Capítulo 10

Herencia

10.1. Introducción

La herencia es un concepto fundamental en la programación orientada a objetos que establece una relación esencial entre clases. Este mecanismo se aplica cuando existen clases que, aunque comparten similitudes notables, no son lo suficientemente idénticas como para ser consideradas como una única clase. En esta situación, la herencia entra en juego para vincular estas clases de manera que los objetos pertenecientes a ambas puedan ser utilizados de manera equivalente, como si efectivamente fueran instancias de la misma clase.

El proceso de herencia posibilita la creación de una nueva clase basada en una clase preexistente, sin la necesidad de duplicar y repetir el código de la clase original. De esta manera, se evita la tarea de copiar y pegar la implementación de una clase para crear una nueva, lo que redundaría en un mantenimiento engorroso y potencialmente propenso a errores.

La herencia implica que una clase derivada (o subclase) hereda propiedades y métodos de una clase base (o superclase). La subclase puede agregar nuevas funcionalidades o modificar las existentes, pero también hereda la estructura y el comportamiento de la superclase.

Esto es útil porque permite organizar y abstraer el código de manera eficiente. Por ejemplo, si se está desarrollando un programa para modelar vehículos, se podría tener una clase base llamada “Vehículo” con propiedades y métodos genéricos como “acelerar” y “frenar”. Luego, se pueden crear subclases tales como “Automóvil” y “Motocicleta” que heredan estas características básicas y agregan sus propias particularidades, ya sea agregando nuevos atributos y métodos o modificando métodos heredados brindando una implementación específica para sus responsabilidades y necesidades.

Al utilizar la herencia, se puede considerar a los objetos pertenecientes a ambas clases como si fueran objetos de la clase base. Esto implica que los objetos heredados heredan tanto las propiedades como los métodos de la clase base, lo que simplifica la organización del código y fomenta la reutilización eficiente de funcionalidades comunes.

10.2. “Es un”

El concepto de la herencia en la programación orientada a objetos se manifiesta mediante la capacidad de definir una clase a partir de otra, estableciendo una relación específica entre ellas. En este contexto, una de las clases involucradas se denomina **clase base**, mientras que la otra se conoce como **clase derivada**.

La clase derivada puede conceptualizarse como un caso especial o una variante de la clase base, lo que implica que todos los objetos pertenecientes a la clase derivada pueden ser considerados como objetos de la clase base. Esta relación se expresa de manera precisa con la afirmación: “un objeto de la clase derivada ES UN objeto de la clase base”. En general, es una práctica útil verificar las relaciones de herencia con esa frase: si la frase no tiene sentido, generalmente es síntoma de que las clases no deben estar relacionadas o lo hacen pero por otro tipo de relación.

Por ejemplo:

Clase base Cliente y Clase derivada ClienteOnline En este contexto, afirmar que “un cliente online ES UN cliente” es totalmente válido. La clase derivada ClienteOnline representa una variante específica de la clase base Cliente, pero todos los clientes en línea son, en última instancia, clientes en sí mismos.

Clase base Auto y Clase derivada Rueda La afirmación “una rueda ES UN auto” es inevitablemente incorrecta, como también lo es “un auto ES UNA rueda”. Esta situación pone en evidencia que tales clases no pueden estar relacionadas mediante herencia. Sin embargo, pueden estar relacionadas mediante algún otro tipo de relación, por caso, composición.

10.3. Implementación

10.3.1. Bloque class

Para implementar herencia en Python no se agregan muchos elementos sintácticos, en general las clases se redactan de una forma bastante similar a aquellas no involucradas en este tipo de relación.

En la clase base no se debe realizar ninguna modificación si se identifica la necesidad de una clase derivada. En la clase derivada se indica en la cabecera del bloque class, a continuación del nombre de la misma, el nombre de la clase base encerrado entre paréntesis:

```
class Derivada(Base):
```

10.3.2. Función super

En los métodos de las clases derivadas se dispone de una referencia denominada `super` la cual referencia al objeto que recibió el mensaje, pero permitiendo acceder únicamente a los miembros heredados. Es, por lo tanto, muy similar a self, pero no requiere ser recibida como un parámetro.

La referencia super es ocasionalmente utilizada de forma errónea para acceder a cualquier miembro declarado en la clase base. Esto evidencia una notoria ignorancia del concepto de la herencia: todo miembro de la clase base es heredado y por lo tanto accesible mediante la referencia self.

El uso de super debe estar focalizado en acceder a aquellos miembros que no pueden ser alcanzados mediante self. Esta situación ocurre puntualmente con dos casos bien delimitados. El método constructor __init__ no es heredado y muy habitualmente es necesario invocar explícitamente desde la derivada al constructor de la base. Asimismo es necesario utilizar super para realizar llamadas a un método de la clase base desde su redefinición en la derivada, ya que si se lo intenta invocar con self, se iniciaría una llamada recursiva.

10.3.3. Constructor

El método constructor de cada clase generalmente recibe parámetros para asignar valores iniciales a los atributos. Cuando se programa un constructor en una clase derivada, el caso más usual es el de recibir parámetros para inicializar los atributos de la propia clase y también los heredados.

A continuación el constructor de la derivada puede realizar una invocación explícita al constructor de la clase base a través de la referencia super:

```
class Derivada(Base):
    def __init__(self, params):
        super.__init__(params)
```

Capítulo 11

Polimorfismo

11.1. Introducción

11.1.1. Concepto

El polimorfismo es un principio fundamental de la programación orientada a objetos que permite que objetos de diferentes clases sean tratados como objetos de una clase común. Esta característica facilita que un mismo método pueda comportarse de manera diferente según el tipo específico del objeto que lo recibe, proporcionando flexibilidad y extensibilidad al código.

El término ”polimorfismo”deriva del griego y significa ”muchas formas”, lo cual describe precisamente su naturaleza: un mismo nombre de método puede tener múltiples implementaciones según la clase que lo contenga. En el contexto de la herencia, el polimorfismo se manifiesta cuando las clases derivadas redefinen métodos heredados de la clase base, proporcionando implementaciones específicas que se ajusten a sus particulares necesidades y características.

11.1.2. Sobreescritura de métodos

La sobreescritura de métodos es el mecanismo mediante el cual una clase derivada proporciona una implementación específica de un método que ya existe en su clase base. En Python, la sintaxis para sobreescribir un método es extremadamente sencilla: simplemente se define un método en la clase derivada con el mismo nombre que el método de la clase base que se desea redefinir. No se requieren palabras clave especiales ni anotaciones adicionales.

```
class Vehiculo:
    def acelerar(self):
        print("El vehículo está acelerando")

    def describir(self):
        print("Este es un vehículo genérico")

class Automovil(Vehiculo):
    def acelerar(self):
        print("El automóvil acelera suavemente")

    def describir(self):
        print("Este es un automóvil con cuatro ruedas")

class Motocicleta(Vehiculo):
    def acelerar(self):
        print("La motocicleta acelera rápidamente")
```

11.1.3. Uso de super

El método especial `__str__` es particularmente útil para demostrar el uso de `super()` en el contexto del polimorfismo, ya que frecuentemente es necesario extender la representación de cadena de la clase base en lugar de reemplazarla completamente.

Cuando se sobreescribe el método `__str__` en una clase derivada, es común utilizar `super()` para invocar la implementación de la clase base y luego agregar información específica de la clase derivada. Esto permite construir representaciones de cadena más completas y jerárquicas.

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

    def __str__(self):
        return f"Persona: {self.nombre}, {self.edad} años"

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

    def __str__(self):
        return f"{super().__str__()}, Carrera: {self.carrera}"

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

    def __str__(self):
        return f"{super().__str__()}, Materia: {self.materia}"
```

En este ejemplo, las clases `Estudiante` y `Profesor` extienden la representación de cadena de la clase `Persona` agregando información específica. El uso de `super().__str__()` permite reutilizar la lógica de la clase base y evitar la duplicación de código.

11.2. Clases y métodos abstractos

11.2.1. Concepto

Una clase abstracta es aquella que no puede ser instanciada directamente y que generalmente contiene uno o más métodos abstractos. Los métodos abstractos son declaraciones de métodos que deben ser implementados obligatoriamente por las clases derivadas, pero que no poseen implementación en la clase base.

Las clases abstractas sirven como plantillas o contratos que definen qué métodos deben implementar las clases derivadas, garantizando así una interfaz común entre diferentes implementaciones. Esto es especialmente útil cuando se desea definir un comportamiento común pero la implementación específica varía según cada subclase.

11.2.2. Uso del Decorador @abstractmethod

Python proporciona el módulo `abc` (Abstract Base Classes) para implementar clases y métodos abstractos. Para crear un método abstracto, se utiliza el decorador `@abstractmethod` y la clase debe heredar de `ABC`.

```python
from abc import ABC, abstractmethod

class Figura(ABC):
    def __init__(self, nombre):
        self.nombre = nombre

    @abstractmethod
    def calcular_area(self):
        pass

    @abstractmethod
    def calcular_perimetro(self):
        pass

    def describir(self):
        return f"Figura: {self.nombre}"

class Rectangulo(Figura):
    def __init__(self, ancho, alto):
        super().__init__("Rectángulo")
        self.ancho = ancho
        self.alto = alto

    def calcular_area(self):
        return self.ancho * self.alto
```

```
    def calcular_perimetro(self):
        return 2 * (self.ancho + self.alto)

class Circulo(Figura):
    def __init__(self, radio):
        super().__init__("Círculo")
        self.radio = radio

    def calcular_area(self):
        return 3.14159 * self.radio ** 2

    def calcular_perimetro(self):
        return 2 * 3.14159 * self.radio
```

Si se intenta instanciar la clase `Figura` directamente o crear una subclase que no implemente todos los métodos abstractos, Python generará un error `TypeError`. Esto garantiza que todas las clases derivadas cumplan con el contrato establecido por la clase abstracta.

11.3. Interfaces

11.3.1. Implementación

Las interfaces definen un conjunto de métodos que una clase debe implementar, sin proporcionar la implementación de dichos métodos. En UML, las interfaces se representan como contratos que especifican qué servicios debe proporcionar una clase, pero no cómo debe proporcionarlos.

Aunque Python no tiene una palabra clave específica para interfaces como otros lenguajes, se pueden implementar utilizando clases abstractas que contengan únicamente métodos abstractos. Esta aproximación permite replicar el concepto de interfaces de UML en Python.

```
from abc import ABC, abstractmethod

class IReproductorMultimedia(ABC):
    @abstractmethod
    def reproducir(self):
        pass

    @abstractmethod
    def pausar(self):
        pass

    @abstractmethod
    def detener(self):
        pass

class IAlmacenamiento(ABC):
    @abstractmethod
    def guardar(self, datos):
        pass

    @abstractmethod
    def cargar(self):
        pass

class ReproductorMP3(IReproductorMultimedia, IAlmacenamiento):
    def __init__(self):
        self.estado = "detenido"
        self.archivos = []

    def reproducir(self):
        self.estado = "reproduciendo"
        print("Reproduciendo archivo MP3")

    def pausar(self):
        self.estado = "pausado"
        print("Reproducción pausada")

    def detener(self):
        self.estado = "detenido"
        print("Reproducción detenida")
```

```
    def guardar(self, archivo):
        self.archivos.append(archivo)
        print(f"Archivo {archivo} guardado")

    def cargar(self):
        return self.archivos
```

11.3.2. Implementación de múltiples interfaces

Python permite que una clase implemente múltiples interfaces mediante herencia múltiple, lo cual es equivalente al concepto de implementación múltiple de interfaces en UML. Esto proporciona gran flexibilidad para diseñar clases que cumplan con varios contratos diferentes.

```
class IVehiculoTerrestre(ABC):
    @abstractmethod
    def conducir(self):
        pass

class IVehiculoAcuatico(ABC):
    @abstractmethod
    def navegar(self):
        pass

class VehiculoAnfibio(IVehiculoTerrestre, IVehiculoAcuatico):
    def __init__(self, modelo):
        self.modelo = modelo

    def conducir(self):
        print(f"{self.modelo} está conduciendo en tierra")

    def navegar(self):
        print(f"{self.modelo} está navegando en agua")
```

Esta implementación permite que un objeto `VehiculoAnfibio` sea tratado polimórficamente como un `IVehiculoTerrestre` o como un `IVehiculoAcuatico`, dependiendo del contexto en el que se utilice, replicando así el comportamiento de las interfaces múltiples de UML en Python.