

◆ Relaciones entre Clases en Programación Orientada a Objetos

📌 Introducción General

- En la vida real, los objetos están relacionados entre sí, y en POO se debe modelar esa colaboración.
 - Las **relaciones entre clases** expresan formas de **acoplamiento** entre ellas.
 - Las principales relaciones son:
 - Asociación
 - Agregación
 - Composición
 - Herencia (se trata por separado)
-

🔗 Asociación

✅ Definición:

- Relación estructural en la que un objeto "conoce" a otro.
- Es una relación **débil**, no implica propiedad ni dependencia fuerte.
- Puede ser **bidireccional** o **unidireccional**.

📌 Características:

- Los objetos se relacionan para colaborar.

- Pueden crearse y destruirse de forma independiente.
- La asociación puede tener multiplicidad (uno a uno, uno a muchos, muchos a muchos).
- Se representa con **líneas simples** en UML.

Ejemplos relevantes del material:

✓ Asociación circular (Provincia - Gobernador):

- Cada objeto necesita una referencia al otro.
- Se resuelve:
 - Inicializando con **None** y usando métodos **set** para establecer la relación luego de la creación.
 - Evitando imports cruzados (problema común de circular import en Python modular).

```
class Provincia:
    def __init__(self, nombre, habitantes, gobernador=None):
        ...
        self.__gobernador = gobernador
    def setGobernador(self, gobernador):
        self.__gobernador = gobernador
```

✓ Asociación en contexto real: Cliente - Factura - Promoción de Shopping

- Cada cliente tiene una lista de facturas.
- Cada factura puede aplicar descuentos según el monto.
- Si el cliente tiene más de tres compras, accede a un **cupón del 12%** del total.

Lógica incluida:

- Descuentos por monto.
- Cálculo de totales y cupón posterior.
- Uso de listas para almacenar facturas.

```
if importe > 250000:  
    descuento = 25000  
elif importe > 100000:  
    descuento = 10000
```

✓ Asociación con clase intermedia: **Prescripción** entre **Médico** y **Paciente**

- **Prescripción** modela una relación **muchos a muchos** entre médico y paciente.
- Cada médico puede hacer múltiples prescripciones a distintos pacientes.
- Cada paciente puede tener múltiples prescripciones de uno o varios médicos.

```
class Prescripcion:  
    def __init__(..., medico, paciente):  
        self.__medico = medico  
        self.__paciente = paciente  
        self.__medico.addPrescripcion(self)  
        self.__paciente.addPrescripcion(self)
```

Clase Asociación (también llamada Clase de Asociación)

¿Qué es?

- Una clase especial que representa la **relación misma**, no solo las entidades que se relacionan.

- Se utiliza cuando se necesita **atributos adicionales** en la relación.

✓ Ejemplo: Registro Civil - Persona - ActaNacimiento

- **ActaNacimiento** es la clase asociación entre **RegistroCivil** y **Persona**.
- Guarda atributos propios de la relación: fecha, número de acta, número de libro.

```
class ActaNacimiento:
    def __init__(self, nroActa, nroLibro, fecha, persona,
registroCivil):
        self.__persona = persona
        self.__registrocivil = registroCivil
```

◆ Agregación

✓ Definición:

- Es un caso especial de asociación.
- Modela una relación “**todo/parte**” donde las partes pueden existir independientemente del todo.
- La destrucción del objeto contenedor **no implica** destrucción de los contenidos.
- Representa una relación de **contenedor contenido**, **sin fuerte dependencia**.

📌 Características clave:

- Se representa con un **rombo blanco** en UML.
- Los objetos contenidos **pueden pertenecer a múltiples objetos contenedores**.

- La relación se establece por referencia, no por composición física.
 - **Los ciclos de vida no están acoplados.**
-

Ejemplo completo del material: Restaurante – Pedido

Clases implicadas:

- **Bebida**: tiene denominación, presentación, precio.
- **Plato**: tiene descripción y precio.
- **Mozo**: tiene ID y nombre.
- **Pedido**: tiene bebidas, platos, un número de mesa y un mozo.

```
class Pedido:
    def __init__(self, numeroMesa, mozo, bebida=None, plato=None):
        self.__bebidas = []
        self.__platos = []
        if bebida:
            self.addBebida(bebida, 1)
        if plato:
            self.addPlato(plato, 1)
```

Métodos de agregación:

- `addBebida(bebida, cantidad)`
- `addPlato(plato, cantidad)`
- `cerrarPedido()` imprime todo y calcula el total.

```
def cerrarPedido(self):
    print('Bebidas')
    for bebida in self.__bebidas:
        print(bebida.getDenominacion(), bebida.getPrecio())
    ...
    print('Total a pagar: ', total)
```

Ejecución:

- Se crean varias bebidas y platos.
- Se agregan a diferentes pedidos.
- Los objetos **bebida** y **plato** son **compartidos entre pedidos distintos**, mostrando la independencia del ciclo de vida.

Comparación rápida con otras relaciones:

Relación	Acoplamiento	Ciclo de vida	Notación UML
	o	compartido	
Asociación	Bajo	Independiente	Línea
Agregación	Medio	Independiente	Rombo blanco
Composición	Alto	Compartido	Rombo negro

Conclusión (Síntesis)

- Las **relaciones entre clases** permiten modelar la interacción realista entre objetos.
- La **asociación** es el vínculo más general: flexible, débil, sin propiedad.
- La **agregación** agrega contención: objetos compuestos pero independientes.

◆ Herencia Simple

✓ ¿Qué es la herencia?

- Es un mecanismo fundamental en la POO que permite crear nuevas clases **reutilizando código existente**.
 - Una clase hija (subclase) **hereda** atributos y métodos de una clase padre (superclase).
 - El objetivo es **evitar la duplicación de código** y permitir la **extensión o especialización** del comportamiento.
-

📌 Características principales

- La subclase puede:
 - Usar los métodos y atributos heredados.
 - Sobrescribir (reescribir) métodos de la clase padre.
 - Agregar nuevos métodos o atributos propios.
 - Python permite usar `super()` para:
 - Invocar el constructor de la clase base.
 - Acceder a métodos heredados de forma ordenada y controlada.
 - Todas las clases en Python **heredan implícitamente de `object`**, la superclase base universal.
-

🔧 Ejemplo del archivo: **Circulo** → **Cilindro**

```
class Circulo:

    def __init__(self, radio):

        self.__radio = radio

    def superficie(self):

        return math.pi * self.__radio**2

    def getRadio(self):

        return self.__radio


class Cilindro(Circulo):

    def __init__(self, radio, altura):

        super().__init__(radio) # Invoca constructor de
Circulo

        self.__altura = altura

    def superficie(self):

        # Reutiliza método de la clase base con super()

        superficieLateral = 2 * math.pi * self.getRadio() *
self.__altura

        superficieBase = 2 * super().superficie()

        return superficieLateral + superficieBase
```

🧠 **Ventajas del uso de `super()`:**

- Evita codificación duplicada.
 - Facilita la extensión de comportamiento sin alterar el original.
 - Permite mantener el principio DRY (*Don't Repeat Yourself*).
-

Adicional: Métodos heredados de `object`

- Incluso si no lo indicamos explícitamente, toda clase hereda de `object`.
- Esto permite usar métodos como:
 - `__init__`, `__str__`, `__eq__`, `__lt__`, etc.
- Estos pueden ser sobrescritos para personalizar comportamiento.

```
print(dir(Punto)) # Muestra todos los métodos heredados
```

◆ **Herencia Múltiple**

¿Qué es?

- Una clase puede heredar de **más de una clase base**.
 - Python lo permite (a diferencia de otros lenguajes como Java), pero requiere cuidado.
 - Puede haber conflictos si varias clases base **definen un método con el mismo nombre**.
-

Riesgos y solución

- Cuando se repiten nombres de métodos en clases base, puede haber ambigüedad.
- Python resuelve esto con el **MRO (Method Resolution Order)**:
 - Define el orden exacto en que se buscarán métodos y atributos en las superclases.
 - El orden es **de izquierda a derecha y de arriba hacia abajo** en la jerarquía.

MRO en acción:

```
print(Ayudante.mro())
```

```
# → [Ayudante, Docente, Alumno, Persona, object] (o el orden  
que se use)
```

Ejemplo completo del archivo: **Persona** → **Docente y Alumno** → **Ayudante**

Clases base:

```
class Persona:

    def __init__(self, dni, apellido, nombre, ...):

        self.__dni = dni

        self.__apellido = apellido

        self.__nombre = nombre
```

```
class Docente(Persona):  
    def __init__(self, dni, apellido, nombre, ..., sueldo):  
        super().__init__(dni, apellido, nombre, ...)  
        self.__sueldo = sueldo
```

```
class Alumno(Persona):  
    def __init__(self, dni, apellido, nombre, ..., carrera):  
        super().__init__(dni, apellido, nombre, ...)  
        self.__carrera = carrera
```

Clase derivada con herencia múltiple:

```
class Ayudante(Docente, Alumno):  
    def __init__(self, dni, apellido, nombre, ..., concepto,  
horasLIA=0):  
        super().__init__(dni, apellido, nombre, ..., sueldo) #  
Llama según el MRO  
        self.__concepto = concepto  
        self.__horasLIA = horasLIA
```

⚠ Importancia del orden en la herencia

```
class Ayudante(Docente, Alumno)

# ≠

class Ayudante(Alumno, Docente)
```

- Cambiar el orden **modifica el MRO**, lo que cambia:
 - Qué constructor se invoca primero.
 - Qué métodos se ejecutan si están duplicados.
-

Variación con ****kwargs** para simplificar constructores

- Cuando las clases intermedias tienen muchos argumentos, se puede usar ****kwargs** para simplificar:

```
class Cilindro(Circulo):

    def __init__(self, radio, **kwargs):

        super().__init__(radio)

        self.__altura = kwargs['altura']
```

Consejos prácticos sobre herencia múltiple

- Usar herencia múltiple **con moderación**.
- Mantener un diseño limpio para evitar ambigüedades.

- Usar `super()` siempre que sea posible para permitir una resolución ordenada.
 - Verificar el orden MRO si se producen comportamientos inesperados.
-

✓ Conclusión

- La **herencia simple** permite extender clases de forma estructurada y reutilizar código fácilmente.
- La **herencia múltiple** añade potencia pero también complejidad: debe usarse con cuidado.
- Python maneja la ambigüedad con el **MRO**, que debe conocerse y respetarse para evitar errores.
- Los ejemplos de la unidad muestran cómo aplicar correctamente ambos tipos de herencia, incluyendo buenas prácticas como el uso de `super()` y `**kwargs`

◆ Composición

✓ ¿Qué es?

- La composición es una relación entre clases tipo "**todo/parte**", pero con **acoplamiento fuerte**.
 - Si el objeto contenedor (el "todo") se destruye, **también se destruyen sus partes**.
 - Es una forma más rígida de la agregación.
 - El ciclo de vida de los objetos está **estrechamente vinculado**.
-

Características clave

- Representa **dependencia total**: la parte **no puede existir sin el todo**.
 - Se representa con un **rombo negro** en diagramas UML.
 - Se implementa creando objetos internos en el constructor del objeto principal.
 - Se suele usar cuando un objeto está **compuesto exclusivamente** por otros, creados en conjunto.
-

Ejemplo del material: **Profesor y CuentaCampus**

Contexto:

- Cuando se crea un objeto **Profesor**, **automáticamente se crea una cuenta** de acceso al campus virtual.
- Esa cuenta está compuesta por:
 - ID de usuario
 - Nombre de usuario (nombre+apellido+dominio)
 - Clave (DNI por defecto)

Implementación:

```
class CuentaCampus:
```

```
    __dominio='@unsj-cuim.edu.ar'
```

```
__idCuenta = 0

@classmethod
def getIdCuenta(cls):
    cls.__idCuenta += 1
    return cls.__idCuenta

def __init__(self, idUsuario, nombreUsuario, clave):
    self.__idUsuario = idUsuario
    self.__nombreUsuario = nombreUsuario
    self.__clave = clave

class Profesor:
    def __init__(self, dni, apellido, nombre):
        self.__dni = dni
        self.__apellido = apellido
        self.__nombre = nombre
        idCuenta = CuentaCampus.getIdCuenta()
        dominio = CuentaCampus.getDominio()
        usuario = nombre.lower() + apellido.lower() + dominio
        self.__cuentaCampus = CuentaCampus(idCuenta, usuario, dni)
```





Comportamiento al eliminar:

- Al hacer `del profesor`, también se borra la cuenta.

```
def __del__(self):  
    print("Borrando cuenta de usuario...")  
    del self.__cuentaCampus
```

Diferencias clave con agregación:

- En agregación, el objeto parte **puede seguir existiendo** luego de destruir el objeto todo.
- En composición, **no puede**.

Relación	Ciclo de vida compartido	Representación UML	Independencia
Agregación	 No	Rombo blanco	 Sí
Composición	 Sí	Rombo negro	 No

◆ Polimorfismo

✓ ¿Qué es?

- El **polimorfismo** permite que objetos de distintas clases respondan de manera distinta al **mismo mensaje** (método).
 - Se basa en dos pilares:
 - **Herencia** (comparten interfaz)
 - **Vinculación dinámica** (el método real se decide en tiempo de ejecución)
-

📌 Tipos en Python:

- **De subtipo**: una subclase puede usarse como si fuera la superclase.
 - **Dinámico**: los métodos se resuelven al momento de ejecución, no en compilación.
 - Esto lo permite el modelo de clases y objetos dinámico de Python.
-

🧠 ¿Cómo se implementa?

- Una clase base define un método (a veces sin cuerpo = método abstracto).

- Las clases hijas sobrescriben ese método con su comportamiento específico.
 - Un contenedor (lista, arreglo, etc.) almacena objetos de distintas subclases.
 - Al invocar el método, se llama al de la clase correspondiente.
-

Ejemplo completo: **Cuerpo**, **Cilindro**, **ParalelepipedoRectangulo**

Clase base:

```
class Cuerpo:

    def __init__(self, altura):

        self.__altura = altura

    def superficieBase(self):

        pass # Método abstracto

    def volumen(self):

        return self.superficieBase() * self.__altura
```

Subclases:

```
class Cilindro(Cuerpo):

    def __init__(self, altura, radio):

        super().__init__(altura)
```

```

        self.__radio = radio

    def superficieBase(self):
        return math.pi * self.__radio**2

class ParalelepipedoRectangulo(Cuerpo):

    def __init__(self, altura, lado1, lado2):
        super().__init__(altura)

        self.__lado1 = lado1

        self.__lado2 = lado2

    def superficieBase(self):
        return self.__lado1 * self.__lado2

```

Clase contenedora: Arreglo

- Se usa un arreglo de NumPy para almacenar cuerpos.

```

class Arreglo:

    def __init__(self, dimension=10):
        self.__cuerpos = np.empty(dimension, dtype=object)

        self.__actual = 0

    def agregarCuerpo(self, unCuerpo):
        self.__cuerpos[self.__actual] = unCuerpo

```

```
        self.__actual += 1

    def calcularVolumenCuerpos(self):

        for i in range(self.__actual):

            print(str(self.__cuerpos[i]), 'Volumen =',
self.__cuerpos[i].volumen())
```

Resultado:

```
Cilindro, altura = 5, radio = 6, Volumen = 565.49
```

```
Paralelepípedo Rectángulo, altura = 3, lado a=5, lado b=4, Volumen =
60.00
```

El mismo método `volumen()` funciona distinto según el objeto → eso es **polimorfismo**.

Herramientas asociadas:

- `isinstance(obj, Clase)` para verificar tipo de objeto.
- `type(obj)` también, pero **no reconoce subclases**, por eso se prefiere `isinstance`.

Ejemplo:

```
if isinstance(obj, Cilindro):
```

```
    ...
```

⚠ Consideraciones clave:

- El método `superficieBase()` debe ser implementado en cada subclase.
- Si no se sobrescribe y se llama `volumen()`, dará error.
- Se recomienda usar clases abstractas (`pass` o `abc.ABC`) para forzar la implementación.

✓ Conclusión

- En **composición**, los objetos contenidos son creados y destruidos junto con el objeto que los contiene. Se usa para modelar una dependencia total entre partes.
- El **polimorfismo** permite escribir código genérico y flexible, haciendo que distintas clases respondan de manera adecuada a la misma interfaz.
- Ambos conceptos son pilares de la P00: **composición** para diseño estructurado y **polimorfismo** para diseño flexible y reutilizable.

◆ Manejo de Errores y Excepciones

✅ ¿Qué es un error en un programa?

- Un **error** es una falla que impide que un programa funcione correctamente.
 - Pueden ocurrir:
 - En tiempo de desarrollo
 - En tiempo de ejecución
 - En producción, si no se detectan a tiempo
-

📌 Tipos de errores (según Python y el material):

1. Errores de sintaxis

- Violaciones de la gramática del lenguaje.
- Son detectados por el intérprete **antes de ejecutar el programa**.
- Ejemplos:
 - Falta de dos puntos (:)
 - Mal uso de paréntesis, comillas, etc.

```
if x == 5  
  
    print("error") # Error de sintaxis
```

2. Errores en tiempo de ejecución

- El código es sintácticamente correcto, pero **falla al ejecutarse**.
- Ocurren, por ejemplo, por:

- División por cero
- Índice fuera de rango
- Variable no definida

```
print(1 / 0)           # ZeroDivisionError  
print(lista[10])       # IndexError  
print(variable_invalida) # NameError
```

3. Errores semánticos (lógicos)

- El programa **corre sin fallar**, pero **hace algo incorrecto**.
- Son los más difíciles de detectar.
- Ejemplo: usar **+** entre un **int** y un **str** puede generar un **TypeError**.

⚠ ¿Qué ocurre si no se maneja un error?

- Python **lanza una excepción**.
- Si no es capturada, el programa **se detiene** y muestra un mensaje de error.

◆ Excepciones

✅ ¿Qué es una excepción?

- Una **excepción** es un mecanismo que se dispara cuando ocurre un error **en tiempo de ejecución**.
 - Permite **interrumpir el flujo normal del programa** y ejecutar código especial para manejar ese error.
-

Mecanismo de manejo en Python

Python tiene una estructura robusta de control de errores:

```
try:

    # Código que puede fallar

except NombreDeError:

    # Código que se ejecuta si ocurre ese error

else:

    # (Opcional) Si no ocurre ningún error

finally:

    # (Opcional) Siempre se ejecuta, ocurra o no error
```

Ejemplo del material:

```
try:

    num1, num2 = eval(input('Ingrese dos números separados por coma: '))

    resultado = num1 / num2

    print('El resultado es:', resultado)
```



```
except ZeroDivisionError:

    print('La división por cero es un error.')

except SyntaxError:

    print('Error de sintaxis. Use coma entre los números.')

except:

    print('Entrada errónea.')

else:

    print('No hubo excepciones.')

finally:

    print('Este bloque se ejecuta siempre.')
```

¿Qué excepciones pueden capturarse?

- Python tiene muchas clases de excepción estándar:
 - `ValueError`, `ZeroDivisionError`, `IndexError`, `TypeError`, `NameError`, etc.
 - También se pueden capturar **todas** con `except:` (aunque **no se recomienda** salvo en pruebas).
-

Excepciones personalizadas

- Se pueden definir **clases propias de excepciones**, que heredan de `Exception` o `BaseException`.
- Útiles para representar **errores de negocio** o reglas de validación personalizadas.

 **Ejemplo del material: `ErrorAuto`, `ChoqueAuto`, `ColorNoValido`**

- `ChoqueAuto`: se lanza si un auto choca circulando a más de 30 km/h.
- `ColorNoValido`: se lanza si el color del auto no está definido.
- Se almacenan referencias al objeto y un mensaje explicativo.

```
class ChoqueAuto(Exception):
    def __init__(self, auto1, auto2, velocidad):
        self.auto1 = auto1
        self.auto2 = auto2
        self.velocidad = velocidad
        super().__init__(f"Choque entre {auto1} y {auto2} a {velocidad} km/h. Llamar al 911.")
```

◆ **Assert y Raise**

 **assert**

 **¿Qué es?**

- Una herramienta de **depuración** que verifica si una condición lógica se cumple.
- Si la condición es falsa, lanza una **excepción AssertionError**.

Sintaxis:

```
assert condición, "Mensaje opcional"
```


Ejemplo del material:

```
class Character:

    def __init__(self, character):

        assert len(character) == 1, "Debe ser un carácter"

        self.__character = character
```

 Si se pasa "ab", se lanza: `AssertionError: Debe ser un carácter`

raise

¿Qué es?

- **raise** lanza una **excepción manualmente**, ya sea estándar o personalizada.
- Útil para validar condiciones propias del programa o negocio.

Sintaxis:

```
raise TipoDeExcepcion("Mensaje opcional")
```

Ejemplo del material:

```
def agregarCliente(self, cliente):  
    if not isinstance(cliente, Cliente):  
        raise TypeError("El objeto no es de tipo Cliente")
```

✓ Luego se captura con un bloque `try-except`:

```
try:  
    gestor.agregarCliente("cadena")  
except TypeError:  
    print("Error de tipos")
```

Conclusión

- Python tiene un **sistema sólido y flexible de manejo de errores**, que permite detectar, controlar y personalizar el comportamiento ante fallas.
- Las instrucciones `try-except`, `assert` y `raise` son herramientas fundamentales para escribir código robusto.
- Las **excepciones personalizadas** permiten adaptarse a necesidades específicas del dominio del programa.
- **Assert** se recomienda solo en desarrollo y pruebas; no en producción.
- **Raise** debe usarse para validar y proteger la lógica del sistema ante condiciones erróneas.

◆ Interfaces en Python (según el material)

✓ ¿Qué es una interfaz?

- Una **interfaz** define un **conjunto de métodos** que una clase debe implementar.
 - No especifica **cómo** se implementan los métodos, solo **qué** métodos deben existir.
 - Es un **contrato de comportamiento**.
-

📌 En Python no existen interfaces formales como en Java o C#, pero se pueden simular:

◆ Alternativas comunes:

1. Clases base con métodos abstractos (usando **pass**)
 2. Uso de **abc** (módulo **abc**, abstract base class)
 3. Uso de bibliotecas externas como **zope.interface** (mencionada en la bibliografía)
-

🔧 ¿Cómo se simula una interfaz en Python?

- Se crea una clase **base** con métodos que **no tienen implementación** (usando **pass**).
- Las subclases deben sobrescribir esos métodos.

🔧 Ejemplo del material:

```
class Cuerpo:
```

```
def superficieBase(self):  
    pass # Método sin implementación (abstracto)  
  
def volumen(self):  
    return self.superficieBase() * self.__altura
```

- Aquí, `Cuerpo` se comporta como una interfaz:
 - Define el método `superficieBase()`, que **debe ser implementado por las subclases**.

Beneficios de usar interfaces (simuladas o reales):

- Definen un **contrato obligatorio** para las subclases.
- Permiten escribir código **genérico y polimórfico**.
- Fomentan la **cohesión** del diseño.
- Mejoran la capacidad de **testeo y mantenimiento**.

Consideraciones importantes:

- Si una subclase no sobrescribe el método abstracto y se lo invoca, Python lanza un `TypeError` o ejecuta `pass`, lo que puede llevar a errores lógicos.
- Para una implementación **formal**, se recomienda usar:

```
from abc import ABC, abstractmethod
```

```
class Cuerpo(ABC):  
  
    @abstractmethod  
  
    def superficieBase(self):  
  
        pass
```

Este enfoque garantiza que **no se puede instanciar la clase base** sin implementar el método.

◆ Clases Abstractas

✓ ¿Qué es una clase abstracta?

- Es una clase que **no se puede instanciar directamente**.
- Se usa como modelo o plantilla para otras clases.
- Contiene uno o más **métodos abstractos** que **deben ser sobrescritos** por las subclases.

📌 Características clave:

Característica	Clase Abstracta	Clase Concreta
----------------	-----------------	----------------

¿Se puede instanciar?	<input checked="" type="checkbox"/> No	<input checked="" type="checkbox"/> Sí
¿Contiene métodos sin implementación?	<input checked="" type="checkbox"/> Sí	<input checked="" type="checkbox"/> No
¿Uso principal?	Definir interfaz base	Crear objetos directos
¿Obliga a sobrescribir métodos?	<input checked="" type="checkbox"/> Sí	<input checked="" type="checkbox"/> No

Ejemplo del material (informal):

```
class Cuerpo:

    def superficieBase(self):

        pass # Método sin implementación

    def volumen(self):

        return self.superficieBase() * self.__altura
```

- `Cuerpo` actúa como **clase abstracta informal**.
- Las subclases (`Cilindro`, `ParalelepipedoRectangulo`) **deben implementar `superficieBase()`**.

Aplicación práctica:

```
class Cilindro(Cuerpo):  
    def superficieBase(self):  
        return math.pi * self.__radio ** 2  
  
class ParalelepipedoRectangulo(Cuerpo):  
    def superficieBase(self):  
        return self.__lado1 * self.__lado2
```

Ventajas del uso de clases abstractas:

- Fuerzan la **implementación obligatoria** de métodos esenciales.
- Permiten programar por **interfaz y no por implementación**.
- Facilitan el **uso de polimorfismo**, ya que todas las subclases comparten la misma interfaz.

¿Cuándo usar una clase abstracta?

- Cuando se quiere definir una **estructura común obligatoria** para todas las subclases.

- Cuando se necesita una **implementación parcial** que será completada por subclases.
 - Cuando se quiere **evitar instanciación directa** de clases incompletas.
-



Conclusión

- **Interfaces** en Python se implementan informalmente usando clases base con métodos `pass`, o formalmente con el módulo `abc`.
- Una **clase abstracta** no puede instanciarse y sirve como plantilla: obliga a las subclases a definir ciertos métodos.
- Ambas herramientas son fundamentales para aplicar **polimorfismo, cohesión y diseño orientado a objetos robusto**.
- El archivo de teoría muestra claramente cómo Python permite estructurar programas complejos con estas ideas, incluso sin soporte formal como en Java.