UNIVERSIDAD TECNOLÓGICA DEL URUGUAY
ITR SUROESTE · FRAY BENTOS
INGENIERÍA MECATRÓNICA
UC de Programación IV · 2022/2
Profesor: Giovani Bolzan Cogo

Práctico de laboratorio II - Programación Orientada a Objetos	Periodo: de 19/09/2025 hasta 03/10/2025

Integrantes

Felipe Morrudo – 5.582.493-4
Hector Pereira – 5.582.582-5
Santiago Moizo – 5.165.430-5
Aldrin Pacheco –

1. Objetivos

Este práctico tiene como objetivo general el modelado Orientado a Objetos (O.O.) de un sistema hipotético con el fin de simular la representación ficticia de un sistema de gestión de una empresa en cuanto a su estructura de ventas, aplicándose los pasos de análisis de estructuras de datos y modelado en diagrama UML (Unified Modeling Language).

De los objetivos puntuales, destácanse los siguientes.

- Análisis de una estructura de datos y sus interrelaciones en UML;
- Implementación de un modelado de datos O.O. en código-fuente;
- Utilización de las estructuras básicas del lenguaje de programación Python v3;
- Adaptación de los conceptos de datos O.O. para un lenguaje de programación a través de abstracción, herencia y encapsulamiento.

2. Fundamentación teórica

El desarrollo del código es en Python, un lenguaje con sintaxis clara y modelo de objetos uniforme, que ofrece tipado dinámico con anotaciones opcionales para documentar expectativas. Este marco permite expresar reglas de negocio de forma directa y apoyarse en una biblioteca estándar amplia para tareas comunes (archivos, formatos de datos, argumentos de ejecución) sin agregar complejidad innecesaria.

2.1. Fechas y tiempos

dateutil.parser aporta un analizador flexible de fechas capaz de reconocer múltiples formatos habituales sin configuraciones extensas.

dateparser complementa lo anterior al interpretar expresiones en lenguaje natural (“hoy”, “mañana”, “15/03/25”), reduciendo errores de carga y normalizando la información temporal en objetos datetime comparables y ordenables.

2.2. Datos en archivos de texto

Con ***json*** guardamos y leemos información (listas, diccionarios, números, textos) en un formato estándar. Eso hace sencillo guardar el estado del sistema y volver a cargarlo después, o intercambiar datos con otras herramientas.

2.3. Rutas y archivos

pathlib.Path permite trabajar con rutas como objetos (no como simples cadenas). Así podemos preguntar si un archivo existe, crear carpetas o leer y escribir texto con métodos directos, y el código funciona igual en distintos sistemas operativos.

2.4. Ejecución del programa

sys da acceso a cosas del entorno de ejecución. Por ejemplo, ***sys.argv*** trae los argumentos con los que se abrió el programa. Esto permite cambiar el comportamiento según esos parámetros, sin tocar la lógica principal.

2.5. Aclaración de datos en función

Aunque Python no obliga a declarar tipos, podemos aclarar qué forma tienen los datos que recibe o devuelve una función. Con ***typing.Iterable***, por ejemplo, decimos "esto acepta algo que se pueda recorrer" (lista, tupla, etc.). Estas aclaraciones ayudan a entender el código y a que el editor avise antes si pasamos algo que no corresponde.

Python y las librerías empleadas nos dan brindan herramientas para: entender fechas variadas, guardar y cargar datos de manera simple, manejar rutas y archivos sin errores, ajustar la ejecución con argumentos, y escribir código más claro indicando qué datos esperamos en cada parte .

3. Metodología

3.1. Herramientas y material

Utilizando *Draw.io* se replico el diagrama UML base presentado en el documento de la práctica, con sus correspondientes clases iniciales: Fecha, Producto, Compra, Persona, Vendedor y Cliente. Junto con sos atributos y metodos respectivos a cada clase.

En la implementación del código, utilizamos el lenguaje Phyton en su versión 3.10, ya que se utilizo la plataforma *Google Collab*, esta plataforma nos permite la edición en conjunto del código, ya que se guarda en la nube y varias personas en simultaneo pueden acceder a este.

3.2. Procedimiento

1. **Diagrama UML.** Se analizo el diagrama UML inicial proporcionado en la consigna de la práctica, con el fin de pensar en posibles modificaciones a implementar para facilitar la gestion del sistema. A continuación, en la Fig.1 se observa el diagrama UML inicial.

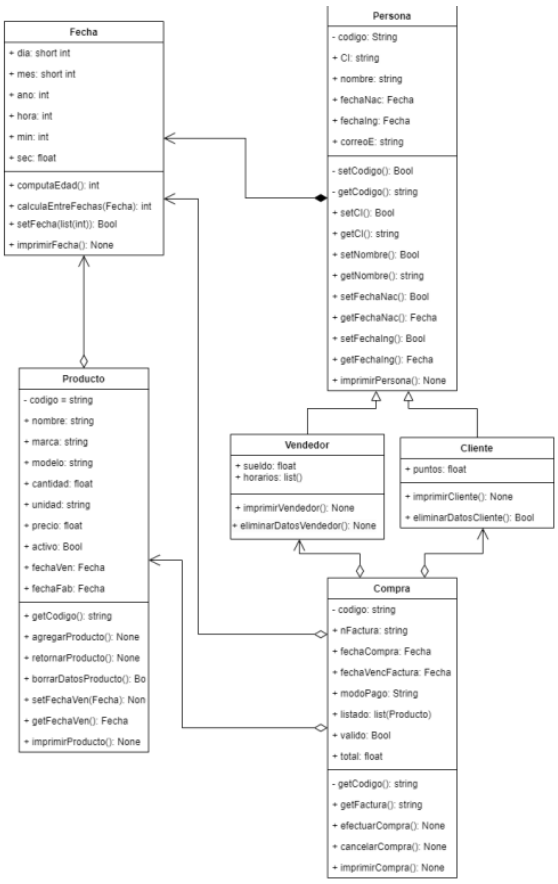

Figura 1: Diagrama UML inicial

Para comodidad y agilización del sistema, se decidió implementar una nueva clase, dicha clase se llama *itemCompra* y su función es encapsular todo lo necesario para poder calcular el subtotal de la linea. La línea refiere a una linea de la factura de una compra, o sea, se guarda el producto seleccionado junto con su cantidad y su precio unitario, posteriormente multiplica el precio unitario por la cantidad para calcular el monto total de ese producto. La clase *Compra* haya el total de la compra sumando todos los *itemCompra*

A su vez, para implementar toda la lógica de menú, e interfaz de usurario, se decide implementar una clase 'Menu' la cual se encargará del manejo de operaciones dentro de la aplicación como tal, recolección de datos del usuario para la creación, modificación y eliminación del resto de clases del sistema.

2. **Código en Phyton.** Después de finalizar el diagramar UML, se procedió a implementar el código en Phyton, utilizando los conceptos de -P.O.O-. Se incorporaron los atributos, métodos, getters y setters necesarios para el correcto funcionamiento de las clases dentro del sistema, ademas de añadir librerías para simplificar el manejo de estas clases. Dentro del código, en la clase 'Menu' se implementó la lógica de navegación para que el usuario pueda interactuar con el sistema. Dentro de este menú el usuario dispone de las siguientes opciones:

   - Consultar información
   - Crear información
   - Borrar información
   - Menu compras
   - Salir

3. **Almacenamiento de datos.** El almacenamiento se maneja con listas en memoria que se sincronizan con cuatro archivos de texto: *dbProductos.txt*, *dbClientes.txt*, *dbVendedores.txt* y *dbCompras.txt*. Al iniciar, el programa lee cada archivo: si no existe o está vacío, arranca con una

lista vacía; si el archivo tiene un arreglo JSON completo, lo carga; y si no, lee una línea por registro y arma las listas de objetos. Durante la ejecución se trabaja solo con esas listas. Al guardar, la función 'save_all' escribe primero en un archivo temporal y luego lo reemplaza por el definitivo, para evitar que quede algo a medio escribir. Las compras se guardan solo con el minimo necesario: se registra el código del cliente, del vendedor y de cada producto, más la cantidad; al volver a cargar, se buscan esos códigos y, si alguno no existe, se avisa para mantener los datos consistentes.

4. **Verificación y validación**.

   Para asegurar el correcto funcionamiento del sistema, se realizaron dos pruebas. Primero, la verificación por partes: se crearon fechas (f1. . . f5) y se comprobó que se construyen sin error y que pueden usarse en clientes, vendedores y compras. Con los clientes (c1, c2) y vendedores (v1, v2), se revisó que los datos cargados por constructor queden accesibles por sus getters (códigos, nombres, puntos, sueldo y horarios). Con los productos (p1, p2, p3) se validó precio y stock inicial. Luego, con los ítems de compra (i1, i2, i3) se verificó el cálculo de subtotal (getPrecio() x cantidad). Finalmente, con las compras (compra1, compra2) se comprobó que el total sea la suma de los subtotales de sus ItemCompra, que acepten la lista de ítems y que conserven las referencias correctas a cliente y vendedor.

   En la segunda etapa se hizo la validación del sistema completo. Se definio una celda de testeo con dos clientes, dos vendedores, tres productos y dos compras, y se ejecutó el guardado con save_all("./db", listaClientes, listaVendedores, listaProductos, listaCompras). Para la correcta validación se tenian que cumplir los siguientes requisitos:

   - Los archivos dbClientes.txt, dbVendedores.txt, dbProductos.txt y dbCompras.txt se generan sin errores
   - Las compras quedan almacenadas por referencias (códigos y cantidades) y, al recargar, cada código se resuelve a un objeto existente
   - Los totales calculados antes de guardar coinciden con los totales recalculados después de volver a cargar
   - Si se fuerza un código inexistente en una compra, el sistema avisa al reconstruir (evita datos rotos).

   Con estas pruebas se confirmó que las clases funcionan de manera aislada y que, integradas y persistidas, mantienen coherencia con los requisitos del práctico.

4. Resultados

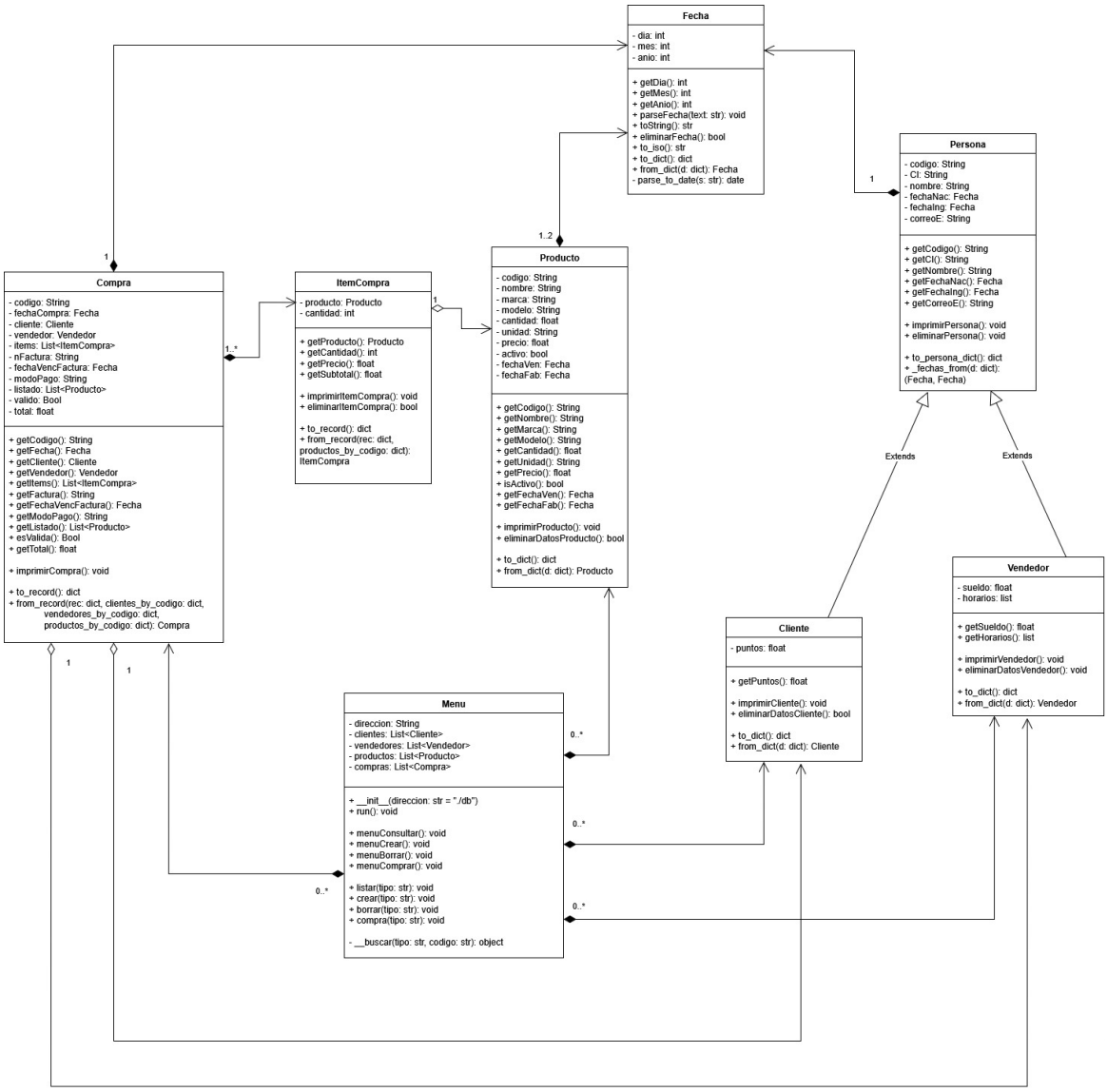

Figura 2: Diagrama UML final. Fuente: Elaboración própia

5. Conclusiones

Descripción de las conclusiones e de lo que se puede y no se puede inferir de los resultados obtenidos presentados en el Apartado 4.

Opcional: Descripción de alternativas y/o mejoras a ponderarse para futuras aplicaciones de estos métodos.

Referencias

[Dow12] A. Downey. *Think Python: How to Think Like a Computer Scientist*. Green Tea Press, 2012.

[Mor00] F. Morero. *Introducción a la OOP*. Grupo EIDOS, Montevideo, 2000.

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[UML] UML-diagrams.org. Uml class and object diagrams overview.

6. Apendices

A. CÓDIGO-FUENTE EN LENGUAJE PYTHON

```
from dateutil import parser
import dateparser
import sys
import json
from pathlib import Path
from typing import Iterable

# Guardado en base de datos


_DB_FILES = {
    "productos":  "dbProductos.txt",
    "vendedores": "dbVendedores.txt",
    "clientes":   "dbClientes.txt",
    "compras":    "dbCompras.txt",
}

def _read_jsonl(path: Path) -> list[dict]:
    '''Lee un archivo JSONL o un JSON array y devuelve una lista de dicts.

    Soporta dos formatos:
      1) **JSON Lines (JSONL)**: un objeto JSON por linea.
      2) **JSON Array**: el archivo completo contiene un array JSON.

    Si el archivo no existe o esta vacio, devuelve `[]`.

    Args:
        path (Path): Ruta del archivo a leer.

    Returns:
        list[dict]: Lista de registros decodificados.

    Raises:
        json.JSONDecodeError: Si el contenido no es JSON valido.

    Ejemplo:
        >>> filas = _read_jsonl(Path("./db/dbClientes.txt"))
        >>> isinstance(filas, list)
        True
    '''
    if not path.exists():
        return []
    txt = path.read_text(encoding="utf-8").strip()
    if not txt:
        return []
    if txt.lstrip().startswith("["):
        return json.loads(txt)
    out = []
    for line in txt.splitlines():
        line = line.strip()
        if line:
            out.append(json.loads(line))
    return out

def _write_jsonl_atomic(path: Path, rows: Iterable[dict]) -> None:
    '''Escribe registros como JSONL de forma atomica y con UTF-8.
```

```
    Crea el directorio si no existe. Primero escribe a un archivo temporal
    (‘.tmp‘) y luego reemplaza el archivo destino, evitando archivos corruptos
    ante fallos a mitad de escritura.

    Args:
        path (Path): Ruta del archivo destino.
        rows (Iterable[dict]): Registros a serializar (uno por linea).

    Returns:
        None

    Raises:
        OSError: Si hay fallos de E/S al escribir o renombrar.
        TypeError: Si algun elemento de ‘rows‘ no es serializable a JSON.

    Ejemplo:
        >>> _write_jsonl_atomic(Path("./db/dbProductos.txt"),
        ...                     (p.to_dict() for p in listaProductos))
    '''
    path.parent.mkdir(parents=True, exist_ok=True)
    tmp = path.with_suffix(path.suffix + ".tmp")
    with tmp.open("w", encoding="utf-8") as f:
        for r in rows:
            f.write(json.dumps(r, ensure_ascii=False, sort_keys=True) + "\n")
    tmp.replace(path)

def save_all(
    base_dir: str | Path,
    listaClientes,
    listaVendedores,
    listaProductos,
    listaCompras
):
    '''Persiste todas las listas de objetos del sistema en archivos TXT (JSONL).

    Escribe:
      - clientes   -> dbClientes.txt
      - vendedores -> dbVendedores.txt
      - productos  -> dbProductos.txt
      - compras    -> dbCompras.txt

    Para clientes, vendedores y productos se usa ‘obj.to_dict()‘.
    Para compras se usa ‘compra.to_record()‘ (formato relacional por codigos).

    Args:
        base_dir (str | Path): Carpeta base que contiene los TXT.
        listaClientes: Lista de instancias ‘Cliente‘.
        listaVendedores: Lista de instancias ‘Vendedor‘.
        listaProductos: Lista de instancias ‘Producto‘.
        listaCompras: Lista de instancias ‘Compra‘.

    Returns:
        None

    Side effects:
        Crea/actualiza archivos dentro de ‘base_dir‘.

    Ejemplo:
        >>> save_all("./db", listaClientes, listaVendedores, listaProductos, listaCompras)
    '''
```

```
    base = Path(base_dir)
    _write_jsonl_atomic(base / _DB_FILES["clientes"],   (c.to_dict() for c in listaClientes)
    )
    _write_jsonl_atomic(base / _DB_FILES["vendedores"], (v.to_dict() for v in
    listaVendedores))
    _write_jsonl_atomic(base / _DB_FILES["productos"],  (p.to_dict() for p in listaProductos
    ))
    _write_jsonl_atomic(base / _DB_FILES["compras"],    (c.to_record() for c in listaCompras
    ))

def load_clientes(path: Path) -> list[Cliente]:
    '''Carga clientes desde un TXT (JSONL/JSON) y devuelve instancias `Cliente`.

    Usa `Cliente.from_dict(d)` para reconstruir cada objeto. Las lineas que
    no puedan decodificarse o mapearse se informan y se omiten.

    Args:
        path (Path): Ruta al archivo de clientes (p.ej. ./db/dbClientes.txt).

    Returns:
        list[Cliente]: Lista de clientes validos.

    Ejemplo:
        >>> clientes = load_clientes(Path("./db/dbClientes.txt"))
        >>> len(clientes) >= 0
        True
    '''
    clientes = []
    for d in _read_jsonl(path):
        try:
            clientes.append(Cliente.from_dict(d))
        except Exception as e:
            print(f"Warning: could not load cliente {d}: {e}")
    return clientes

def load_vendedores(path: Path) -> list[Vendedor]:
    '''Carga vendedores desde un TXT (JSONL/JSON) y devuelve instancias `Vendedor`.

    Usa `Vendedor.from_dict(d)` para reconstruir cada objeto. Registros con
    errores se informan y omiten.

    Args:
        path (Path): Ruta al archivo de vendedores (p.ej. ./db/dbVendedores.txt).

    Returns:
        list[Vendedor]: Lista de vendedores validos.

    Ejemplo:
        >>> vendedores = load_vendedores(Path("./db/dbVendedores.txt"))
        >>> all(hasattr(v, "getSueldo") for v in vendedores)
        True
    '''
    vendedores = []
    for d in _read_jsonl(path):
        try:
            vendedores.append(Vendedor.from_dict(d))
        except Exception as e:
            print(f"Warning: could not load vendedor {d}: {e}")
    return vendedores

def load_productos(path: Path) -> list[Producto]:
```

```
    '''Carga productos desde un TXT (JSONL/JSON) y devuelve instancias `Producto`.

    Usa `Producto.from_dict(d)` para reconstruir cada objeto. Registros con
    errores se informan y omiten.

    Args:
        path (Path): Ruta al archivo de productos (p.ej. ./db/dbProductos.txt).

    Returns:
        list[Producto]: Lista de productos validos.

    Ejemplo:
        >>> productos = load_productos(Path("./db/dbProductos.txt"))
        >>> any(p.getPrecio() > 0 for p in productos)
        True
    '''
    productos = []
    for d in _read_jsonl(path):
        try:
            productos.append(Producto.from_dict(d))
        except Exception as e:
            print(f"Warning: could not load producto {d}: {e}")
    return productos

def load_compras(path: Path, clientes_by_codigo, vendedores_by_codigo, productos_by_codigo):
    '''Carga compras desde TXT (JSONL) y devuelve instancias `Compra`.

    Reconstruye cada compra a partir de un registro relacional:
      - fecha en ISO (yyyy-mm-dd) -> `Fecha(d, m, y)`
      - `cliente_codigo` y `vendedor_codigo` se resuelven con los *indices*
        `clientes_by_codigo` y `vendedores_by_codigo`.
      - `items` se reconstruyen con `ItemCompra(producto, cantidad)`, donde
        el `producto` se busca en `productos_by_codigo`.
      - Campos opcionales de factura: `nFactura`, `modoPago`,
        `fechaVencFactura` (ISO), `valido`, `total`, `listado`.

    Los registros con referencias inexistentes (cliente/vendedor/producto)
    se informan y se omiten parcialmente o totalmente, segun el caso.

    Args:
        path (Path): Ruta al archivo de compras (p.ej. ./db/dbCompras.txt).
        clientes_by_codigo (dict[str, Cliente]): indice por codigo de cliente.
        vendedores_by_codigo (dict[str, Vendedor]): indice por codigo de vendedor.
        productos_by_codigo (dict[str, Producto]): indice por codigo de producto.

    Returns:
        list[Compra]: Lista de compras reconstruidas.

    Ejemplo:
        >>> clientes = load_clientes(Path("./db/dbClientes.txt"))
        >>> vendedores = load_vendedores(Path("./db/dbVendedores.txt"))
        >>> productos = load_productos(Path("./db/dbProductos.txt"))
        >>> compras = load_compras(Path("./db/dbCompras.txt"),
        ...                        {c.getCodigo(): c for c in clientes},
        ...                        {v.getCodigo(): v for v in vendedores},
        ...                        {p.getCodigo(): p for p in productos})
        >>> isinstance(compras, list)
        True
    '''
    compras = []
    if not path.exists():
```

```
        return compras

    with path.open("r", encoding="utf-8") as f:
        for line in f:
            line = line.strip()
            if not line:
                continue
            rec = json.loads(line)

            # Fecha de compra
            y, m, d = map(int, rec["fecha"].split("-"))
            fecha = Fecha(d, m, y)

            # Fecha vencimiento de factura (si existe)
            fecha_venc = None
            if rec.get("fechaVencFactura"):
                y2, m2, d2 = map(int, rec["fechaVencFactura"].split("-"))
                fecha_venc = Fecha(d2, m2, y2)

            cliente = clientes_by_codigo.get(rec["cliente_codigo"])
            vendedor = vendedores_by_codigo.get(rec["vendedor_codigo"])

            if cliente is None or vendedor is None:
                print(f"Warning: Cliente/Vendedor missing in {rec['codigo']}, skipping")
                continue

            items = []
            for i in rec["items"]:
                producto = productos_by_codigo.get(i["producto_codigo"])
                if producto is None:
                    print(f"Warning: Producto {i['producto_codigo']} not found, skipping")
                    continue
                items.append(ItemCompra(producto, i["cantidad"]))

            compra = Compra(
                codigo=rec["codigo"],
                fechaCompra=fecha,
                cliente=cliente,
                vendedor=vendedor,
                items=items,
                nFactura=rec.get("nFactura", ""),
                modoPago=rec.get("modoPago", ""),
                fechaVencFactura=fecha_venc,
                valido=bool(rec.get("valido", False)),
                total=float(rec.get("total", 0.0)),
                listado=[productos_by_codigo.get(p) for p in rec.get("listado", []) if p in
    productos_by_codigo],
            )
            compras.append(compra)

    return compras

# Menu

class Menu:
  def __init__(self, direccion: str):
    # Para usarse luego
    self.__direccion = direccion

    # Leer listas de la base de datos
    self.__clientes   = load_clientes(Path(self.__direccion + "/dbClientes.txt"))
```

```
    self.__vendedores = load_vendedores(Path(self.__direccion + "/dbVendedores.txt"))
    self.__productos  = load_productos(Path(self.__direccion + "/dbProductos.txt"))

    # Listas de solo de codigos para cargar compras
    clientes_by   = {c.getCodigo(): c for c in self.__clientes}
    vendedores_by = {v.getCodigo(): v for v in self.__vendedores}
    productos_by  = {p.getCodigo(): p for p in self.__productos}

    # Leer lista de compras de la base de datos
    self.__compras = load_compras(Path(self.__direccion + "/dbCompras.txt"), clientes_by,
    vendedores_by, productos_by)

  def run(self): # ------------------------------------------------------------> Run
    # Menu principal
    print("\n")
    print("////////// Menu principal ///////////")
    print("1 - Consultar informacion")
    print("2 - Crear informacion")
    print("3 - Borrar informacion")
    print("4 - Menu compras")
    print("0 - Salir")

    print("\n")
    switch = input("Ingrese una opcion:")
    if switch == "1":   self.menuConsultar()
    elif switch == "2": self.menuCrear()
    elif switch == "3": self.menuBorrar()
    elif switch == "4": self.menuComprar()
    elif switch == "0":
      print("... Guardando datos y saliendo del programa ...")
      save_all(
        self.__direccion,
        self.__clientes,
        self.__vendedores,
        self.__productos,
        self.__compras
      )
      sys.exit()
    else:
      print("--- Opcion invalida ---")
      self.run()


  def menuConsultar(self): # ------------------------------------------------> Menu
    Consultar
    print("\n")
    print("////////// Consultar ///////////")
    print("1 - Listar clientes")
    print("2 - Listar vendedores")
    print("3 - Listar productos")
    print("4 - Listar compras")
    print("0 - Volver")

    print("\n")
    switch = input("Ingrese una opcion:")
    if switch =="1":   self.listar("clientes")
    elif switch =="2": self.listar("vendedores")
    elif switch =="3": self.listar("productos")
    elif switch =="4": self.listar("compras")
    elif switch =="0": self.run()
    else:
```

```
    print("--- Opcion invalida ---")
    self.menuConsultar()
  self.run()


def menuCrear(self): # ----------------------------------------------------> Menu Crear
  print("\n")
  print("////////// Menu crear //////////")
  print("1 - Crear cliente")
  print("2 - Crear vendedor")
  print("3 - Crear producto")
  print("0 - Volver")

  print("\n")
  switch = input("Ingrese una opcion:")
  if switch =="1":   self.crear("clientes")
  elif switch =="2": self.crear("vendedores")
  elif switch =="3": self.crear("productos")
  elif switch =="0": self.run()
  else:
    print("--- Opcion invalida ---")
    self.menuCrear()


def menuBorrar(self): # ----------------------------------------------------> Menu Borrar
  print("\n")
  print("////////// Menu borrar //////////")
  print("1 - Borrar cliente")
  print("2 - Borrar vendedor")
  print("3 - Borrar producto")
  print("0 - Volver")

  print("\n")
  switch = input("Ingrese una opcion:")
  if switch =="1":   self.borrar("clientes")
  elif switch =="2": self.borrar("vendedores")
  elif switch =="3": self.borrar("productos")
  elif switch =="0": self.run()
  else:
    print("--- Opcion invalida ---")
    self.menuBorrar()


def menuComprar(self): # ----------------------------------------------------> Menu
  Comprar
  print("\n")
  print("////////// Menu comprar //////////")
  print("1 - Crear compra")
  print("2 - Suspender compra")
  print("0 - Volver")

  print("\n")
  switch = input("Ingrese una opcion:")
  if switch =="1":   self.compra("crear")
  elif switch =="2": self.compra("suspender")
  elif switch =="0": self.run()
  else:
    print("--- Opcion invalida ---")
    self.menuComprar()
```

```
def listar(self, tipo): # ----------------------------------------------------> Listar
  print("\n")
  if tipo =="clientes":
    print("======== Listado de clientes ========")
    for c in self.__clientes:
        c.imprimirCliente()

  elif  tipo =="vendedores":
    print("======== Listado de vendedores ========")
    for v in self.__vendedores:
        v.imprimirVendedor()

  elif  tipo =="productos":
    print("======== Listado de productos ========")
    for p in self.__productos:
        p.imprimirProducto()

  elif  tipo =="compras":
    print("======== Listado de compras ========")
    for c in self.__compras:
        c.imprimirCompra()

def crear(self, tipo): # -----------------------------------------------------> Crear
  print("\n")
  if tipo =="clientes":
    print("=== Creacion de cliente ===")
    try:
      # Tomar valores del usuario
      codigo = input("Codigo: ")
      CI = input("CI: ")
      nombre = input("Nombre: ")
      correoE = input("Correo electronico: ")
      puntos = input("Puntos acumulados: ")

      # Convertir fechas de lenguaje natural a formato fijo
      fechaNac = Fecha(0,0,0)
      fechaNac.parseFecha(input("Fecha de nacimiento (DD/MM/YYYY): "))
      fechaIng = Fecha(0,0,0)
      fechaIng.parseFecha(input("Fecha de ingreso (DD/MM/YYYY): "))

      # Crear objeto cliente
      cliente = Cliente(codigo,
                        CI,
                        nombre,
                        fechaNac,
                        fechaIng,
                        correoE,
                        float(puntos))

      cliente.imprimirCliente() # Imprimir instancia cliente
      self.__clientes.append(cliente) # Agregar instancia a la lista
      print("Cliente creado exitosamente.")

    # Imprimir en caso de error
    except ValueError as e:
      print(f"Error creando Cliente: {e}.")


  elif  tipo =="vendedores":
    # Obtencion y validacion de datos para creacion de vendedores
    print("=== Creacion de vendedor ===")
```

```
    try:
        # Toma de datos generales
        codigo = input("Codigo: ")
        CI = input("CI: ")
        nombre = input("Nombre: ")
        correoE = input("Correo electronico: ")
        sueldo = input("Sueldo: ")

        # Guardar horarios separados por coma
        horarios_str = input("Horarios (separados por coma): ")
        horarios = [h.strip() for h in horarios_str.split(',')]

        # Conversion de fecha de lenguaje natural
        fechaNac = Fecha(0,0,0)
        fechaNac.parseFecha(input("Fecha de nacimiento (DD/MM/YYYY): "))
        fechaIng = Fecha(0,0,0)
        fechaIng.parseFecha(input("Fecha de ingreso (DD/MM/YYYY): "))

        # Crear instancia de Vendedor
        vendedor = Vendedor(codigo,
                            CI,
                            nombre,
                            fechaNac,
                            fechaIng,
                            correoE,
                            float(sueldo),
                            horarios)

        vendedor.imprimirVendedor() # Imprimir instancia
        self.__vendedores.append(vendedor) # Agregar instancia a la lista
        print("Vendedor creado exitosamente.")

    # En caso de errores
    except ValueError as e:
        print(f"Error creando Vendedor: {e}.")

elif tipo == "productos":
    # Obtencion y validacion de datos para creacion de productos
    print("=== Creacion de producto ===")

    try:
      # Datos generales
      codigo = input("Codigo: ")
      nombre = input("Nombre: ")
      marca = input("Marca: ")
      modelo = input("Modelo: ")
      cantidad = input("Cantidad: ")
      unidad = input("Unidad: ")
      precio = input("Precio: ")
      activo_str = input("Activo (s/n): ").lower().strip()

      # Conversion de fecha
      fechaFab = Fecha(0, 0, 0)
      fechaFab.parseFecha(input("Fecha de fabricacion (DD/MM/YYYY): "))
      fechaVen = Fecha(0, 0, 0)
      fechaVen.parseFecha(input("Fecha de vencimiento (DD/MM/YYYY): "))

      # Instancia de producto
      producto = Producto(
          codigo=codigo,
          nombre=nombre,
```

```
            marca=marca,
            modelo=modelo,
            cantidad=float(cantidad),
            unidad=unidad,
            precio=float(precio),
            activo=(activo_str == "s"),
            fechaVen=fechaVen,
            fechaFab=fechaFab
        )

        producto.imprimirProducto() # Imprimir instancia
        self.__productos.append(producto) # Agregar instancia a la lista
        print("Producto creado exitosamente.")

      # En caso de errores
      except ValueError as e:
        print(f"Error creando producto: {e}. Verifique los valores ingresados.")

    self.run()

  def borrar(self, tipo): # ----------------------------------------------------> Borrar
    print("\n")

    if tipo =="clientes":
      # Obtencion y validacion de datos para eliminacion de clientes
      print("=== Borrado de clientes ===")
      self.listar("clientes")
      codigo = input("\nCodigo del cliente a borrar: ")
      found_cliente = None
      for cliente in self.__clientes[:]:
        if isinstance(cliente, Cliente):
            if cliente.getCodigo() == codigo:
              found_cliente = cliente
              break
        else:
             print(f"Adventencia: Saltando item invalido: {cliente}")


      if found_cliente:
          # Eliminar cliente de la lista y borrar datos
          self.__clientes.remove(found_cliente)
          found_cliente.eliminarDatosCliente()
          print("Cliente eliminado.")
          self.listar("clientes")
      else:
          print("Cliente no encontrado.")


    elif  tipo =="vendedores":
      # Obtencion y validacion de datos para eliminacion de vendedores
      print("=== Borrado de vendedores ===")
      self.listar("vendedores")
      codigo = input("\nCodigo del vendedor a borrar: ")
      found_vendedor = None
      for vendedor in self.__vendedores[:]:
        if isinstance(vendedor, Vendedor):
            if vendedor.getCodigo() == codigo:
              found_vendedor = vendedor
              break
        else:
             print(f"Adventencia: Saltando item invalido: {vendedor}")
```

```
    if found_vendedor:
        # Eliminar vendedor de la lista y borrar datos
        self.__vendedores.remove(found_vendedor)
        found_vendedor.eliminarDatosVendedor()
        print("Vendedor eliminado.")
        self.listar("vendedores")
    else:
        print("Vendedor no encontrado.")


  elif  tipo =="productos":
    # Obtencion y validacion de datos para eliminacion de productos
    print("=== Borrado de productos ===")
    self.listar("productos")
    codigo = input("\nCodigo del producto a borrar: ")
    found_producto = None
    for producto in self.__productos[:]:
      if isinstance(producto, Producto):
          if producto.getCodigo() == codigo:
            found_producto = producto
            break
      else:
           print(f"Adventencia: Saltando item invalido: {producto}")


    if found_producto:
        # Eliminar vendedor de la lista y borrar datos
        self.__productos.remove(found_producto)
        found_producto.eliminarDatosProducto()
        print("Producto eliminado.")
        self.listar("productos")
    else:
        print("Producto no encontrado.")

  self.run()

def __buscar(self, tipo, codigo): # ----------------------------------------> __buscar()
  # Metodo privado para la busqueda de elementos a traves de sus codigos
  if tipo == "clientes":
    for cliente in self.__clientes:
      if isinstance(cliente, Cliente):
          if cliente.getCodigo() == codigo:
            return cliente
      else:
           print(f"Adventencia: Saltando item invalido: {cliente}")

  elif tipo == "vendedores":
    for vendedor in self.__vendedores:
      if isinstance(vendedor, Vendedor):
          if vendedor.getCodigo() == codigo:
             return vendedor
      else:
           print(f"Adventencia: Saltando item invalido: {vendedor}")

  elif tipo == "productos":
    for producto in self.__productos:
      if isinstance(producto, Producto):
          if producto.getCodigo() == codigo:
            return producto
```

```
        else:
            print(f"Adventencia: Saltando item invalido: {producto}")

    elif tipo == "compras":
      for compra in self.__compras:
        if isinstance(compra, Compra):
            if compra.getCodigo() == codigo:
              return compra
        else:
            print(f"Adventencia: Saltando item invalido: {compra}")

    return None


def compra(self, tipo): # ------------------------------------------------------> Crear
    Compra
    print("\n") # Logica programatica de compra
    if tipo == "crear":
        print("=== Creacion de compras ===")
        codigo = input("Codigo: ")

        # Convertir de lenguaje natural a formato consistente
        fechaCompra = Fecha(0, 0, 0)
        try:
            fechaCompra.parseFecha(input("Fecha de compra (DD/MM/AAAA): "))
        except ValueError as e:
            print(f"Error creando Compra: Formato de Fecha invalido - {e}")
            self.menuComprar()
            return
        # nFactura y modoPago
        nFactura = input("Numero de factura: ")
        modoPago = input("Modo de pago (ej: efectivo, tarjeta): ")

        # Convertir de lenguaje natural a formato consistente
        fechaVencFactura = Fecha(0, 0, 0)
        try:
            fechaVencFactura.parseFecha(input("Fecha de vencimiento factura (DD/MM/AAAA): ")
    )
        except ValueError:
            fechaVencFactura = None

        # Especificar cliente
        cliente_codigo = input("Codigo cliente: ")
        cliente = self.__buscar("clientes", cliente_codigo)
        if not isinstance(cliente, Cliente):
            print("Cliente no encontrado o invalido.")
            self.menuComprar()
            return

        # Especificar vendedor
        vendedor_codigo = input("Codigo vendedor: ")
        vendedor = self.__buscar("vendedores", vendedor_codigo)
        if not isinstance(vendedor, Vendedor):
            print("Vendedor no encontrado o invalido.")
            self.menuComprar()
            return

        # Especificar items (productos y cantidades)
        items = []
        listado = []
        while True:
```

```
            producto_codigo = input("Producto (Codigo): ")
            producto = self.__buscar("productos", producto_codigo)
            if not isinstance(producto, Producto):
                print("No agregado: producto no encontrado o invalido.")
                continue

            try:
                # Validacion de cantidades
                cantidad = int(input("Cantidad: "))
                if cantidad <= 0:
                    print("La cantidad tiene que ser positiva.")
                    continue
                if producto.getCantidad() < cantidad:
                    print("No hay suficiente stock.")
                    continue
                item = ItemCompra(producto, cantidad)
                items.append(item)
                listado.append(producto)
            except ValueError:
                print("Cantidad invalida. Por favor ingrese un numero.")
                continue

            if input("Desea agregar otro item? (s/n): ").lower() == "n":
                break
        # Ultima verificacion
        if len(items) == 0:
            print("No se agregaron items a la compra.")
            self.menuComprar()
            return

        # Calcular total de compra
        total = sum(it.getSubtotal() for it in items)

        compra = Compra(
            codigo=codigo,
            fechaCompra=fechaCompra,
            cliente=cliente,
            vendedor=vendedor,
            items=items,
            nFactura=nFactura,
            fechaVencFactura=fechaVencFactura,
            modoPago=modoPago,
            listado=listado,
            valido=True,
            total=total
        )

        compra.imprimirCompra() # Imprimir objeto
        self.__compras.append(compra) # Agregar objeto a la lista
        print("Compra creada exitosamente.")


    elif  tipo =="suspender":
      print("=== Suspencion de compras ===")
      # Leer codigo de compra
      codigo = input("Codigo de la compra a suspender: ")
      found_compra = None
      for compra in self.__compras[:]:
        # Verificar que compra sea una instancia de Compra
        if isinstance(compra, Compra):
            if compra.getCodigo() == codigo:
```

```
                found_compra = compra
                break
        else:
            print(f"Adventencia: Saltando item invalido: {compra}")


      if found_compra:
          self.__compras.remove(found_compra) # Quitar compra de la lista
          found_compra.suspenderCompra() # Suspender compra y reponer stock
          print("Compra suspendida.")
      else:
          print("Compra no encontrada.")

    self.run()

# Fecha

class Fecha:
  def __init__(self,
               dia: int,
               mes: int,
               anio: int):
    # Atributos
    self.__dia = dia
    self.__mes = mes
    self.__anio = anio

  # Getters
  def getDia(self) -> int:
    return self.__dia

  def getMes(self) -> int:
    return self.__mes

  def getAnio(self) -> int:
    return self.__anio

  # Convertidor de string de fecha en lenguaje natural
  # a formato fijo, usando parser y dateparser
  def parseFecha(self, text: str):
    fecha =  self.__parse_to_date(text).strftime("%d/%m/%Y")
    self.__dia = fecha.split("/")[0]
    self.__mes = fecha.split("/")[1]
    self.__anio = fecha.split("/")[2]

  # Funcion utilizada por parseFecha
  def __parse_to_date(self, s: str):
    try:
        return parser.parse(s).date()
    except Exception:
        dt = dateparser.parse(s)
        if dt is None:
            raise ValueError(f"Could not parse date string: {s}")
        return dt.date()

  # Retornar los atributos como un string de fecha
  def toString(self) -> str:
    return f"{self.__dia}/{self.__mes}/{self.__anio}"

  # Eliminar fecha
  def eliminarFecha(self) -> bool:
```

```
    self.__dia = 0
    self.__mes = 0
    self.__anio = 0
    return True

  # Retornar string con formato ISO
  def to_iso(self) -> str:
    d = int(self.getDia())
    m = int(self.getMes())
    y = int(self.getAnio())
    return f"{y:04d}-{m:02d}-{d:02d}"

  # Retornar diccionario de fecha en ISO
  def to_dict(self) -> dict:
      return {"iso": self.to_iso()}

  # Leer fechas del diccionario
  @classmethod
  def from_dict(cls, d: dict) -> "Fecha":
      if isinstance(d, dict) and "iso" in d:
          y, m, d = map(int, d["iso"].split("-"))
          return cls(d, m, y)
      if isinstance(d, str):
          y, m, d = map(int, d.split("-"))
          return cls(d, m, y)
      raise ValueError(f"Formato invalido de fecha: {d}")

# Persona

class Persona:
  def __init__(self,
               codigo: str,
               CI: str,
               nombre: str,
               fechaNac: Fecha,
               fechaIng: Fecha,
               correoE: str):

    # Atributos
    self.__codigo = codigo      # privado
    self.__CI = CI              # privado
    self.__nombre = nombre      # privado
    self.__fechaNac = fechaNac  # privado
    self.__fechaIng = fechaIng  # privado
    self.__correoE = correoE    # privado

  # Getters
  def getCodigo(self) -> str:
    return self.__codigo

  def getCI(self) -> str:
    return self.__CI

  def getNombre(self) -> str:
    return self.__nombre

  def getFechaNac(self) -> Fecha:
    return self.__fechaNac

  def getFechaIng(self) -> Fecha:
    return self.__fechaIng
```

```
  def getCorreoE(self) -> str:
    return self.__correoE

  # ----- Otros metodos -----
  def imprimirPersona(self) -> None:
      print("\n======== Persona ==========")
      print(f"Codigo: {self.__codigo}")
      print(f"CI: {self.__CI}")
      print(f"Nombre: {self.__nombre}")
      print(f"Fecha de Nacimiento: {self.__fechaNac.toString()}")
      print(f"Fecha de Ingreso: {self.__fechaIng.toString()}")
      print(f"Correo: {self.__correoE}")

  def eliminarPersona(self) -> None:
      self.__codigo = ""
      self.__CI = ""
      self.__nombre = ""
      self.__fechaNac = self.__fechaNac.eliminarFecha()
      self.__fechaIng = self.__fechaIng.eliminarFecha()
      self.__correoE = ""

  # Leer fechas del diccionario
  def _fechas_from(d: dict) -> tuple["Fecha", "Fecha"]:
      y1, m1, d1 = map(int, d["fechaNac"].split("-"))
      fechaNac = Fecha(d1, m1, y1)
      y2, m2, d2 = map(int, d["fechaIng"].split("-"))
      fechaIng = Fecha(d2, m2, y2)
      return fechaNac, fechaIng

  # Retornar diccionario de Persona
  def to_persona_dict(self) -> dict:
    return {
      "codigo": self.getCodigo(),
      "CI": self.getCI(),
      "nombre": self.getNombre(),
      "fechaNac": self.getFechaNac().to_iso(),
      "fechaIng": self.getFechaIng().to_iso(),
      "correoE": self.getCorreoE(),
    }



# Vendedor

class Vendedor(Persona):
  def __init__(self,
               codigo: str,
               CI: str,
               nombre: str,
               fechaNac: Fecha,
               fechaIng: Fecha,
               correoE: str,
               sueldo: float,
               horarios: list):

    super().__init__(codigo, CI, nombre, fechaNac, fechaIng, correoE)
    self.__sueldo = sueldo
    self.__horarios = horarios  # lista de horarios
```

```
  def getSueldo(self) -> float:
    return self.__sueldo

  def getHorarios(self) -> list:
    return self.__horarios

  def imprimirVendedor(self) -> None:
    print("\n======== Vendedor ==========")
    self.imprimirPersona()
    print(f"\nSueldo: {self.__sueldo}")
    print(f"Horarios: {self.__horarios}")

  def eliminarDatosVendedor(self) -> None:
    self.sueldo = 0.0
    self.horarios = []
    self.eliminarPersona()
    print("Datos del vendedor eliminados.")

  # Retornar diccionario de atributos
  def to_dict(self) -> dict:
    base = self.to_persona_dict()
    base["sueldo"] = self.getSueldo()
    base["horarios"] = list(self.getHorarios())
    return base

  # Retornar atributos de diccionario
  @classmethod
  def from_dict(cls, d: dict) -> "Vendedor":
      fN, fI = Persona._fechas_from(d)
      return cls(
          d["codigo"], d["CI"], d["nombre"],
          fN, fI, d["correoE"], float(d["sueldo"]), list(d.get("horarios", []))
      )

# Cliente

class Cliente(Persona):
  def __init__(self,
               codigo: str,
               CI: str,
               nombre: str,
               fechaNac: Fecha,
               fechaIng: Fecha,
               correoE: str,
               puntos: float):

      super().__init__(codigo, CI, nombre, fechaNac, fechaIng, correoE)
      self.__puntos = puntos
  # Getters
  def getPuntos(self) -> float:
    return self.__puntos

  # Metodos de utilidad
  def imprimirCliente(self) -> None:
    print("\n======== Cliente ==========")
    self.imprimirPersona()
    print(f"\nPuntos acumulados: {self.__puntos}")

  def eliminarDatosCliente(self) -> bool:
    self.__puntos = 0.0
    self.eliminarPersona()
```

```
    print("Datos del cliente eliminados.")

  # Retornar diccionario de atributos
  def to_dict(self) -> dict:
    base = self.to_persona_dict()
    base["puntos"] = self.getPuntos()
    return base

  # Retornar atributos de diccionario
  @classmethod
  def from_dict(cls, d: dict) -> "Cliente":
      fN, fI = Persona._fechas_from(d)
      return cls(
          d["codigo"], d["CI"], d["nombre"],
          fN, fI, d["correoE"], float(d.get("puntos", 0.0))
      )

# Producto

class Producto:
    def __init__(self,
                 codigo: str,
                 nombre: str,
                 marca: str,
                 modelo: str,
                 cantidad: float,
                 unidad: str,
                 precio: float,
                 activo: bool,
                 fechaVen: Fecha,
                 fechaFab: Fecha):
        self.__codigo = codigo
        self.__nombre = nombre
        self.__marca = marca
        self.__modelo = modelo
        self.__cantidad = cantidad
        self.__unidad = unidad
        self.__precio = precio
        self.__activo = activo
        self.__fechaVen = fechaVen
        self.__fechaFab = fechaFab

    def getCodigo(self) -> str:
        return self.__codigo

    def getNombre(self) -> str:
        return self.__nombre

    def getMarca(self) -> str:
        return self.__marca

    def getModelo(self) -> str:
        return self.__modelo

    def getCantidad(self) -> float:
        return self.__cantidad

    def getUnidad(self) -> str:
        return self.__unidad

    def getPrecio(self) -> float:
```

```
    return self.__precio

def isActivo(self) -> bool:
    return self.__activo

def getFechaVen(self) -> Fecha:
    return self.__fechaVen

def getFechaFab(self) -> Fecha:
    return self.__fechaFab

def setCantidad(self, cantidad: float) -> None:
    self.__cantidad = cantidad

def imprimirProducto(self) -> None:
    print("\n======== Producto ==========")
    print(f"Codigo: {self.__codigo}")
    print(f"Nombre: {self.__nombre}")
    print(f"Marca: {self.__marca}")
    print(f"Modelo: {self.__modelo}")
    print(f"Cantidad: {self.__cantidad} {self.__unidad}")
    print(f"Precio: {self.__precio}")
    print(f"Activo: {'Si' if self.__activo else 'No'}")
    print(f"Fabricacion: {self.__fechaFab.toString() if self.__fechaFab else '-'}")
    print(f"Vencimiento: {self.__fechaVen.toString() if self.__fechaVen else '-'}")

def eliminarDatosProducto(self) -> bool:
  self.__codigo = ""
  self.__nombre = ""
  self.__marca = ""
  self.__modelo = ""
  self.__cantidad = 0.0
  self.__unidad = ""
  self.__precio = 0.0
  self.__activo = False
  self.__fechaVen = None
  self.__fechaFab = None
  print("\nDatos del producto eliminados.")
  return True

def to_dict(self) -> dict:
  return {
    "codigo": self.getCodigo(),
    "nombre": self.getNombre(),
    "marca": self.getMarca(),
    "modelo": self.getModelo(),
    "cantidad": self.getCantidad(),
    "unidad": self.getUnidad(),
    "precio": self.getPrecio(),
    "activo": self.isActivo(),
    "fechaVen": self.getFechaVen().to_dict() if self.__fechaVen else None,
    "fechaFab": self.getFechaFab().to_dict() if self.__fechaFab else None
  }

@classmethod
def from_dict(cls, d: dict) -> "Producto":
  fechaVen = Fecha.from_dict(d["fechaVen"]) if d.get("fechaVen") else None
  fechaFab = Fecha.from_dict(d["fechaFab"]) if d.get("fechaFab") else None
  return cls(
    d["codigo"],
    d["nombre"],
```

```
        d["marca"],
        d["modelo"],
        float(d["cantidad"]),
        d["unidad"],
        float(d["precio"]),
        bool(d["activo"]),
        fechaVen,
        fechaFab
      )

# ItemCompra

class ItemCompra:
  def __init__(self,
               producto: Producto,
               cantidad: int):
    self.__producto = producto
    self.__cantidad = cantidad

  def getProducto(self) -> Producto:
    return self.__producto

  def getCantidad(self) -> int:
    return self.__cantidad

  def getPrecio(self) -> float:
    return self.__producto.getPrecio()

  def getSubtotal(self) -> float:
    return self.getPrecio() * self.__cantidad

  def imprimirItemCompra(self) -> None:
    print("\n======== ItemCompra ==========")
    print(f"- Producto: ")
    self.__producto.imprimirProducto()
    print(f"\nCantidad: {self.__cantidad}")
    print(f"- Precio unitario: {self.__producto.getPrecio()}")
    print(f"- Subtotal: {self.getPrecio() * self.__cantidad}")

  def eliminarItemCompra(self) -> bool:
    self.__producto = None
    self.__cantidad = 0
    print("\nDatos del item de compra eliminados.")
    return True

  def to_record(self) -> dict:
    return {
      "producto_codigo": self.getProducto().getCodigo(),
      "cantidad": self.getCantidad(),
    }

  @classmethod
  def from_record(cls, rec: dict, productos_by_codigo: dict[str, Producto]) -> "ItemCompra":
    prod = productos_by_codigo.get(rec["producto_codigo"])
    if prod is None:
      raise ValueError(f"Producto {rec['producto_codigo']} no existe (al cargar compra)")
    return cls(prod, int(rec["cantidad"]))

# Compra

class Compra:
```

```
def __init__(self,
             codigo: str,
             fechaCompra: "Fecha",
             cliente: "Cliente",
             vendedor: "Vendedor",
             items: list["ItemCompra"],
             nFactura: str = "",
             fechaVencFactura: "Fecha" = None,
             modoPago: str = "",
             listado: list["Producto"] = None,
             valido: bool = False,
             total: float = 0.0):
  self.__codigo = codigo
  self.__fechaCompra = fechaCompra
  self.__cliente = cliente
  self.__vendedor = vendedor
  self.__items = list(items) if items else []
  self.__nFactura = nFactura
  self.__fechaVencFactura = fechaVencFactura
  self.__modoPago = modoPago
  self.__listado = listado if listado else []
  self.__valido = valido
  self.__total = total

  # actualizar stock
  for item in self.__items:
    producto = item.getProducto()
    producto.setCantidad(producto.getCantidad() - item.getCantidad())
    print(f"Stock del producto {producto.getNombre()} actualizado.")

# ---- Getters -----
def getCodigo(self) -> str:
  return self.__codigo

def getFecha(self) -> "Fecha":
  return self.__fechaCompra

def getCliente(self) -> "Cliente":
  return self.__cliente

def getVendedor(self) -> "Vendedor":
  return self.__vendedor

def getItems(self) -> list["ItemCompra"]:
  return list(self.__items)

def getFactura(self) -> str:
  return self.__nFactura

def getFechaVencFactura(self) -> "Fecha":
  return self.__fechaVencFactura

def getModoPago(self) -> str:
  return self.__modoPago

def getListado(self) -> list["Producto"]:
  return list(self.__listado)

def esValida(self) -> bool:
  return self.__valido
```

```
def getTotal(self) -> float:
    if not self.__valido:
        return sum(it.getSubtotal() for it in self.__items)
    return self.__total

def suspenderCompra(self) -> None:
  self.__valido = False
  # Actualizar stock de productos
  for item in self.__items:
    producto = item.getProducto()
    producto.setCantidad(producto.getCantidad() + item.getCantidad())
    print(f"Stock del producto {producto.getNombre()} actualizado.")

def imprimirCompra(self) -> None:
  estado = "VaLIDA" if self.__valido else "ANULADA"
  print("\n======== Compra ==========")
  print(f"Codigo de compra: {self.__codigo}")
  print(f"Fecha: {self.__fechaCompra.toString()}")
  print(f"Factura: {self.__nFactura or '-'}  |  Pago: {self.__modoPago or '-'}  |  Vence:
  {self.__fechaVencFactura.toString() if self.__fechaVencFactura else '-'}")
  print(f"Estado: {estado}")
  print("Cliente:")
  self.__cliente.imprimirCliente()
  print("\nVendedor:")
  self.__vendedor.imprimirVendedor()
  print("\nItems:")
  for it in self.__items:
    it.imprimirItemCompra()
  print(f"\nTotal: {self.getTotal():.2f}")

# Retornar diccionario de atributos
def to_record(self) -> dict:
  return {
      "codigo": self.getCodigo(),
      "fecha": self.getFecha().to_iso(),
      "cliente_codigo": self.getCliente().getCodigo(),
      "vendedor_codigo": self.getVendedor().getCodigo(),
      "items": [it.to_record() for it in self.getItems()],
      "nFactura": self.getFactura(),
      "modoPago": self.getModoPago(),
      "fechaVencFactura": self.getFechaVencFactura().to_iso() if self.getFechaVencFactura
  () else None,
      "listado": [p.getCodigo() for p in self.getListado()],
      "valido": self.esValida(),
      "total": float(self.getTotal()),
  }

# Retornar atributos de diccionario
@classmethod
def from_record(
    cls,
    rec: dict,
    clientes_by_codigo: dict[str, "Cliente"],
    vendedores_by_codigo: dict[str, "Vendedor"],
    productos_by_codigo: dict[str, "Producto"],
) -> "Compra":
    fecha = Fecha(*map(int, rec["fecha"].split("-")[::-1]))
    cliente = clientes_by_codigo.get(rec["cliente_codigo"])
    vendedor = vendedores_by_codigo.get(rec["vendedor_codigo"])

    if cliente is None:
```

```
            raise ValueError(f"Cliente {rec['cliente_codigo']} no existe (al cargar compra)")
        if vendedor is None:
            raise ValueError(f"Vendedor {rec['vendedor_codigo']} no existe (al cargar compra)"
    )

        items = [ItemCompra.from_record(r, productos_by_codigo) for r in rec.get("items", [])]
        listado = [productos_by_codigo[cod] for cod in rec.get("listado", []) if cod in
    productos_by_codigo]
        fechaVencFactura = Fecha(*map(int, rec["fechaVencFactura"].split("-")[::-1])) if rec.
    get("fechaVencFactura") else None

        return cls(
            codigo=rec["codigo"],
            fechaCompra=fecha,
            cliente=cliente,
            vendedor=vendedor,
            items=items,
            nFactura=rec.get("nFactura", ""),
            fechaVencFactura=fechaVencFactura,
            modoPago=rec.get("modoPago", ""),
            listado=listado,
            valido=bool(rec.get("valido", False)),
            total=float(rec.get("total", 0.0)),
        )

# Valores de prueba

# Fechas
f1 = Fecha(1, 1, 1990)
f2 = Fecha(15, 6, 1985)
f3 = Fecha(10, 10, 2000)
f4 = Fecha(5, 5, 1995)
f5 = Fecha(20, 3, 2024)
f6 = Fecha(20, 4, 2024)

# Clientes
c1 = Cliente(CI="11111111", codigo="01", correoE="maria@mail.com", fechaIng=f1, fechaNac=f2,
     nombre="Maria", puntos=10.0)
c2 = Cliente(CI="22221111", codigo="02", correoE="carlos@mail.com", fechaIng=f3, fechaNac=f4
    , nombre="Carlos", puntos=300.0)
listaClientes = [c1, c2]

# Vendedores
v1 = Vendedor(CI="1111235123", codigo="01", correoE="juan@mail.com", fechaIng=f1, fechaNac=
    f2, nombre="Juan", horarios=["9-18"], sueldo=123.2)
v2 = Vendedor(CI="123123123123", codigo="02", correoE="diego@mail.com", fechaIng=f3,
    fechaNac=f5, nombre="Diego", horarios=["10-19"], sueldo=1231233.2)
listaVendedores = [v1, v2]

# Productos
p1 = Producto(codigo="P00", nombre="Laptop", marca="Dell", modelo="Inspiron", cantidad=100,
    unidad="unid", precio=1000.1, activo=True, fechaVen=f6, fechaFab=f1)
p2 = Producto(codigo="P01", nombre="Tablet", marca="Samsung", modelo="Tab A", cantidad=200,
    unidad="unid", precio=500.5, activo=True, fechaVen=f6, fechaFab=f2)
p3 = Producto(codigo="P02", nombre="Celular", marca="Apple", modelo="iPhone", cantidad=300,
    unidad="unid", precio=200.2, activo=True, fechaVen=f6, fechaFab=f3)
listaProductos = [p1, p2, p3]

# ItemCompra
i1 = ItemCompra(cantidad=2, producto=p1)   # 2 Laptops
i2 = ItemCompra(cantidad=10, producto=p2)  # 10 Tablets
```

```
i3 = ItemCompra(cantidad=20, producto=p3)  # 20 Celulares

# Compras
compra1 = Compra(
    codigo="C01",
    nFactura="F001",
    fechaCompra=f1,
    fechaVencFactura=f6,
    modoPago="Tarjeta",
    listado=[p1, p2, p3],
    valido=True,
    total=i1.getSubtotal() + i2.getSubtotal() + i3.getSubtotal(),
    cliente=c1,
    vendedor=v1,
    items=[i1, i2, i3]
)

compra2 = Compra(
    codigo="C02",
    nFactura="F002",
    fechaCompra=f5,
    fechaVencFactura=f6,
    modoPago="Efectivo",
    listado=[p2, p2, p2],
    valido=True,
    total=(i2.getSubtotal() * 3),  # 3 veces el mismo item
    cliente=c2,
    vendedor=v2,
    items=[i2, i2, i2]
)

listaCompras = [compra1, compra2]

# Guardar
save_all(
    "./db",
    listaClientes,
    listaVendedores,
    listaProductos,
    listaCompras
)

# Resultado

menu = Menu(direccion="./db")
menu.run()
```