Informe sobre Programación Orientada a Objetos en Python

Herencia, Clases Abstractas, Polimorfismo, Interfaces y MRO

Nicolás Castillo Azócar

Profesor: Guido Mellado

Asignatura: Programación II

11 de octubre de 2025

Índice

1. Introducción

La Programación Orientada a Objetos (POO) es un paradigma de programación que busca modelar el software utilizando conceptos inspirados en el mundo real, como clases y objetos. Los pilares de la POO incluyen la abstracción, la herencia, el polimorfismo y el encapsulamiento.

En este informe se abordarán los siguientes conceptos clave:

- **Herencia**: reutilización de código a través de clases derivadas.
- **Clases Abstractas**: definición de plantillas obligatorias para subclases.
- **Polimorfismo**: capacidad de usar métodos comunes en diferentes tipos de objetos.
- **Interfaces**: contratos que garantizan consistencia entre distintas clases.
- **Method Resolution Order (MRO)**: mecanismo de resolución de métodos en herencia múltiple.

Adicionalmente, se incluirán secciones opcionales sobre `super()`, *duck typing* y *dunder methods*, útiles para enriquecer el dominio de POO en Python.

2. Herencia

La **herencia** permite que una clase hija (o subclase) adquiera atributos y métodos de una clase padre (o superclase). Esto fomenta la reutilización de código y la jerarquía lógica de los objetos.

Ejemplo básico:

```
class Persona:
    def presentacion(self):
        print("Soy una persona")

class Estudiante(Persona):
    def presentacion(self):
        # Uso de super() para extender el comportamiento
        super().presentacion()
        print("...y tambi n soy estudiante")

e = Estudiante()
e.presentacion()
# Salida:
# Soy una persona
# ...y tambi n soy estudiante
```

La relación entre una clase y su subclase suele representarse con el principio *"es-un"*. Por ejemplo: un estudiante **es una** persona, un círculo **es una** figura geométrica.

3. Clases Abstractas

Las **clases abstractas** son aquellas que no pueden ser instanciadas directamente. Sirven como plantillas que obligan a las subclases a implementar ciertos métodos. En Python se definen con el módulo `abc`.

```python
from abc import ABC, abstractmethod

class Figura(ABC):
    @abstractmethod
    def area(self):
        pass

class Circulo(Figura):
    def __init__(self, radio):
        self.radio = radio
    def area(self):
        return 3.14 * self.radio ** 2
```

Esto asegura que toda clase derivada de `Figura` debe implementar el método `area()`.

4. Polimorfismo

El **polimorfismo** consiste en la capacidad de utilizar un mismo método en diferentes clases, adaptando su comportamiento.

Ejemplo de polimorfismo con clases

```python
class Perro:
    def sonido(self): return "Guau!"

class Gato:
    def sonido(self): return "Miau!"

animales = [Perro(), Gato()]
for animal in animales:
    print(animal.sonido())
# Salida: Guau! / Miau!
```

Polimorfismo en funciones internas

En Python, funciones como `len()` son polimórficas: pueden operar con strings, listas, tuplas o diccionarios.

5. Interfaces

En lenguajes como Java, las interfaces son tipos que definen un conjunto de métodos sin implementación. En Python, se simulan con clases abstractas.

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

class Pajaro(Volador):
```

```
    def volar(self): print("El p jaro vuela")
```

También existen alternativas modernas como los **protocolos**, que permiten declarar interfaces de forma implícita.

6. Method Resolution Order (MRO)

Cuando existe herencia múltiple, Python utiliza el algoritmo **C3 linearization** para definir el orden de búsqueda de métodos.

```
class A: pass
class B(A): pass
class C(A): pass
class D(B, C): pass

print(D.mro())
# [D, B, C, A, object]
```

Este orden garantiza coherencia y evita ambigüedades. En general, Python busca de izquierda a derecha en la declaración de herencia.

7. Ejemplo Matemático en POO

Un ejemplo clásico es calcular el área de un círculo. La fórmula es:

$$A = \pi r^2$$

En Python:

```
import math
print(math.pi * 3**2)  # 28.27
```

8. Esquema Conceptual

El siguiente esquema muestra la relación de conceptos:

$$\text{Clases} \rightarrow \text{Herencia} \rightarrow \text{Polimorfismo}$$

$$\text{Clases Abstractas} \Rightarrow \text{Interfaces}$$

9. Conclusión

Los conceptos de POO permiten construir programas más claros, reutilizables y fáciles de mantener. Python ofrece mecanismos simples pero potentes para aplicar herencia, abstracción, polimorfismo e interfaces. El conocimiento del MRO resulta clave al trabajar con herencia múltiple.

Temas Opcionales (Ampliación)

Sección Opcional / Ampliación:

La función `super()`

Permite llamar métodos de la superclase desde la subclase. Se usa principalmente en constructores y para extender métodos ya existentes.

Sección Opcional / Ampliación:

Duck Typing

El *duck typing* se basa en la frase: "Si camina como un pato y suena como un pato, probablemente sea un pato". En Python importa el comportamiento, no el tipo.

```
def hazlo_volar(obj):
    obj.volar()

class Avion:
    def volar(self): print("El avi n vuela")

class SuperHeroe:
    def volar(self): print("El superh roe vuela")

hazlo_volar(Avion())
hazlo_volar(SuperHeroe())
```

Sección Opcional / Ampliación:

Dunder Methods

Los *dunder methods* (doble subrayado, como __init__) permiten redefinir comportamientos internos de Python.

```
class Vector:
    def __init__(self, x, y):
        self.x, self.y = x, y
    def __add__(self, otro):
        return Vector(self.x + otro.x, self.y + otro.y)

v1 = Vector(1,2)
v2 = Vector(3,4)
v3 = v1 + v2
```

Aquí redefinimos el operador `+` gracias al método especial `__add__`.