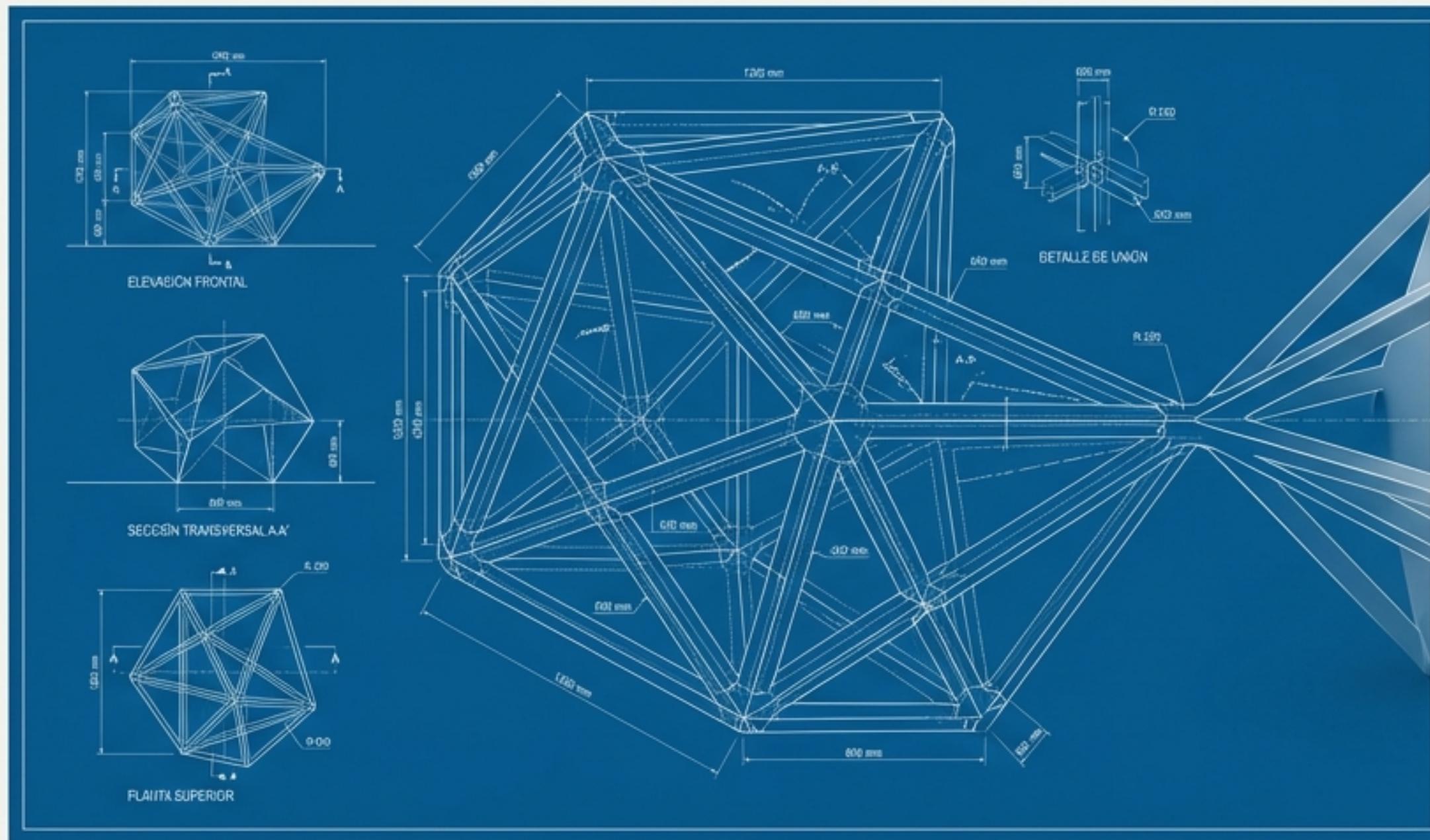


De la Arquitectura al Artefacto

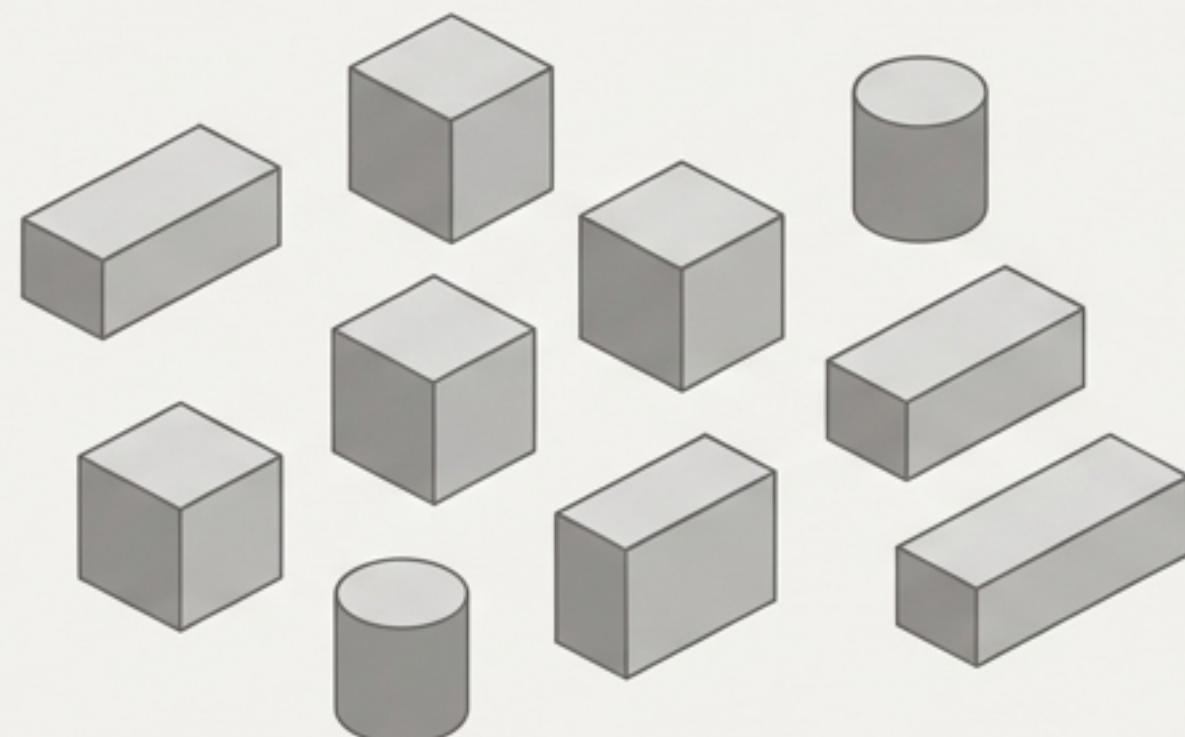
Maestría en Diseño Orientado a Objetos con Python



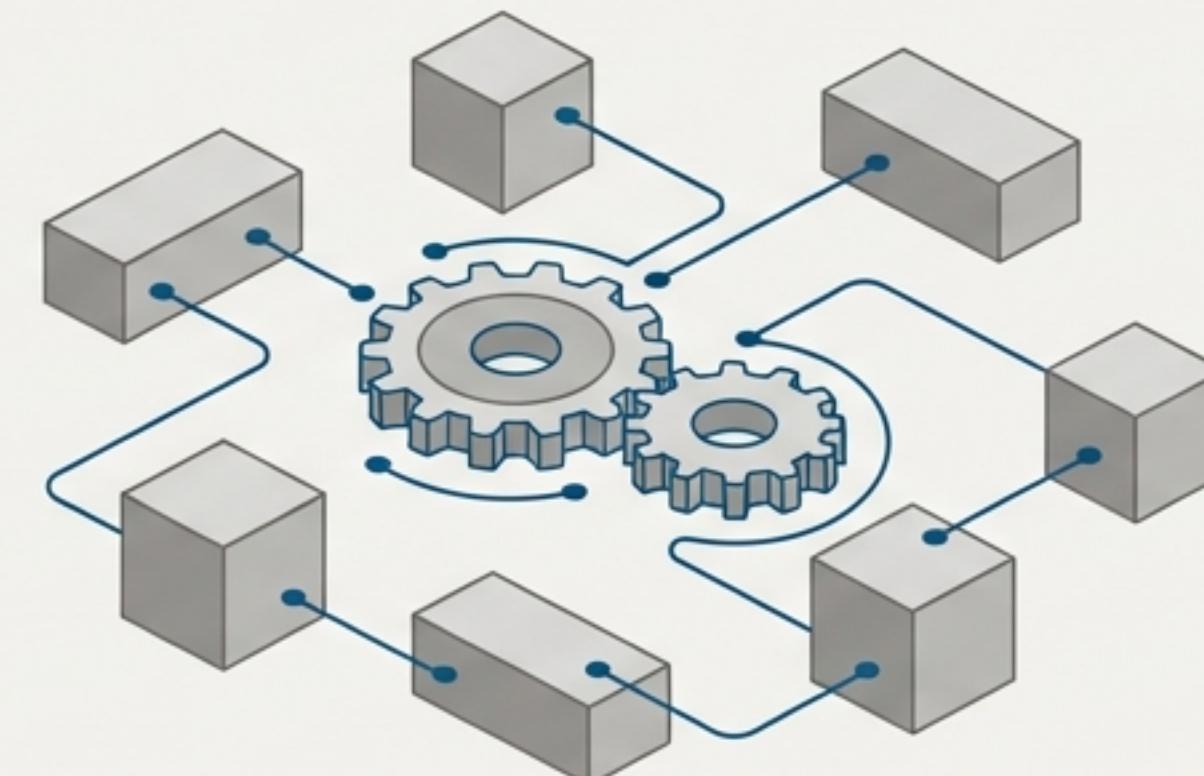
El Desafío No Es Crear Clases, Sino Orquestar Sus Interacciones

La Programación Orientada a Objetos (POO) es la piedra angular del software moderno. En Python, todo es un objeto, pero la verdadera maestría emerge al definir cómo estos objetos se relacionan. Un diseño robusto no se basa en jerarquías rígidas de herencia, sino en un ecosistema flexible de componentes que interactúan.

Aislamiento



Orquestación



Complejidad: Cómo modelar sistemas complejos de forma coherente.



Mantenibilidad: Cómo construir software que evolucione sin romperse.



Flexibilidad: Cómo favorecer patrones que permitan el cambio, como la **Colaboración** y la **Composición**.

El Fundamento Arquitectónico: Una Taxonomía de Relaciones

Para resolver problemas, los objetos deben comunicarse. La naturaleza de esta comunicación define la arquitectura del sistema. En POO, estas relaciones se categorizan por la fuerza de su conexión y la dependencia de su ciclo de vida.



Distinguir entre estos matices es imperativo para tomar decisiones de diseño efectivas, ya que cada una implica diferentes compromisos de memoria, acoplamiento y mantenimiento.

Anatomía de las Relaciones: Un Análisis Comparativo

Tipo de Relación	Semántica	Ciclo de Vida	Acoplamiento	Ejemplo Canónico
Asociación	“Conoce a”	Independiente	Bajo	 Un Estudiante usa un Libro.
Colaboración	“Usa a” / “Coopera con”	Transitorio (método)	Muy Bajo	 Un Conductor usa un GPS para una ruta.
Agregación	“Tiene un” (Débil)	Independiente	Medio	 Un Departamento tiene Profesores.
Composición	“Contiene un” (Fuerte)	Dependiente / Vinculado	Alto	 Una Casa tiene Habitaciones.

La Regla de Oro del Diseño Moderno

“Favorecer la Composición sobre la Herencia”

Herencia (“Es-un”)

- Crea estructuras fuertemente acopladas.
- La subclase depende de la implementación del padre.
- Riesgo del “problema de la clase base frágil”: cambios en el padre pueden romper la funcionalidad del hijo de formas inesperadas.



Herencia

Composición (“Tiene-un”)

- Permite un diseño modular con componentes independientes.
- El comportamiento se ensambla, no se hereda.
- Permite cambiar el comportamiento en tiempo de ejecución intercambiando componentes.



Composición

La Mecánica Interna: La Filosofía Pythonica de Encapsulación

“Todos somos adultos responsables”. Python carece de modificadores de acceso estrictos como `private` o `protected`. En su lugar, se basa en convenciones de nomenclatura.



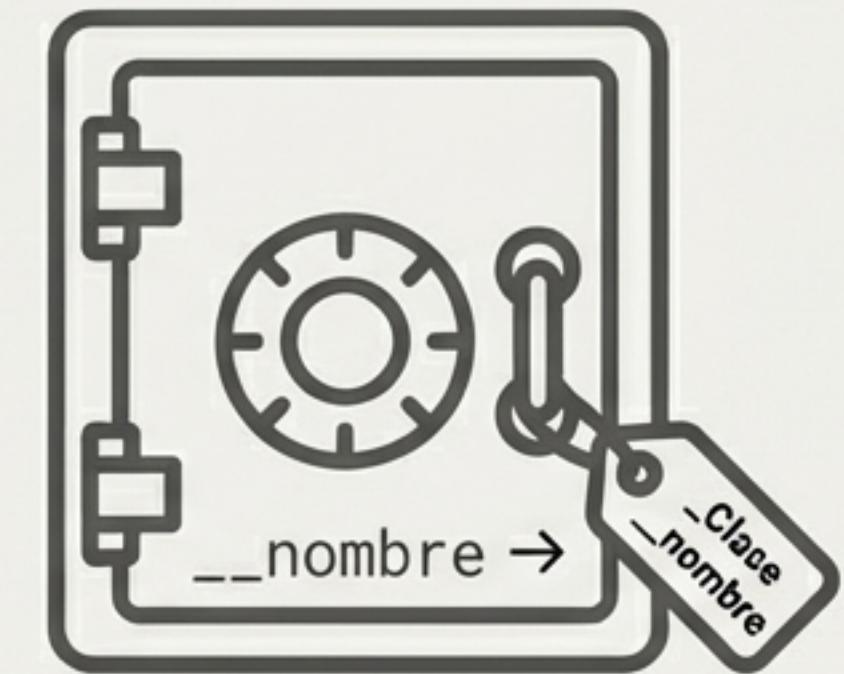
`nombre` (**Público**)

La interfaz estable de la clase.



`_nombre` (**Protegido**)

Destinado a uso interno o de subclases. Acceder desde fuera es mala práctica.



`__nombre` (**Privado**)

Activa el *name mangling* (`_NombreClase__nombre`). Su propósito es evitar conflictos de nombres en herencias complejas, no la seguridad estricta.

De Métodos Verbosos a Propiedades Elegantes

El Problema: El Estilo Java (No Pythonico)

Se prohíbe el acceso directo a atributos, forzando la creación de métodos `getNombre()` y `setNombre()`.

```
# Estilo No Pythonico
class Persona:
    def __init__(self, nombre):
        self._nombre = nombre

    def get_nombre(self):
        return self._nombre

    def set_nombre(self, valor):
        self._nombre = valor
```

La Solución: El Decorador `@property`

Permite empezar con atributos públicos y añadir lógica (validación, cálculo) después, sin romper la API externa.

```
# La Vía Pythonica
class Producto:
    def __init__(self, precio):
        self._precio = 0
        self.precio = precio # Llama al setter

    @property
    def precio(self): # Getter
        return self._precio

    @precio.setter
    def precio(self, valor): # Setter con validación
        if valor < 0:
            raise ValueError("El precio no puede ser
negativo.")
        self._precio = valor
```

Migrar a `@property` cuando se necesite controlar el acceso, validar la entrada o computar un valor dinámicamente.

El Mito de la Sobrecarga: Flexibilidad a la Manera de Python

Concepto Clave:

Python no soporta sobrecarga de métodos nativa. La última definición de una función con el mismo nombre sobrescribe a las anteriores.

Estrategias de Simulación:

- **Argumentos por Defecto (La Vía Estándar):**

La forma más común. Un solo método maneja múltiples casos de uso.

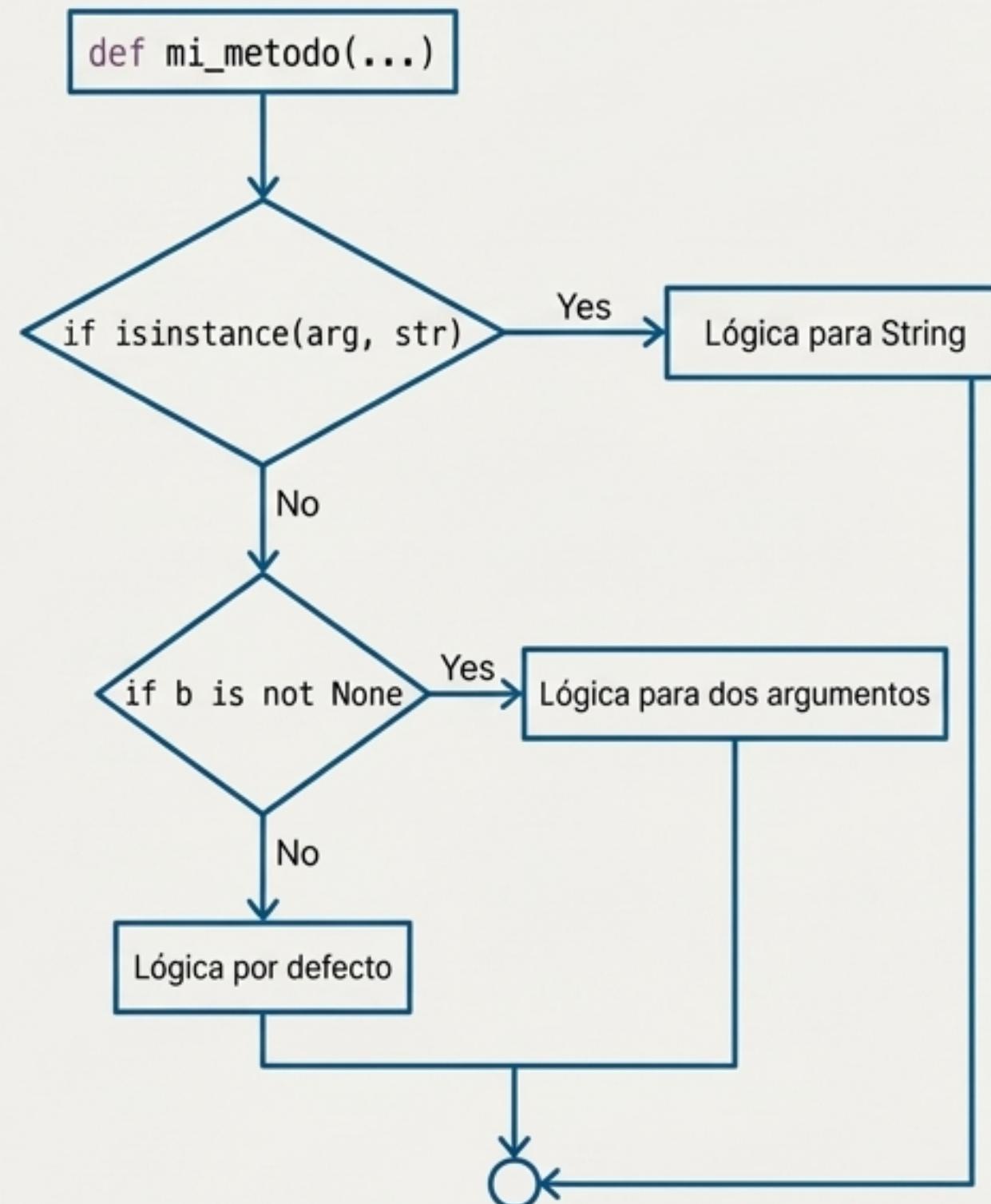
```
def conectar(self, host, puerto=80, timeout=30): ...
```

- **Argumentos Variables (`*args`, `**kwargs`):**

Para un número indeterminado de argumentos posicionales o de palabra clave.

- **Constructores Alternativos (`@classmethod`):**

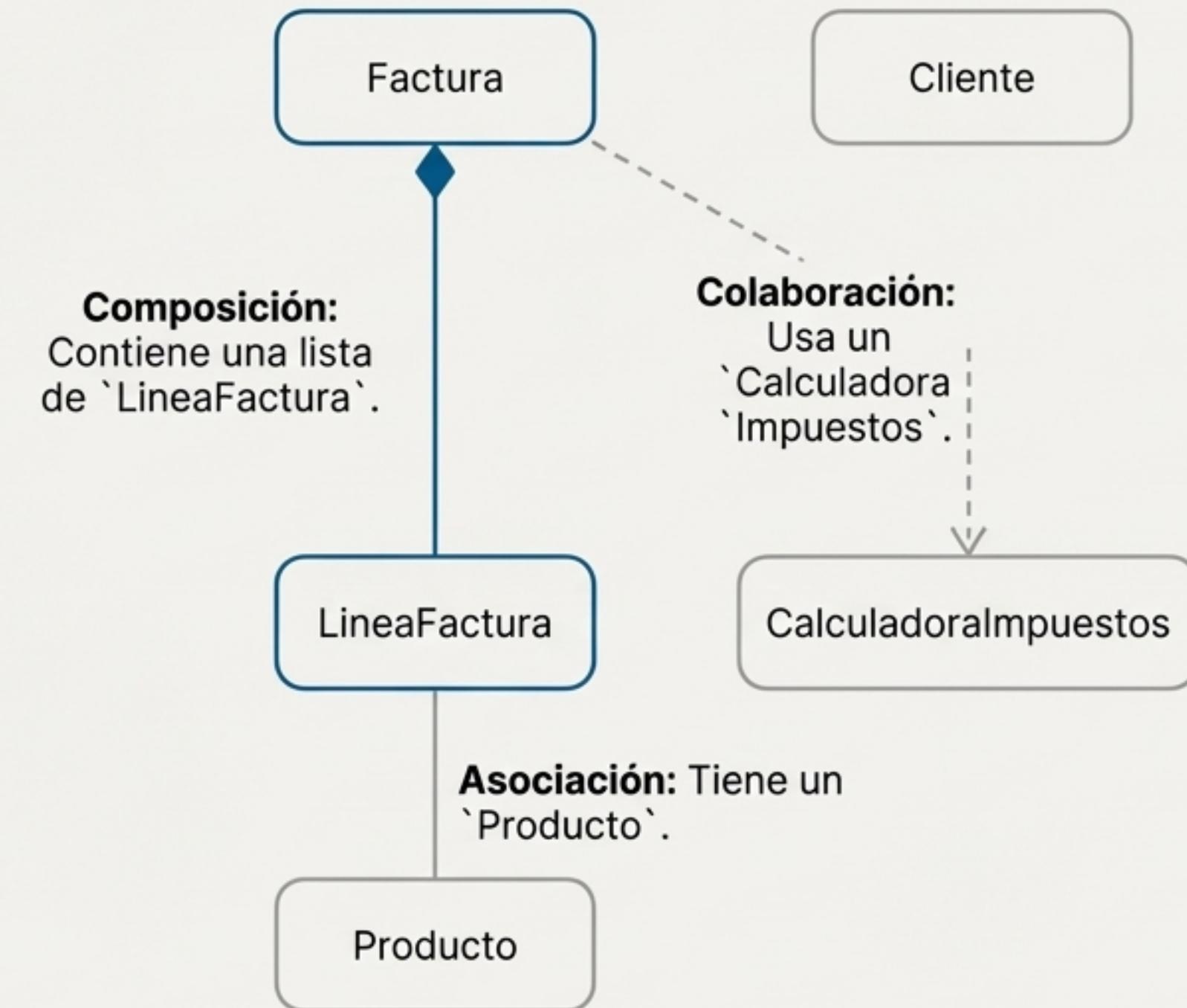
Para crear objetos desde diferentes tipos de entrada (e.g., `from_string`, `from_json`) sin sobrecargar `__init__`.



La Síntesis Práctica: Diseñando un Sistema de Facturación

Análisis del Dominio: Se requiere un programa para generar facturas detalladas. Las entidades clave son:

Callout: Esta arquitectura demuestra una regla de diseño clave: **Componer datos** (Líneas, Productos), pero **Colaborar con lógica** (Servicios, Calculadoras).



Código Anotado: El Colaborador y las Entidades

```
# 1. SERVICIO DE COLABORACIÓN
class CalculadoraImpuestos:
    def __init__(self, tasa: float = 0.21):
        self._tasa = tasa
    def calcular_impuesto(self, monto: float) -> float:
        return monto * self._tasa
    ...

# 2. ENTIDAD DE DOMINIO
class Producto:
    def __init__(self, nombre: str, precio: float):
        self._nombre = nombre
        self.precio = precio # Llama al setter
    @property
    def precio(self) -> float:
        return self._precio
    @precio.setter
    def precio(self, valor: float):
        if valor < 0:
            raise ValueError("El precio no puede ser negativo.")
        self._precio = valor
```

Colaborador: Lógica externa inyectable. Permite cambiar la estrategia fiscal sin modificar la `Factura`. Cumple el Principio Abierto/Cerrado.

Robustez en Construcción: La validación del setter se ejecuta incluso al crear el objeto, previniendo estados inválidos desde el inicio.

Encapsulación Pythonica: El decorador `@property` protege la integridad del estado (`_precio`) con una API limpia y pública.

Código Anotado: Construyendo el Agregador Compuesto

```
class Factura:  
    def __init__(self, cliente: Cliente,  
                 servicio_impuestos: CalculadoraImpuestos):  
  
        self.numero_factura = ...  
        self.cliente = cliente  
  
        # Colaboración: Inyección de Dependencia  
        self._servicio_impuestos = servicio_impuestos ←  
  
        # Composición: La lista de ítems nace aquí  
        self._items: List[LineaFactura] = [] ←  
        ...
```

Decisión de Colaboración: La `Factura` recibe el calculador, no lo crea.

Beneficio: Desacoplamiento total y alta capacidad de prueba (testability), ya que el servicio puede ser reemplazado por un 'mock'.

Decisión de Composición Fuerte: La lista `'_items` es parte integral de la `Factura`. Las `LineaFactura` que contiene viven y mueren con la factura, demostrando un ciclo de vida vinculado.

Código Anotado: Polimorfismo y Estado Computado

Código Fuente (`Factura` - métodos clave)

```
# Simulación de Sobrecarga
def agregar_item(self, producto_o_nombre: Union[Producto, str], ←
    cantidad: int = 1, precio: float = 0.0):
    if isinstance(producto_o_nombre, Producto):
        # Lógica para Firma A
    ...
    elif isinstance(producto_o_nombre, str):
        # Lógica para Firma B
    ...

# Propiedades Computadas
@property
def total_neto(self) -> float: ←
    return sum(item.subtotal for item in self._items)

@property
def monto_impuesto(self) -> float: ←
    # Uso del Colaborador
    return self._servicio_impuestos.calcular_impuesto(self.total_neto)
```

Sobrecarga Simulada: Un único método proporciona una API flexible usando `isinstance` para bifurcar la lógica. Permite agregar productos existentes o crear nuevos 'al vuelo'.

Delegación por Composición: Para calcular el total, la `Factura` itera y delega el cálculo del subtotal a cada uno de sus componentes (`LineaFactura`).

Delegación por Colaboración: Para el impuesto, la `Factura` delega el cálculo al servicio externo, manteniendo su propia lógica simple y enfocada.

El Artefacto Final: Ejecución y Resultado

El Bloque de Ejecución (if __name__ == "__main__")

```
if __name__ == "__main__":
    # 1. Configurar Colaborador
    impuesto_general = CalculadoraImpuestos(tasa=0.19) ←

    # 2. Crear Entidades
    cliente_vip = Cliente(...)
    laptop = Producto(...)

    # 3. Instanciar Factura (inyectando colaborador)
    factura = Factura(cliente_vip, impuesto_general) ←

    # 4. Usar API flexible (sobrecarga simulada)
    factura.agregar_item(laptop, 1)
    factura.agregar_item("Servicio Instalación", 1, 50.00) ←

    # 5. Generar Reporte
    factura.generar_recibo() ←

    # 6. Demostrar Robustez
    try:
        laptop.precio = -100.00 ←
    except ValueError as e:
        print(f"ALERTA DE SEGURIDAD: {e}")
```

La Salida del Programa

```
=====
FACTURA: FAC-20231027-a4b1
Cliente: María García (maria.garcia@email.com)
Fecha: 2023-10-27 10:30
-----
PRODUCTO           CANT   SUBTOTAL
-----
Laptop Gamer      x 1   = $1500.00
Servicio Instalación x 1   = $50.00
-----
Neto:              $ 1550.00
Impuesto (19.0%): $ 294.50
-----
TOTAL A PAGAR:    $ 1844.50
=====
```

ALERTA DE SEGURIDAD CAPTURADA: Error Crítico: El precio de 'Laptop Gamer' no puede ser negativo.

Principios de Diseño para un Software Robusto

"El dominio de estos bloques de construcción fundamentales es el umbral que separa a un programador de scripts de un ingeniero de software en Python."

1. Componer Datos, Colaborar con Lógica

Use Composición para la estructura interna de sus objetos (Factura -> LineaFactura). Use Colaboración para la lógica y servicios externos (Factura -> CalculadoraImpuestos).

2. La Inyección de Dependencias Desacopla y Habilita Pruebas

Al pasar colaboradores en el constructor, **desacoplamos la lógica de negocio** de la fiscal. Esto permite sustituir el colaborador real por un "mock" en pruebas unitarias, un pilar de la ingeniería de software moderna.

3. Las Propiedades Garantizan Estados Válidos

Use @property para **proteger la integridad de sus datos**. Es imposible crear un Producto con **precio negativo, eliminando una clase entera de bugs**.

4. Los Patrones Pythonicos Simplifican el Diseño

Características como los **módulos (singletons naturales)**, los **métodos de clase (fábricas)** y las **funciones** como ciudadanos de primera clase a menudo ofrecen soluciones más simples y elegantes que los patrones de diseño tradicionales del "Gang of Four".