Proyecto final: PyShop, una tienda en línea con POO

En este proyecto construyes desde cero el sistema de PyShop: productos, descuentos, carrito, catálogo y errores propios. El objetivo no es solo que los totales salgan bien, sino que el diseño aplique lo visto en el curso: encapsulación, herencia, polimorfismo, abstracción, composición y excepciones. Usa solo la biblioteca estándar de Python 3.11 o posterior. El notebook de referencia adjunto contiene una solución completa: consúltalo al terminar, no antes.

Requisitos mínimos

1. **Excepciones propias.** Una raíz `ErrorTienda(Exception)` y tres subclases: `PrecioInvalidoError`, que además herede de `ValueError`; `StockInsuficienteError`, con los atributos `producto`, `disponible`, `solicitado` y `faltante`; y `ProductoNoEncontradoError`, que además herede de `KeyError`.
2. **Producto abstracto con dos subclases.** `Producto(ABC)` con nombre, precio, stock y un SKU automático que empieza en 100, más `reponer`, `retirar`, `subtotal`, `__repr__` y `__str__`. El método `costo_envio(cantidad)` es abstracto. `ProductoFisico` tiene `peso_kg` y cobra 49 + 15 × peso × cantidad; `ProductoDigital` no se agota y su envío es 0.
3. **Property con validación.** `precio` es una `@property` cuyo setter lanza `PrecioInvalidoError` si el valor es menor o igual a cero, también al construir el objeto. `stock` es de solo lectura.
4. **Descuento abstracto con dos subclases.** `Descuento(ABC)` con `aplicar(subtotal)`, que devuelve el monto a descontar. `DescuentoPorcentaje(p)` y `DescuentoFijo(monto, minimo=0)`, que solo aplica si el subtotal alcanza el mínimo.
5. **Carrito con composición y confirmación atómica.** `Linea` es un `@dataclass(frozen=True)` con producto y cantidad. `Carrito(cliente)` guarda una lista de líneas y un descuento opcional, y ofrece `agregar`, `subtotal`, `envio`, `total`, `__len__` (número de piezas) y `__iter__`. `confirmar()` valida todo el pedido antes de descontar stock y devuelve el total; si algo no alcanza, lanza `StockInsuficienteError` sin modificar ningún producto.
6. **Tienda con búsqueda.** `Tienda(nombre)` guarda el catálogo en un diccionario SKU → producto, con `registrar(producto)` y `buscar(sku)`, que lanza `ProductoNoEncontradoError` con `raise ... from None`.

Datos de prueba

```
tienda = Tienda("PyShop")
taza = tienda.registrar(ProductoFisico(
    "Taza", 149.0, stock=20, peso_kg=0.4))
teclado = tienda.registrar(ProductoFisico(
    "Teclado", 899.0, stock=3, peso_kg=1.2))
ebook = tienda.registrar(
    ProductoDigital("Ebook de Python", 299.0))

car = Carrito("Ana")
car.agregar(taza, 2)
car.agregar(teclado, 1)
car.agregar(ebook, 1)
```

Casos de prueba esperados

Tu solución debe reproducir exactamente estas cifras. Escríbelas como `assert`:

- `str(taza)` → `Taza — $149.00 MXN`; los SKU son 100, 101 y 102.
- Envío: taza × 2 = 61.0, teclado × 1 = 67.0, ebook = 0.0.
- Carrito de Ana: 4 piezas, subtotal 1496.0, envío 128.0 y **total 1624.0**.
- Con `DescuentoPorcentaje(10)`, el total es **1474.4**.
- Con `DescuentoFijo(200, minimo=1500)`, el total sigue en 1624.0: el subtotal 1496 no alcanza el mínimo.
- `confirmar()` (sin descuento) devuelve 1624.0 y deja el stock en 18 tazas y 2 teclados.
- Después, un carrito de Luis con 5 teclados lanza `StockInsuficienteError` con el mensaje `Teclado: hay 2 y se pidieron 5` y **faltante 3**.
- `taza.precio = -10` se puede atrapar con `except ValueError` y el precio sigue en 149.0.
- `tienda.buscar(999)` se puede atrapar con `except ErrorTienda`.
- `Producto("X", 10)` lanza `TypeError` y `taza.stock = 1000` lanza `AttributeError`.

Agrega una prueba de atomicidad. Con los productos recién creados (20 tazas y 3 teclados), un carrito con 2 tazas y dos líneas de 2 teclados cada una debe fallar al confirmar y dejar el stock en 20 y 3.

```
def lanza(tipo_error, funcion):
    try:
        funcion()
    except tipo_error:
        return True
```

```
    return False

eva = Carrito("Eva")
eva.agregar(taza, 2)
eva.agregar(teclado, 2)
eva.agregar(teclado, 2)
assert lanza(StockInsuficienteError, eva.confirmar)
assert (taza.stock, teclado.stock) == (20, 3)
```

Extras opcionales

- Manejar el dinero con `decimal.Decimal` o con centavos enteros.
- `DescuentoPorPiezas` u otro descuento nuevo sin modificar `Carrito`.
- Un tercer tipo de producto, por ejemplo perecedero con fecha de caducidad.
- Historial de pedidos confirmados y ventas totales en `Tienda`.
- Notas con `add_note` en los errores de stock.
- Pruebas con `unittest` o `pytest`.

Entregables

1. El código: un módulo `pyshop.py` o un notebook que corra de principio a fin sin errores.
2. Las pruebas: todos los casos esperados como `assert`, incluida la prueba de atomicidad.
3. Una demostración: un pedido de punta a punta que imprima las líneas del carrito, el subtotal, el envío y el total.
4. Una nota breve (media página) que explique dos decisiones de diseño: por qué una clase es abstracta y por qué el carrito usa composición en lugar de herencia.

Rúbrica (100 puntos)

- **Excepciones (15 puntos):** jerarquía con raíz común, herencia de `ValueError` y `KeyError`, atributo `faltante`.
- **Productos y encapsulación (20 puntos):** clase abstracta, `super().__init__`, setter que valida, stock de solo lectura, `__repr__` y `__str__`.
- **Polimorfismo y descuentos (15 puntos):** envío y descuentos sin cadenas de `if isinstance`.
- **Carrito y tienda (25 puntos):** composición, totales correctos, `confirmar` atómico y búsqueda con su error propio.
- **Pruebas (15 puntos):** todos los casos esperados pasan, incluidos los de error.
- **Claridad (10 puntos):** nombres claros, métodos cortos y la nota de diseño.

Los extras no suman puntos, pero son la mejor práctica para consolidar lo aprendido.