegurar que se accedan mediante métodos adecuados.
4. Demostrar polimorfismo permitiendo que los objetos de diferentes clases derivadas se manejen de forma uniforme.
5. Utilizar la herencia para evitar la duplicación de código y promover la reutilización.

Programación Orientada a Objeto

Paso 1: Definir la Clase Abstracta

Primero, definimos la clase abstracta `PolizaSeguro` que incluirá atributos y métodos comunes, así como métodos abstractos que deben ser implementados por las clases derivadas.

```
from abc import ABC, abstractmethod

class PolizaSeguro(ABC):
 def __init__(self, titular, prima, duracion):
 self._titular = titular # Atributo protegido
 self._prima = prima # Atributo protegido, representa el costo de la póliza
 self._duracion = duracion # Atributo protegido, duración en años de la póliza

 @abstractmethod
 def calcular_cobertura(self):
 pass

 def obtener_detalle(self):
 return f"Titular: {self._titular}, Prima: {self._prima}€, Duración:
{self._duracion} años"
```

Programación Orientada a Objeto

Paso 2: Crear Clases Derivadas

Luego, implementamos clases específicas para cada tipo de póliza de seguros, extendiendo la clase base y proporcionando implementaciones concretas de los métodos abstractos.

```
class SeguroAuto(PolizaSeguro):
 def __init__(self, titular, prima, duracion, modelo_auto):
 super().__init__(titular, prima, duracion)
 self._modelo_auto = modelo_auto # Atributo específico del seguro de auto

 def calcular_cobertura(self):
 return f"Cobertura completa para el vehículo modelo {self._modelo_auto}.
{self.obtener_detalle()}"

class SeguroVivienda(PolizaSeguro):
 def __init__(self, titular, prima, duracion, direccion):
 super().__init__(titular, prima, duracion)
 self._direccion = direccion # Atributo específico del seguro de vivienda

 def calcular_cobertura(self):
 return f"Cobertura total de la vivienda en {self._direccion}.
{self.obtener_detalle()}"

class SeguroVida(PolizaSeguro):
 def __init__(self, titular, prima, duracion, beneficio):
 super().__init__(titular, prima, duracion)
 self._beneficio = beneficio # Atributo específico del seguro de vida

 def calcular_cobertura(self):
 return f"Beneficio asegurado de {self._beneficio}€. {self.obtener_detalle()}"
```

Programación Orientada a Objeto

Paso 3: Demostración de Polimorfismo y Uso de las Clases

Finalmente, creamos instancias de cada tipo de póliza y las usamos de manera uniforme para demostrar el polimorfismo.

```
# Creación de instancias
seguro_auto = SeguroAuto("Juan Pérez", 300, 5, "Toyota Corolla")
seguro_vivienda = SeguroVivienda("Ana Gómez", 250, 10, "Calle Falsa 123")
seguro_vida = SeguroVida("Sofía Martín", 400, 20, 100000)

# Lista de pólizas
polizas = [seguro_auto, seguro_vivienda, seguro_vida]

# Imprimir detalles y coberturas
for poliza in polizas:
 print(poliza.calcular_cobertura())
```

Este ejercicio demuestra la aplicación efectiva de herencia, encapsulamiento, polimorfismo y abstracción en un escenario práctico. La clase base proporciona la estructura común para las diferentes pólizas, mientras que las clases derivadas especifican comportamientos únicos. El encapsulamiento protege la integridad de los datos, y el polimorfismo facilita la manipulación uniforme de diferentes tipos de pólizas, simplificando el manejo y la expansión del sistema.

Programación Orientada a Objeto
TA Ejercicio General

Enunciado del Ejercicio

Contexto: Eres un desarrollador en una empresa de software de gestión de eventos que está desarrollando una aplicación para manejar diferentes tipos de eventos como conferencias, conciertos y talleres. Cada tipo de evento tiene características y comportamientos únicos, pero también comparten ciertas funcionalidades.

Objetivo: Implementar un sistema de clases en Python que utilice los principios de la Programación Orientada a Objetos (POO) —herencia, encapsulamiento, polimorfismo y abstracción— para modelar esta variedad de eventos.

Requisitos:

1. Crear una clase abstracta `Evento` que sirva como base para los diferentes tipos de eventos. Debe incluir atributos comunes y métodos abstractos.
2. Implementar clases derivadas específicas para `Conferencia`, `Concierto` y `Taller`.
3. Utilizar encapsulamiento para proteger los atributos importantes y asegurar que se accedan mediante métodos adecuados.
4. Demostrar polimorfismo permitiendo que los objetos de diferentes clases derivadas se manejen de forma uniforme.
5. Utilizar la herencia para evitar la duplicación de código y promover la reutilización.

Programación Orientada a Objeto

Solución

```
from abc import ABC, abstractmethod

class Evento(ABC):
    def __init__(self, nombre, lugar, fecha):
        self._nombre = nombre  # Atributo protegido
        self._lugar = lugar    # Atributo protegido
        self._fecha = fecha    # Atributo protegido

    @abstractmethod
    def describir_evento(self):
        pass

    def obtener_informacion(self):
        return f"Evento: {self._nombre} en {self._lugar}, Fecha: {self._fecha}"

class Conferencia(Evento):
    def __init__(self, nombre, lugar, fecha, tema):
        super().__init__(nombre, lugar, fecha)
        self._tema = tema  # Atributo específico de Conferencia

    def describir_evento(self):
        return f"Conferencia sobre {self._tema}. {self.obtener_informacion()}"

class Concierto(Evento):
    def __init__(self, nombre, lugar, fecha, artista):
        super().__init__(nombre, lugar, fecha)
        self._artista = artista  # Atributo específico de Concierto

    def describir_evento(self):
        return f"Concierto de {self._artista}. {self.obtener_informacion()}"

class Taller(Evento):
    def __init__(self, nombre, lugar, fecha, habilidad):
        super().__init__(nombre, lugar, fecha)
        self._habilidad = habilidad  # Atributo específico de Taller

    def describir_evento(self):
        return f"Taller de {self._habilidad}. {self.obtener_informacion()}"

# Creación y uso de las instancias
evento1 = Conferencia("Cumbre de Innovación", "Centro de Convenciones", "2023-09-15",
"Tecnología Blockchain")
evento2 = Concierto("Noche de Jazz", "Auditorio Ciudad", "2023-10-05", "John Coltrane Quartet")
evento3 = Taller("Desarrollo de Software", "Tech Hub", "2023-11-20", "Programación en Python")

for evento in [evento1, evento2, evento3]:
    print(evento.describir_evento())
```

Programación Orientada a Objeto