ChatGPT

# Herencia y Polimorfismo en Python

En la programación orientada a objetos, **herencia** y **polimorfismo** son dos pilares fundamentales. La **herencia** permite definir relaciones jerárquicas entre clases: una *clase hija* puede reutilizar atributos y métodos de una *clase padre*. Esto fomenta la reutilización del código y la modularidad [1] [2]. Por su parte, el **polimorfismo** (del griego *muchas formas*) significa que distintos objetos pueden ser tratados mediante una misma interfaz común, ejecutando cada uno su propia versión de un método [3] [4]. En Python, gracias al tipado dinámico, no es necesario que las clases compartan explícitamente una interfaz: basta con que implementen los métodos que se quieren llamar [3] [4]. Esto hace que el código sea más flexible y fácil de mantener.

## ¿Qué es el Polimorfismo?

El polimorfismo permite escribir código genérico que "funciona para objetos de diferentes clases" mientras comparten el mismo nombre de método [4] [5]. Por ejemplo, si tenemos varias clases que implementan un método `hablar()`, podemos llamar siempre a `objeto.hablar()` sin preocuparnos del tipo concreto de objeto. Cada clase ofrecerá su propia implementación de `hablar()`, y el resultado variará según la clase real del objeto. En otras palabras, el mismo método invocado en objetos de distintas clases produce *diferentes comportamientos* [4] [5].

- **Flexibilidad:** Podemos pasar objetos de distintas subclases a una misma función; ésta llamará al método correspondiente de cada objeto sin cambios adicionales.
- **Reutilización y mantenimiento:** Al trabajar con jerarquías de clases, las nuevas subclases pueden extender el comportamiento heredado y adaptarlo. El polimorfismo facilita que una función operativa trabaje "a través" de la clase base sin conocer todos los detalles de las clases hijas [5].

> *"El polimorfismo nos ayuda a escribir código que puede funcionar con objetos de clases diferentes, aunque esas clases tengan comportamientos distintos"* [5]. Esto ejemplifica cómo podemos resolver problemas (por ejemplo, imprimir detalles o procesar objetos) de forma genérica, sin necesidad de duplicar código para cada tipo de clase.

## ¿Qué es la Herencia?

La herencia permite que una clase hija obtenga automáticamente los atributos y métodos de su clase padre [1]. Así, las subclases **extienden** o **especializan** el comportamiento general de la clase padre. Las ventajas principales de la herencia son la **reutilización del código** (no hay que reescribir todo desde cero) y la construcción de **jerarquías lógicas** de clases [1] [2]. Por ejemplo, podemos definir una clase genérica `Persona` con atributos como `nombre` e `id`, y luego crear subclases `Estudiante` o `Profesor` que hereden estos atributos y agreguen otros propios.

- **Clases padre/hija:** La sintaxis en Python es `class ClaseHija(ClasePadre):`. La clase hija obtiene todo el comportamiento de la clase padre y puede añadir nuevos métodos o atributos propios [6].
- **Reutilización:** Al definir funcionalidad común en la clase padre, las clases hijas la reutilizan. Esto evita la duplicación de código.

- **Relación *"es un"*:** La herencia modela relaciones "es un": por ejemplo, un `Estudiante` **es un** `Persona`.

Como señala DataCamp, "la herencia fomenta la reutilización del código … y admite el polimorfismo, lo que hace que el código sea más flexible" [2]. En otras palabras, gracias a la herencia podemos aplicar polimorfismo: varias subclases pueden compartir la misma interfaz definida en la clase base pero comportarse de modo distinto al sobrescribir métodos.

## Herencia simple y múltiple

- **Herencia simple (única):** Una clase hija hereda de una sola clase padre. Es el caso más común. Por ejemplo, `class Estudiante(Persona):` indica que `Estudiante` hereda solo de `Persona` [6]. La herencia simple permite especializar el comportamiento de la clase padre añadiendo nuevos atributos o métodos en la hija.
- **Herencia múltiple:** Python permite que una clase herede de varias clases padre a la vez. Por ejemplo: `class EstudianteTrabajador(Estudiante, Empleado):` haría que `EstudianteTrabajador` tenga miembros de `Estudiante` y de `Empleado`. Esto combina los atributos y métodos de cada padre [7]. La herencia múltiple puede ser útil pero requiere cuidado, pues pueden surgir conflictos (por ejemplo, si dos padres definen un mismo método). Python resuelve estos conflictos siguiendo el *Orden de Resolución de Métodos* (MRO).

En resumen, la **herencia simple** implica un único padre, mientras que la **herencia múltiple** permite varios padres simultáneamente [8]. En ambos casos, las subclases pueden *sobrescribir* métodos heredados para ajustarlos a sus necesidades específicas.

## Sobrescritura de métodos

Cuando una subclase redefine un método que ya existe en la clase padre, hablamos de **sobrescritura** (override). Esto permite que la clase hija modifique o amplíe el comportamiento heredado. Como explica Oregoom, "en una subclase es posible sobrescribir los métodos heredados de la superclase; esto significa que puedes redefinir un método en la subclase para que tenga un comportamiento diferente al de la superclase" [9].

Por ejemplo, si la clase `Persona` define un método `get_detalles()`, la clase `Estudiante` puede sobrescribirlo para incluir información adicional (como el grado o materias). Al llamar a `get_detalles()` sobre un objeto de tipo `Estudiante`, se ejecutará la versión de `Estudiante` en lugar de la de `Persona`. Esto es esencial para el polimorfismo: diferentes clases comparten el mismo nombre de método, pero cada una lo implementa de forma apropiada a su contexto.

## Uso de la función `isinstance()`

La función incorporada `isinstance(obj, Clase)` permite verificar en tiempo de ejecución si `obj` es instancia de `Clase` o de alguna de sus subclases [10]. Esto es útil cuando trabajamos con jerarquías de clases y necesitamos distinguir comportamientos según el tipo concreto del objeto. Por ejemplo, podemos contar cuántos elementos de una lista de objetos son de un tipo determinado. Como indica Oregoom, "`isinstance()` *devuelve* `True` *si el objeto es una instancia de la clase indicada o de alguna de sus subclases*" [10]. Es una forma segura de comprobar tipos sin depender de comparaciones directas de clases.

## Ejemplos en código

A continuación se presentan ejemplos que ilustran los conceptos anteriores:

- **Herencia y polimorfismo con animales:** Definimos una clase base `Animal` y subclases `Perro` y `Gato` que sobrescriben el método `hablar()`. Luego creamos una lista de animales y, sin distinguir sus tipos, llamamos a `hablar()` en cada uno. También usamos `isinstance()` para contar cuántos perros hay en la lista.

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

    def hablar(self):
        return f"{self.nombre} hace un sonido."

class Perro(Animal):
    def hablar(self):
        return f"{self.nombre} dice 'guau'."

class Gato(Animal):
    def hablar(self):
        return f"{self.nombre} dice 'miau'."

animales = [Perro("Luna"), Gato("Mia"), Perro("Rex")]

for a in animales:
    print(a.hablar())

# Contar cuántos objetos son de tipo Perro
num_perros = sum(isinstance(a, Perro) for a in animales)
print("Número de perros en la lista:", num_perros)
```

En este código **cada** `Perro` **y** `Gato` **hereda de** `Animal`. Ambos sobrescriben `hablar()`, por lo que al recorrer la lista y llamar a `a.hablar()`, se ejecuta la versión específica de cada clase (polimorfismo). La función `isinstance(a, Perro)` devuelve `True` sólo para los objetos de clase `Perro` (o sus subclases, si las hubiera) [10], permitiendo contar los perros de forma sencilla.

- **Herencia y sobrescritura en un sistema escolar:** Creamos una clase `Persona` con un método `get_detalles()`, y dos subclases que añaden información extra. Luego definimos una función `imprimir_detalles()` que acepta cualquier `Persona` y llama a su método, mostrando el polimorfismo en acción.

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

    def get_detalles(self):
```

```
        return f"Nombre: {self.nombre}, ID: {self.id}"

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

    def get_detalles(self):
        return f"Nombre: {self.nombre}, ID: {self.id}, Grado: {self.grado}"

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

    def get_detalles(self):
        return f"Nombre: {self.nombre}, ID: {self.id}, Asignatura:
{self.asignatura}"

def imprimir_detalles(persona):
    print(persona.get_detalles())

# Creamos objetos de tipo Estudiante y Profesor
lista_personas = [
    Estudiante("Ana", "E001", "10mo"),
    Profesor("Luis", "P001", "Matemáticas"),
    Estudiante("Carlos", "E002", "11mo")
]

for persona in lista_personas:
    imprimir_detalles(persona)
```

Aquí, `Estudiante` y `Profesor` **heredan de** `Persona` y **sobrescriben** el método `get_detalles()` [9]. La función `imprimir_detalles()` acepta objetos de tipo `Persona` (o cualquier subclase), demostrando polimorfismo: internamente llama al `get_detalles()` adecuado para cada objeto. De este modo, el mismo código (`imprimir_detalles`) resuelve el problema de mostrar información de estudiantes y profesores sin necesidad de escribir lógica separada para cada caso.

Estos ejemplos ilustran cómo usar herencia y sobrescritura para implementar polimorfismo en Python: **las subclases redefinen métodos heredados**, permitiendo tratar objetos heterogéneos de forma uniforme. Además, funciones como `isinstance()` ayudan a distinguir el tipo real de un objeto cuando sea necesario [10]. Con estos conceptos podemos diseñar soluciones limpias y mantenibles para problemas de diversa complejidad en Python.

**Fuentes:** Definiciones y explicaciones extraídas de recursos educativos sobre POO en Python [3] [1] [11] [12] [8] [4] [9] [10].

---

1 2 4 5 6 7 8 11 12 Herencia Python: Buenas prácticas para el código reutilizable | DataCamp
https://www.datacamp.com/es/tutorial/python-inheritance

3 Polimorfismo | El Libro De Python
https://ellibrodepython.com/polimorfismo-en-programacion

9 10 ▷ Herencia en Python → 〖 Tutorial de Python 〗
https://oregoom.com/python/herencia/