Proyecto final: aplicación de pedidos e inventario de PyShop

Durante el curso consultaste la base de PyShop, cambiaste datos con transacciones, manejaste errores y aceleraste consultas con índices. Ahora vas a juntar todo en una aplicación pequeña pero completa: un módulo de acceso a datos y un programa de consola para registrar clientes, productos y pedidos, controlar el inventario y generar reportes. Se usan Python 3.12 o superior y el módulo estándar `sqlite3`; pandas y SQLAlchemy solo para los extras. El notebook `proyecto-app-pedidos` contiene una solución de referencia: consúltalo al terminar, no antes.

Contexto

El equipo de PyShop registra los pedidos a mano y descubre tarde que vendió productos sin existencias. Quiere una herramienta que nunca deje un pedido a medias, que avise con claridad cuando algo no se puede hacer y que responda en segundos las preguntas de siempre: cuánto se vende por categoría, ciudad y mes, qué productos se agotan y qué clientes no han comprado.

Datos

Usa la función `crear_bd` de la primera celda de cualquier notebook del curso, con **otra semilla** y en un archivo:

```
con = crear_bd("pyshop.db", semilla=2026)
for t in ("clientes", "productos",
          "pedidos", "detalle"):
    sql = f"SELECT COUNT(*) FROM {t}"
    print(t, con.execute(sql).fetchone()[0])
con.close()
# clientes 300, productos 40,
# pedidos 2000, detalle 3489
```

Si tu tabla `detalle` no tiene 3489 filas, revisa la semilla. Todas las demás cifras debes obtenerlas tú.

Requisitos mínimos

1. **Módulo `pyshop_db.py`.** Una función `conectar(ruta)` que abra la base con `PRAGMA foreign_keys = ON` y `sqlite3.Row`, y una excepción propia `ErrorTienda` (con al menos una subclase, como `StockInsuficiente`).

2. **Altas.** `registrar_cliente` y `registrar_producto`, con parámetros y transacción. Un correo o nombre repetido, un precio no positivo o un dato faltante deben terminar en `ErrorTienda` con un mensaje claro.
3. **Pedidos.** `registrar_pedido(con, cliente_id, items)`, con `items` como lista de `(producto_id, cantidad)`: inserta el pedido y su detalle con el precio del momento y descuenta el stock, todo en **una** transacción. Si un artículo no alcanza, el producto o el cliente no existe, o la lista está vacía, la base no debe cambiar.
4. **Reportes**, sin contar los pedidos cancelados en el ingreso:
    - R1. Ingreso y número de pedidos por categoría.
    - R2. Ingreso por ciudad.
    - R3. Ingreso y pedidos por mes.
    - R4. Los cinco productos con más unidades vendidas.
    - R5. Clientes sin pedidos (con `LEFT JOIN`).
    - R6. Productos con stock menor que un límite que recibe la función.
5. **Índices.** Crea al menos dos y justifica cada uno con la salida de `EXPLAIN QUERY PLAN` antes (`SCAN`) y después (`SEARCH`). Explica por qué no indexaste alguna otra columna.
6. **Programa de consola** `app.py` con `argparse` y subcomandos: `reporte`, `pedido` y `devolver` (o un menú con `input`, si lo prefieres). Ante un `ErrorTienda` muestra el mensaje y termina con código 1, sin rastreo.
7. **Pruebas** en `test_pyshop_db.py`: al menos seis funciones con `assert` sobre bases `:memory:`, incluidas un pedido válido que descuenta stock, un pedido imposible que no deja rastro (pedidos, detalle y stock iguales) y un correo repetido.

Pistas de implementación

Con `RETURNING` obtienes el id nuevo en la misma sentencia:

```
sql = ("INSERT INTO pedidos (cliente_id, fecha, estado) "
       "VALUES (?, ?, 'pagado') RETURNING id")
pid = con.execute(sql, (cliente, fecha)).fetchone()[0]
```

Valida el stock antes de escribir para dar un buen mensaje, pero deja que el `CHECK (stock >= 0)` siga como red de seguridad. Traduce solo los errores que entiendes:

```
try:
    with con:
        ...
except sqlite3.IntegrityError as e:
    if "FOREIGN KEY" in str(e):
        msg = "No existe el cliente."
```

```
            raise ErrorTienda(msg) from e
        raise
```

Para los pedidos de un mes usa un rango sobre `fecha` (`fecha >= ? AND fecha < ?`) y no `substr`, o el índice no servirá. Para un subcomando como `pedido 5 --item 3:1 --item 7:2`, `add_argument("--item", action="append")` junta todos los artículos en una lista.

Los datos son sintéticos: si un mes o una ciudad vende más, descríbelo, pero no le inventes una causa.

Extras opcionales

- **Devoluciones.** `devolver_pedido(con, pedido_id)` marca el pedido como cancelado y regresa al stock sus cantidades, todo o nada; rechaza un pedido inexistente o ya cancelado.
- **Exportar a CSV.** Con `pd.read_sql_query` y `to_csv(index=False)`, guarda R1, R2 y R3 en archivos.
- **SQLAlchemy.** Repite un reporte con `create_engine("sqlite:///pyshop.db")` y `text()` con parámetros `:nombre`.
- **Escritura en lote.** Importa 100 clientes nuevos con `executemany` en una transacción y mide contra un `commit` por fila.

Entregables

1. **`pyshop_db.py`**: conexión, excepciones, altas, pedidos, reportes e índices.
2. **`app.py`**: el programa de consola.
3. **`test_pyshop_db.py`**: las pruebas, que deben pasar con `python -m pytest` o ejecutando el archivo.
4. **`INFORME.md`** (una página): ejemplos de uso de la consola con su salida, los planes antes y después de cada índice y tres conclusiones con cifras de los reportes.

Rúbrica (100 puntos, más 10 de extras)

- **Conexión y estructura (10 puntos):** `conectar` único, SQL separado de la lógica, conexiones cerradas.
- **Altas y errores (15 puntos):** parámetros siempre, excepciones específicas y mensajes útiles; ningún `except` vacío.
- **Pedidos con transacción (25 puntos):** todo o nada comprobado, stock correcto, precio del momento.
- **Reportes (20 puntos):** seis consultas correctas, sin cancelados en el ingreso.
- **Índices (10 puntos):** al menos dos, justificados con el plan.
- **Consola (10 puntos):** subcomandos claros y errores sin rastreo.

- **Pruebas (10 puntos):** seis o más, aisladas en `:memory:`, incluido el caso imposible.