Herencia, Clases Abstractas, Polimorfismo, Interfaces y MRO

Informe — Programación II

Paulo Villalobos
Profesor: Guido Mellado

7 de octubre de 2025

Resumen

Este informe explica y ejemplifica los conceptos de Herencia, Clases Abstractas, Polimorfismo, Interfaces y Method Resolution Order (MRO) en Python, incluyendo ejemplos de código, una fórmula y recomendaciones sobre buenas prácticas de diseño.

Índice

1. Introducción

En este informe se revisan los conceptos fundamentales de la orientación a objetos en Python y mecanismos relacionados como los Protocolos y el Duck Typing. Se incluyen ejemplos y una explicación del orden de resolución de métodos (MRO).

2. Herencia

En un problema textual, la herencia se identifica por la relación "es un/a". Por ejemplo, un Estudiante **es una** Persona, por lo tanto hereda sus atributos y métodos.

La herencia permite que una clase (subclase) reutilice comportamiento y atributos definidos en otra (superclase). Ejemplo simple:

```
class Animal:
    def __init__(self,nombre):
        self.nombre = nombre
    def dormir(self):
        print(f"{self.nombre} está durmiendo")
```

2.1. Herencia múltiple

Python permite herencia múltiple: una clase puede heredar de varias. Hay que tener cuidado con el MRO para evitar ambigüedades.

3. Clases Abstractas

Las clases abstractas definen una interfaz parcial que obliga a las subclases a implementar ciertos métodos. Se usan con `abc.ABC` y `@abstractmethod`.

```
from abc import ABC, abstractmethod

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

    @abstractmethod
    def hacer_sonido(self):
        pass

    def dormir(self):
        print(f"{self.nombre} está durmiendo...")
```

Ejemplo de implementación:

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

4. Polimorfismo

El polimorfismo permite que diferentes clases respondan al mismo mensaje (método) con comportamientos distintos. También puede referirse a extender métodos de la superclase usando `super()`.

```
class Persona:
    def presentación(self) -> str:
        return f"Hola, soy {self.nombre}"

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

    def presentación(self) -> str:
        base = super().presentación()
        return f"{base} y estudio {self.carrera}"
```

5. Interfaces (simuladas en Python)

En Python se simula una interfaz usando clases abstractas donde todos los métodos son abstractos. Ejemplo: `Volador`.

```
from abc import ABC, abstractmethod

class Volador(ABC):
    @abstractmethod
    def volar(self): pass

    @abstractmethod
    def aterrizar(self): pass
```

6. Protocolos y Duck Typing

Los Protocolos (módulo `typing`) permiten tipado estructural (duck typing). Si un objeto tiene los métodos esperados, puede usarse aunque no herede de una clase en particular.

```
from typing import Protocol

class Nadador(Protocol):
    def nadar(self) -> None: ...
```

6.1. Ejemplo Duck Typing

```
class Pato:
    def cuack(self): print("Cuac!")
    def nadar(self): print("Estoy nadando")

class Persona:
    def cuack(self): print("Imito un cuac!")
    def nadar(self): print("Me meto al agua...")
```

7. Method Resolution Order (MRO)

Explica cómo Python busca un método en la jerarquía de clases. Ejemplo:

```
class A:
    def saludo(self): print("Hola desde A")
class B(A): pass
class C(B): pass

obj = C()
obj.saludo()  # Busca en C, luego B, luego A
```

En caso de herencia múltiple, el orden sigue la regla C3 linearization.

8. Fórmula de ejemplo

Ejemplo de fórmula simple (requerido por la guía): La fórmula del área del círculo:

$$A = \pi r^2$$

9. Conclusión

En conclusión, estos conceptos permiten escribir programas más estructurados y reutilizables. El uso de herencia, clases abstractas e interfaces ayuda a definir comportamientos comunes, mientras que el polimorfismo y el MRO dan flexibilidad al diseño orientado a objetos en Python.