RESUMEN DE TODO LO VISTO EN MODULO 1

**1. ¿Qué son los MÓDULOS en Python?**

Un **módulo** es simplemente un archivo .py que contiene código organizado: funciones, clases, variables.

Sirven para:

- Organizar el proyecto en partes claras.
- Reutilizar código.
- Evitar que todo esté en un solo archivo gigante.
- Separar responsabilidades (inventario, ventas, reportes).

**✓ Cómo crear un módulo**

Solo crea un archivo .py:

inventario.py

ventas.py

reportes.py

principal.py

Ejemplo dentro de inventario.py:

```
def agregar_producto():
    print("Producto agregado!")
```

**✓ Cómo importarlo en otro archivo**

```
from inventario import agregar_producto


agregar_producto()
```

---

**🧩 2. ¿Qué es una FUNCIÓN?**

Una **función** es un bloque de código que realiza una tarea específica.

**✓ Ejemplo básico**

```
def saludar():
    print("Hola!")
```

saludar()

**✓ Función con parámetros y retorno**

```
def sumar(a, b):
  return a + b


resultado = sumar(5, 3)
print(resultado)   # 8
```

**✓ ¿Por qué usar funciones?**

- Te evitan repetir código.
- Organizan mejor tu programa.
- Permiten reutilizar lógica.

---

### 🧩 3. ¿Qué es una VARIABLE?

Una variable es un "contenedor" donde guardas información.

**✓ Tipos comunes**

```
entero = 10
decimal = 3.14
texto = "Hola"
booleano = True
lista = [1, 2, 3]
diccionario = {"nombre": "Laptop", "precio": 1200}
```

**✓ Buenas prácticas**

- Usar nombres descriptivos.
- Escribir en minúsculas y con guiones bajos: precio_total.

---

### 🧩 4. ¿Qué son las LAMBDA?

Una **lambda** es una función anónima ("sin nombre"), usada para operaciones rápidas.

✓ **Ejemplo básico**

```
sumar = lambda x, y: x + y

print(sumar(3, 4))  # 7
```

✓ **Ejemplo de tu proyecto (cálculo de venta)**

```
calcular_total = lambda precio, cantidad: precio * cantidad * (1 - descuento)
```

Se usa porque:

- Es corta.
- No vale la pena crear una función completa def.

---

🧩 **5. ¿Qué es un DICCIONARIO? (Muy usado en tu proyecto)**

Almacena información en pares **clave: valor**.

✓ **Ejemplo**

```
producto = {
   "nombre": "Laptop",
   "precio": 1500,
   "stock": 10
}
```

✓ **Acceder a valores**

```
print(producto["nombre"])
```

---

🧩 **6. ¿Qué es una LISTA?**

Una lista es un conjunto de elementos ordenados.

Ejemplo:

```
frutas = ["manzana", "pera", "uva"]
```

Tu proyecto usa listas para guardar el historial de ventas:

```
historial_ventas = []
```

---

🧩 **7. JSON — ¿Qué es y para qué sirve?**

JSON es un formato usado para guardar datos en texto estructurado (muy usado en APIs).

Es parecido a un diccionario de Python.

**✓ Ejemplo JSON**

```
{
  "nombre": "Laptop",
  "precio": 1200,
  "stock": 10
}
```

**✓ Guardar JSON en Python**

```
import json

with open("datos.json", "w") as archivo:
    json.dump(producto, archivo, indent=4)
```

**✓ Cargar JSON**

```
with open("datos.json", "r") as archivo:
    datos = json.load(archivo)
```

---

### 🧩 8. CSV — ¿Qué es y para qué sirve?

CSV significa **Comma Separated Values**
(Valores separados por comas).

Ejemplo CSV:

```
nombre,precio,stock
Laptop,1200,10
Tablet,800,5
```

**✓ Guardar CSV**

```
import csv

with open("inventario.csv", "w", newline="") as archivo:
    escritor = csv.writer(archivo)
```

```
    escritor.writerow(["nombre","precio","cantidad"])
    escritor.writerow(["Laptop",1200,10])
```

**✓ Cargar CSV**

```
with open("inventario.csv") as archivo:
    lector = csv.reader(archivo)
    for fila in lector:
        print(fila)
```

---

### 🧩 9. ¿Qué es el CRUD?

CRUD significa:

- **C**reate → Crear
- **R**ead → Leer
- **U**pdate → Actualizar
- **D**elete → Eliminar

Tu sistema lo aplica así:

| Operación | Función |
|---|---|
| Crear | agregar_producto() |
| Leer | ver_productos() |
| Actualizar | actualizar_producto() |
| Eliminar | eliminar_producto() |

---

### 🧩 10. Resumen de tus módulos

**✓ inventario.py**

Maneja:

- productos
- agregar
- ver
- actualizar

- eliminar
- guardar/cargar CSV

✓ **ventas.py**

Maneja:

- registrar ventas
- historial de ventas
- descuentos
- guardar/cargar CSV

✓ **reportes.py**

Crea reportes:

- productos más vendidos
- ventas por marca
- ingresos
- desempeño del inventario

✓ **principal.py**

Contiene el **menú principal**:

- muestra opciones
- llama las funciones según la selección

---

🧩 **11. EJEMPLO FINAL (super simple)**

```
from inventario import agregar_producto

from ventas import registrar_venta

from reportes import reporte_ingresos


agregar_producto()

registrar_venta()

reporte_ingresos()
```

RESUMEN 2 MAS PASO A PASO

**Objetivo de la clase**

Al terminar sabrás:

1. Qué es un **módulo** y cómo organizar un proyecto en módulos.
2. Cómo crear **funciones**, cómo usarlas y por qué.
3. Qué son las **lambdas** y cuándo usarlas.
4. Qué tipos de **variables** existen y buenas prácticas.
5. Cómo usar **CSV** y **JSON** para guardar/cargar datos.
6. Cómo implementar un **CRUD** completo y un menú interactivo.
7. Cómo probar tu proyecto en VS Code y resolver errores comunes.

Duración estimada práctica: 45–60 min (pero lo explico paso a paso aquí).

---

**1. Preparación (prerrequisitos)**

- Tener Python 3 instalado.
- Un editor (VS Code recomendado).
- Carpeta del proyecto: TIENDA_ELECTRONICA (o el nombre que prefieras).
- Archivos: principal.py, inventario.py, ventas.py, reportes.py.

---

**2. Módulos — ¿qué son y por qué usarlos?**

**Definición:** Un módulo = un archivo .py que contiene funciones, variables y/o clases.
**Ventajas:** organización, reutilización, claridad.

**Ejemplo práctico**

Crea inventario.py con:

```
# inventario.py

inventario = {}

def agregar_producto():
    pass
```

En principal.py importas:

from inventario import agregar_producto

**Clase:** Piensa en cada módulo como una “caja de herramientas” con responsabilidad única (inventario, ventas, reportes).

---

**3. Funciones — construcción y buenas prácticas**

**Qué son:** bloques de código con nombre que realizan una tarea.
**Estructura básica:**

```
def nombre_funcion(param1, param2):
    # cuerpo
    return resultado
```

**Buenas prácticas:**

- Nombre descriptivo en minúsculas y con guion bajo: agregar_producto.
- Funciones cortas (máx. una responsabilidad).
- Validar entradas dentro de la función.
- Documentar con docstring.

**Ejemplo (agregar producto):**

```
def agregar_producto():
    nombre = input("Nombre: ")
    precio = float(input("Precio: "))
    # validar y guardar en inventario
```

---

**4. Lambdas — funciones anónimas para operaciones pequeñas**

**Qué son:** expresiones lambda que crean funciones pequeñas sin def.
**Sintaxis:** lambda argumentos: expresión

**Cuándo usarlas:** cálculos sencillos, como aplicar descuento en una venta.

**Ejemplo:**

```
descuento = 0.2
calcular_total = lambda precio, qty: round(precio * qty * (1 - descuento), 2)
total = calcular_total(100, 2)  # 160.0
```

**Profesor:** usa lambda cuando la función es pequeña y se usa in situ. Si la lógica crece, usa def.

---

**5. Variables y tipos más usados**

- int — enteros (ej. stock = 10)
- float — decimales (ej. precio = 199.99)
- str — texto (ej. nombre = "Laptop")
- bool — True/False (ej. disponible = True)
- list — lista ordenada (ej. ventas = [])
- dict — diccionario (ej. producto = {"nombre":"Laptop","precio":700})

**Regla:** nombres descriptivos: precio_unitario, cantidad_vendida.

---

**6. Estructuras de datos del proyecto (cómo las usamos)**

- **Inventario**: dict anidado con IDs

inventario = {

1: {"nombre": "Laptop", "marca": "Acer", "categoria": "Portátiles", "precio": 700.0, "stock": 10}

}

- **Ventas**: list de dict

ventas = [

{"cliente":"Ana","tipo":"vip","producto":"Laptop","marca":"Acer","cantidad":1,"descuento":0.2,"total":560}

]

**Profesor:** usar IDs para inventario facilita referencias y evita duplicidad de nombres.

---

**7. CRUD — paso a paso (con ejemplos)**

**Crear (Create)**

Función agregar_producto():

1. Pedir nombre, marca, categoría, precio, stock, garantía.
2. Validar (no negativos, tipo correcto).

3. Generar ID incremental.
4. Guardar en inventario[pid] = {...}.

**Leer (Read)**

Función ver_productos():

- Iterar for pid, p in inventario.items(): e imprimir formateado.

**Actualizar (Update)**

Función actualizar_producto():

1. Pedir palabra clave.
2. Buscar coincidencias (lista).
3. Si varias, pedir selección (nº).
4. Preguntar campo y nuevo valor.
5. Validar y asignar.

**Eliminar (Delete)**

Función eliminar_producto():

- Similar a actualizar: busca, confirma, del inventario[pid].

---

**8. CSV — guardar y cargar (práctico)**

**Guardar inventario (CSV)**

Encabezado: id,nombre,marca,categoria,precio,stock,garantia_meses

```
import csv

def guardar_inventario_csv(ruta="inventario.csv"):
  with open(ruta, "w", newline="", encoding="utf-8") as f:
    writer = csv.writer(f)
    writer.writerow(["id","nombre","marca","categoria","precio","stock","garantia_meses"])
    for pid, p in inventario.items():
      writer.writerow([pid, p["nombre"], p["marca"], p["categoria"], p["precio"], p["stock"], p["garantia_meses"]])
```

**Cargar inventario (CSV)**

```
def cargar_inventario_csv(ruta="inventario.csv"):
```

```
  with open(ruta, "r", encoding="utf-8") as f:
    reader = csv.DictReader(f)
    inventario.clear()
    for row in reader:
      pid = int(row["id"])
      inventario[pid] = {
        "nombre": row["nombre"],
        "marca": row["marca"],
        "categoria": row["categoria"],
        "precio": float(row["precio"]),
        "stock": int(row["stock"]),
        "garantia_meses": int(row["garantia_meses"])
      }
```

**Consejo de profesor:** siempre validar encabezado exacto y envolver en try/except para evitar que el archivo mal formado rompa el programa.

---

**9. JSON — guardar y cargar (cuando usarlo)**

**JSON** guarda estructuras completas y mantiene tipos (útil cuando quieres persistencia más rica).

**Guardar:**

```
import json
with open("inventario.json","w",encoding="utf-8") as f:
  json.dump(inventario, f, indent=2, ensure_ascii=False)
```

**Cargar:**

```
with open("inventario.json","r",encoding="utf-8") as f:
  inventario = json.load(f)
```

**Profesor:** JSON es ideal para guardar dicts anidados directamente; CSV es mejor para tablas simples (filas/columnas).

---

**10. Menú interactivo (principal.py) — estructura y control de errores**

```
def main():
  while True:
    print("1. Agregar producto")
    # ...
    opcion = input("Elija: ")
    if opcion == "1": agregar_producto()
    elif opcion == "0": break
```

**Profesor:** envolver llamadas con try/except para que errores no cierren la app.

---

**11. Errores comunes y cómo solucionarlos (tipo profesor)**

- ImportError: verificar nombres exactos y que el archivo está guardado.
- KeyError: intentar acceder a una clave de diccionario que no existe (validar con in).
- ValueError: conversión int/float inválida — usar try/except.
- FileNotFoundError: archivo CSV/JSON no existe — informar al usuario.
- Ellipsis (...) en datos: indica que se cargaron placeholders; limpia CSV.

---

**12. Ejercicios prácticos tipo clase (hazlos y verifica)**

**Ejercicio 1 — Agregar y listar**

1. Ejecuta python principal.py.
2. Opción 1: agrega un producto nuevo.
3. Opción 2: confirma que aparece en la lista.

**Ejercicio 2 — Vender y revisar stock**

1. Opción 5: vende 2 unidades (buscar por palabra clave).
2. Opción 6: ver historial.
3. Opción 2: verificar decremento de stock.

**Ejercicio 3 — Guardar y cargar CSV**

1. Opción 11: guardar inventario.
2. Sal del programa y borra inventario (o reinicia).
3. Opción 12: cargar inventario — verificar restauración.

**Ejercicio 4 — Corregir un CSV con errores**

1. Abre ventas.csv y coloca una fila inválida (ej. falta campo).
2. Ejecuta cargar_ventas_csv() y observa contador de filas omitidas.

---

**13. Resumen de todas las variables importantes (con explicación y ejemplo)**

- inventario (dict): almacena productos por id.
  Ejemplo: inventario[1]["precio"] = 700.0
- ventas_historial / historial_ventas (list): lista de dict con ventas.
  Ejemplo: ventas_historial.append({"cliente":"Ana", "producto":"Laptop", "cantidad":1})
- DESCUENTOS / discounts (dict): reglas por tipo de cliente.
  Ejemplo: descuento = DESCUENTOS.get("vip", 0)
- calcular_total (lambda): función anónima para calcular total.
  Ejemplo: calcular_total = lambda precio, qty: precio * qty * (1 - descuento)
- pid, pid_seleccionado (int): identificador de producto.
  Ejemplo: nuevo_id = max(inventario.keys()) + 1
- Strings de interacción: input("...") → siempre validar el retorno antes de usar.

---

**14. Ejemplo completo mínimo (Código compacto)**

Pega esto en un archivo mini_demo.py y ejecútalo para ver la idea central:

```
# mini_demo.py

inventario = {}

def agregar_producto_simple():

  pid = max(inventario.keys())+1 if inventario else 1

  nombre = input("Nombre: ")

  precio = float(input("Precio: "))

  inventario[pid] = {"nombre":nombre,"precio":precio,"stock":int(input("Stock: "))}

def ver_productos_simple():

  for pid,p in inventario.items():

    print(pid,p)

# flujo rápido

agregar_producto_simple()
```

ver_productos_simple()

---

**15. Buenas prácticas finales (como profesor)**

- Documenta funciones con """docstring""".
- Mantén cada módulo con una responsabilidad.
- Valida siempre entradas de usuario.
- Evita código duplicado: encapsula en funciones.
- Usa csv.DictReader/DictWriter para más robustez.
- Versiona con git (commits pequeños).

---

**16. Recursos y siguiente paso (recomendado por el profesor)**

- Practica: implementa filtros por marca y rango de precios.
- Añade persistencia JSON además de CSV.
- Agrega pruebas unitarias básicas (unittest) para funciones críticas (ej. cálculo de total).

CSV Y JSON

### 📘 1. ¿Qué es un archivo CSV?

CSV significa **Comma Separated Values** → son archivos donde los datos están separados por comas.

Ejemplo de un CSV:

nombre,precio,cantidad

Manzanas,2500,10

Peras,3000,5

📌 Son ideales para listas y tablas simples.

---

### 📘 2. ¿Qué es un archivo JSON?

JSON significa **JavaScript Object Notation**, y almacena datos como **diccionarios** o **listas de diccionarios**.

Ejemplo:

```
[
 {
  "id": 1,
  "nombre": "Televisor",
  "marca": "Samsung",
  "precio": 1200000
 }
]
```

📌 Es perfecto para estructuras complejas como inventarios y ventas.

---

## 🟦 3. Mejorar Manejo de CSV en Python

---

### ✅ GUARDAR CSV — versión mejorada

✓ Maneja errores
✓ Limpia los datos
✓ Evita archivos corruptos

```
import csv


def guardar_csv(ruta, lista, encabezados):
  try:
    with open(ruta, "w", newline="", encoding="utf-8") as archivo:
      escritor = csv.DictWriter(archivo, fieldnames=encabezados)
      escritor.writeheader()
      escritor.writerows(lista)

    print(f"Archivo CSV guardado correctamente en: {ruta}")

  except PermissionError:
    print("❌ No tienes permisos para escribir en esa carpeta.")
```

```
    except Exception as e:
        print(f"❌ Error inesperado al guardar CSV: {e}")
```

---

### ✅ CARGAR CSV — versión mejorada

✓ Detecta encabezados incorrectos
✓ Convierte tipos automáticamente
✓ Ignora filas inválidas
✓ Evita crasheos

```
import csv


def cargar_csv(ruta, encabezados):
    datos = []
    errores = 0

    try:
        with open(ruta, "r", encoding="utf-8") as archivo:
            lector = csv.DictReader(archivo)

            # Verifica que los encabezados coincidan
            if lector.fieldnames != encabezados:
                print("❌ El archivo CSV no tiene los encabezados correctos.")
                return [], 1

            for fila in lector:
                try:
                    datos.append(fila)
                except:
                    errores += 1
```

```
        print(f"Archivo CSV cargado. Errores encontrados: {errores}")
        return datos, errores

    except FileNotFoundError:
        print("❌ El archivo CSV no existe.")
        return [], 1
    except Exception as e:
        print(f"❌ Error al cargar CSV: {e}")
        return [], 1
```

---

## 🟩 4. Mejorar Manejo de JSON en Python

---

### ✅ GUARDAR JSON — versión PRO

✓ Permite caracteres especiales (ñ, á, é...)
✓ Formato más bonito
✓ Maneja errores

```
import json

def guardar_json(ruta, data):
    try:
        with open(ruta, "w", encoding="utf-8") as archivo:
            json.dump(data, archivo, indent=4, ensure_ascii=False)
        print(f"Archivo JSON guardado correctamente en: {ruta}")

    except PermissionError:
        print("❌ No tienes permisos para guardar el archivo.")
    except Exception as e:
        print(f"❌ Error al guardar JSON: {e}")
```

---

✅ **CARGAR JSON — versión PRO**

✓ Revisa si el archivo está vacío
✓ Verifica el formato
✓ Evita errores de decodificación

```
import json
import os


def cargar_json(ruta):
    if not os.path.exists(ruta):
        print("❌ El archivo JSON no existe.")
        return []

    try:
        with open(ruta, "r", encoding="utf-8") as archivo:
            contenido = archivo.read().strip()

            if not contenido:
                print("⚠ El archivo JSON está vacío.")
                return []

            return json.loads(contenido)

    except json.JSONDecodeError:
        print("❌ El archivo JSON tiene formato inválido.")
        return []
    except Exception as e:
        print(f"❌ Error al cargar JSON: {e}")
        return []
```

---

## 🟩 5. ¿Cuándo usar CSV y cuándo usar JSON?

| Tipo de archivo | Cuándo usarlo |
|---|---|
| **CSV** | Datos tipo tabla simples: inventarios, ventas, listas |
| **JSON** | Datos complejos, relaciones, objetos, configuraciones |

---

## 🟦 6. Ejemplo completo — Guardar Inventario en CSV y JSON

Supongamos este inventario:

```
inventario = [
  {"id": 1, "nombre": "Laptop", "marca": "HP", "precio": 2000000, "stock": 5},
  {"id": 2, "nombre": "Mouse", "marca": "Logitech", "precio": 80000, "stock": 15}
]
```

---

### 🔷 Guardar CSV

```
encabezados = ["id", "nombre", "marca", "precio", "stock"]
guardar_csv("inventario.csv", inventario, encabezados)
```

---

### 🔷 Guardar JSON

```
guardar_json("inventario.json", inventario)
```

---

## 🟪 7. Grandes Mejores Prácticas (importante en tu proyecto)

**✓ Siempre valida encabezados en CSV**

**✓ Siempre usa try / except**

**✓ Evita perder datos con archivos corruptos**

**✓ Usa UTF-8 siempre**

**✓ Usa indent=4 en JSON para que sea legible**

**✓ Usa ensure_ascii=False para soportar español**