Programación1-Laboratorio2

Juan Diego Hernández Castellanos juanhernandez125@unisangil.edu.co
Sindy Carolina Pinilla Murcia Sindypinilla224@unisangil.edu.co

Abstract-This work presents the development of a set of Python programs designed to address five structured computational problems: geometric figure calculation, log data processing, sensor network simulation, product inventory management, and banking system simulation. Each program was created following the first three phases of the problem-solving methodology—analysis, design, and implementation. The solutions were constructed using modular programming, separating the logic into independent functions, scripts, and classes when required. For the geometric calculator, the implementation was divided into dedicated modules for computation and interface handling. The log processing script uses iterative structures to extract user activity patterns from sequential data. The sensor network simulation employs matrix traversal with nested loops to identify critical temperature values. The product management system and banking simulator were built using custom classes to model objects and behaviors. Overall, the project focuses on the structured construction, modular design, and algorithmic flow that support each solution.

Keywords modular programming, Python scripts, flow diagrams, geometric calculations, data processing, sensor networks, inventory management, banking simulation, object-oriented programming

Resumen - Este trabajo presenta el desarrollo de un conjunto de programas en Python diseñados para resolver cinco problemas computacionales estructurados: cálculo de figuras geométricas, procesamiento de datos de registros, simulación de una red de sensores, gestión de productos e implementación de un sistema bancario. Cada programa fue construido siguiendo las primeras tres fases de la metodología de resolución de problemas: análisis, diseño e implementación. Las soluciones se desarrollaron aplicando programación modular, separando la lógica en funciones, scripts y clases según la necesidad. En la calculadora geométrica, la implementación se dividió en módulos específicos para los cálculos y la interfaz. El procesamiento de logs utiliza estructuras iterativas para extraer patrones de actividad de datos secuenciales. La simulación de sensores recorre una matriz mediante bucles anidados para identificar valores críticos de temperatura. El sistema de gestión de productos y el simulador bancario fueron implementados mediante clases que modelan objetos y comportamientos. En conjunto, el proyecto se enfoca en la construcción estructurada, el diseño modular y el flujo algorítmico que soporta cada solución.

Palabras clave: programación modular, scripts en Python, diagramas de flujo, cálculos geométricos, procesamiento de datos, redes de sensores, gestión de inventario, simulación bancaria, programación orientada a objetos

I. INTRODUCCIÓN

El desarrollo de soluciones computacionales estructuradas permite abordar diferentes tipos de problemas mediante técnicas de diseño modular y principios fundamentales de programación. En este trabajo se presentan cinco implementaciones en Python, cada una orientada a resolver un caso específico mediante el uso de funciones, clases, estructuras iterativas y manejo de datos. Para cada ejercicio se aplicaron las fases iniciales de la metodología de resolución de problemas, organizando el análisis, el diseño y la codificación de manera independiente. El enfoque principal se centra en la construcción lógica de cada programa, en la división de responsabilidades entre módulos y en el uso adecuado de estructuras de control para garantizar claridad y coherencia en su funcionamiento.

II. METODOLOGÍA

La **metodología** utilizada para el desarrollo de los programas se basó en las primeras fases del proceso de resolución de problemas en programación: análisis, diseño, codificación, depuración y mantenimiento. Cada una de estas etapas se aplicó de manera independiente a los diferentes ejercicios propuestos, garantizando un desarrollo estructurado y coherente para cada solución.

En la fase de **análisis**, se identificaron las entradas, procesos y salidas de cada programa, además de definir el flujo general de ejecución. Para los ejercicios que requerían una organización más clara de sus módulos —como el procesamiento de datos a gran escala y el conjunto de problemas algorítmicos— se elaboró un diagrama de programación modular, permitiendo dividir el sistema en componentes funcionales y mejorar la separación de responsabilidades.

La etapa de **diseño** permitió establecer la estructura lógica de los programas, seleccionando las funciones, clases y estructuras de control necesarias para su implementación.

Documento enviado para revisión en2025.Fue desarrollado con los conocimientos y recursos brindados por la Unisangil Chiquinquirá

Posteriormente, en la fase de codificación, se desarrollaron los scripts en Python siguiendo una organización clara por ejercicios y aplicando técnicas de programación modular cuando correspondía.

Durante la **depuración**, se verificó el funcionamiento de cada módulo mediante pruebas controladas que permitieron detectar y corregir errores lógicos y de ejecución. Finalmente, la fase de mantenimiento consistió en revisar, ajustar y optimizar el código para asegurar su legibilidad, estabilidad y potencial reutilización en futuros proyectos o ejercicios similares.

A. Análisis.

En esta fase se identificaron los elementos fundamentales de cada ejercicio: los datos de entrada requeridos, los procesos necesarios para generar los resultados y las salidas esperadas. Para los ejercicios que involucraban múltiples funciones o componentes —como el procesamiento de datos a gran escala y los problemas algorítmicos— se elaboró un **diagrama de programación modular** con el fin de dividir el problema en módulos independientes y facilitar la comprensión del flujo general del programa. El análisis permitió establecer la estructura base de cada solución antes de avanzar a las etapas de diseño y codificación.

Procesamiento de datos a gran escala: Para este ejercicio se planteó un sistema capaz de leer y procesar registros de actividad de usuarios almacenados de forma secuencial. El análisis se centró en identificar las estructuras necesarias para recorrer los datos, contabilizar accesos por usuario y generar una lista final con los resultados. Se definieron módulos separados para la lectura de datos, el procesamiento mediante ciclos repetitivos y la construcción del resultado final, lo que permitió organizar las funciones de manera clara y modular.

```
Procesamiento de Datos a Gran Escala
    Módulo de Entrada de Datos
        leer_registros()

    Módulo de Procesamiento
        procesar_registros()
        contar_accesos()

    Módulo de Resultados
        generar_lista_final()
```

Fig1: identación diagrama programación modular(datos)

Calculadora de figuras geométricas: En este ejercicio se diseñó una calculadora capaz de obtener el área de diferentes figuras geométricas mediante funciones independientes. Durante el análisis se identificaron las entradas requeridas para cada figura, los cálculos asociados y la interacción del usuario con el sistema. Se dividió el programa en tres componentes principales: un módulo principal para la ejecución general, un módulo de cálculos que contiene las funciones de cada figura y un módulo de interfaz encargado de mostrar opciones y solicitar datos, lo que permitió una estructura ordenada y modular.

```
Calculadora de Figuras Geométricas
    Módulo Principal (Principal.py)
        ejecutar_aplicacion()

    Módulo de Cálculo (Figuras.py)
        area_cuadrado()
        area_rectangulo()
        area_triangulo()
        area_circulo()
        otras_figuras()

    Módulo de Interfaz (Interfaz.py)
        mostrar_menu()
        solicitar_datos()
        mostrar_resultado()
```

Fig2: identacion diagrama programación modular (caluladora)

B. Diseño.

La fase de diseño consistió en establecer la estructura lógica y funcional de cada programa antes de su implementación. Para los ejercicios que requerían una organización modular, se definió la distribución de responsabilidades entre los diferentes scripts y funciones, permitiendo separar el procesamiento, la interacción con el usuario y las operaciones internas de cada sistema. El diseño incluyó la selección de estructuras de control apropiadas, como ciclos iterativos y condicionales, así como la definición de funciones específicas para encapsular tareas puntuales.

En el caso de la calculadora de figuras geométricas, el diseño se basó en dividir el programa en un módulo principal, un módulo de cálculos y un módulo de interfaz, facilitando la comunicación entre componentes. Para el procesamiento de datos a gran escala, el diseño se centró en el recorrido secuencial de los registros y en la acumulación de resultados por usuario mediante funciones independientes. Los demás ejercicios se diseñaron siguiendo principios similares, asignando a cada clase, función o sección del código una responsabilidad bien definida para mantener claridad, coherencia y facilidad de mantenimiento.

C. Codificación

La fase de codificación consistió en implementar las estructuras y componentes definidos durante el diseño, utilizando el lenguaje Python como base para todos los ejercicios. Cada programa fue desarrollado siguiendo una organización clara por archivos, donde cada script cumplió una función específica dentro del sistema. Se implementaron funciones independientes para encapsular cálculos y procesos repetitivos, así como clases para los ejercicios que requerían modelar entidades con atributos y comportamientos propios.
Durante esta etapa se emplearon estructuras de control como

ciclos for y while, condicionales y manejo de datos mediante listas y matrices, según las necesidades de cada ejercicio. En los programas modulares, el código fue dividido en scripts separados para facilitar la comunicación entre componentes y mantener una estructura ordenada. La codificación se realizó de forma progresiva, verificando el comportamiento de cada función o clase conforme se implementaba para asegurar coherencia con el diseño planteado.

D. Depuración

En esta etapa se verificó el funcionamiento correcto de cada programa mediante pruebas dirigidas y revisión del comportamiento de las funciones y módulos. La depuración incluyó la identificación de errores lógicos, comprobación de condiciones incorrectas, validación de entradas y corrección de resultados inesperados durante la ejecución. Para ello, se realizaron ejecuciones parciales de cada script, comprobando que los ciclos, condicionales y estructuras de datos operaran de acuerdo con lo especificado en el diseño.

La depuración también permitió asegurar que los módulos se comunicaran correctamente entre sí en los ejercicios que empleaban programación modular. A partir de estas pruebas, se ajustaron valores, corrigieron cálculos, reorganizaron partes del código y se eliminaron fragmentos innecesarios, con el objetivo de obtener programas funcionales y consistentes.

E. Mantenimiento

La fase de mantenimiento consistió en realizar ajustes posteriores a la ejecución inicial de los programas, con el fin de mejorar su claridad, organización y estabilidad. Se revisó la estructura del código para optimizar funciones, simplificar secciones redundantes y asegurar que los nombres de variables y módulos fueran coherentes con su propósito. En los ejercicios modulares, se verificó que cada script mantuviera responsabilidades bien definidas y que pudiera ser modificado sin afectar el funcionamiento general del sistema.

Además, se adecuaron partes del código para facilitar su comprensión y futura reutilización, incorporando comentarios y una organización más limpia cuando fue necesario. El mantenimiento también contempló pequeñas mejoras destinadas a prevenir errores en usos posteriores, garantizando que cada programa quedara en un estado óptimo para futuras modificaciones o ampliaciones.

III. DESARROLLO

En esta sección se presentan las características principales de cada uno de los ejercicios desarrollados en el laboratorio. Se describe la estructura interna de los programas, las funciones o clases empleadas y el funcionamiento general de cada solución, destacando la lógica aplicada en su implementación.

A. Calculadora de Figuras Geométricas.

Este ejercicio consistió en diseñar un sistema capaz de calcular el área de diferentes figuras geométricas utilizando funciones específicas para cada una de ellas. El programa se estructuró en tres scripts: un módulo principal encargado del flujo general de ejecución, un módulo de cálculos que contiene las funciones matemáticas necesarias y un módulo de interfaz que administra la interacción con el usuario. El sistema permite seleccionar la figura deseada, solicitar los valores requeridos y mostrar el resultado final. La organización modular facilitó la separación de responsabilidades y permitió mantener el código claro y escalable.

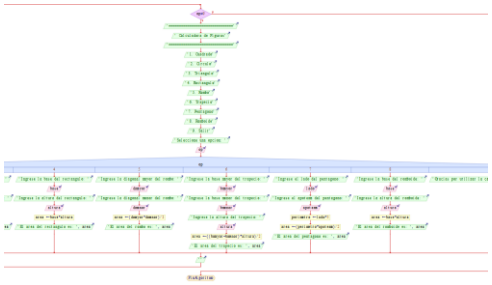

Fig1:diagrama deflujo calculadora

```
from Interfaz import(
    menu,
    solicitar_datos_cuadrado,
    solicitar_datos_circulo,
    solicitar_datos_triangulo,
    solicitar_datos_rectamgulo,
    solicitar_datos_rombo,
    solicitar_datos_trapecio,
    solicitar_datos_pentagono,
    solicitar_datos_romboide,
    mostrar_area_cuadrado,
    mostrar_area_circulo,
    mostrar_area_triangulo,
    mostrar_area_rectangulo,
    mostrar_area_rombo,
    mostrar_area_trapecio,
    mostrar_area_pentagono,
    mostrar_area_romboide
)
from Figuras import(
    area_cuadrado,
```

Fig. 2. Captura de la ejecución del programa de cálculo de áreas.

B. Procesamiento de datos a gran escala.

En este ejercicio se desarrolló un programa para procesar registros secuenciales de actividad de usuarios. El sistema lee datos que contienen nombre de usuario, hora de entrada y salida, procesándolos uno a uno mediante un ciclo while. Posteriormente, un ciclo for identifica la cantidad total de accesos por usuario. Los resultados se almacenan en una lista final compuesta por pares del tipo [usuario, número_de_accesos]. La implementación se dividió en módulos independientes que manejan la lectura, el procesamiento y la generación del resultado final, permitiendo un flujo estructurado y fácil de mantener.

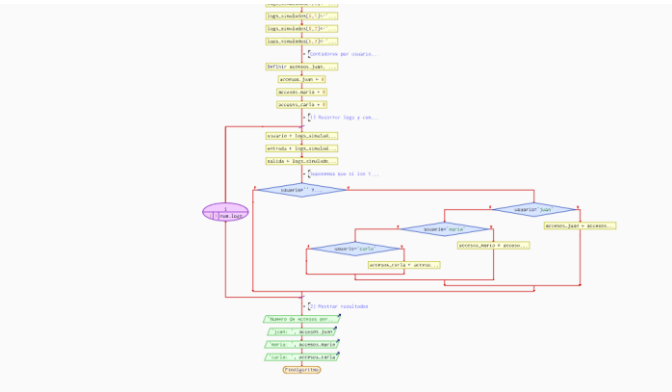

Fig 3: diagrama de flujo de procesamiento de datos

```python
# procesar_logs.py
from datos_logs import logs_simulados


# PROCESAR DATOS CON WHILE



logs_procesados = []  # copia local de los registros

i = 0
while i < len(logs_simulados):
    registro = logs_simulados[i]
    # Cada registro tiene: [usuario, entrada, salida]
    if len(registro) == 3:
        usuario, entrada, salida = registro
        logs_procesados.append([usuario, entrada, salida])
    i += 1  # siguiente registro


# 2. CONTAR ACCESOS CON FOR


usuarios_unicos = []

# Sacar usuarios sin repetir
for registro in logs_procesados:
    usuario = registro[0]
    if usuario not in usuarios_unicos:
        usuarios_unicos.append(usuario)

# Contar accesos por cada usuario
resultados = []  # [nombre_usuario, numero_de_accesos]

for usuario in usuarios_unicos:
    conteo = 0 #variable de conteo para sumar 1 por cada acceso
```

Fig 4 : captura del programa de procesamiento de datos

C. Simulación de Red de Sensores.

Este ejercicio consistió en la creación de una matriz que simula un conjunto de sensores distribuidos en una fábrica. La matriz es generada con valores aleatorios que representan temperaturas, y posteriormente es recorrida mediante ciclos anidados con el fin de identificar valores críticos que superan un umbral definido (por ejemplo, 80°C). El programa encapsula la detección en una función específicamente diseñada para recorrer la matriz y registrar los valores que exceden el límite. La simulación permite representar de forma básica el monitoreo de variables distribuidas en múltiples ubicaciones.

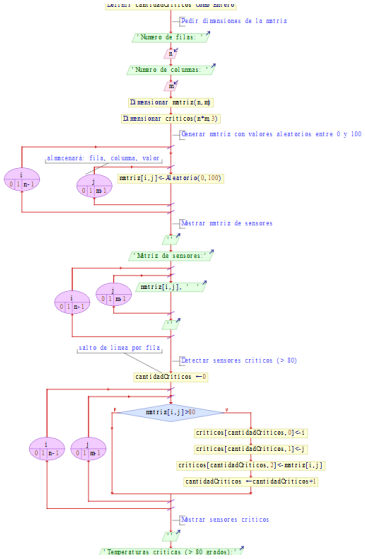

Fig 5: diagrama de flujo red de sensores

```python
You, 3 days ago | 1 author (You)
#simulación de red de sensores
import random

def generar_matriz (n,m):
    matriz = []
    for j in range(n):        You, 3 days ago • 3programasmas#1
        fila = []
        for i in range(m):
            fila.append(random.randint(0,100))
        matriz.append(fila)
    return matriz

def detectar_criticos(matriz):
    criticos = []
    for i in range(len(matriz)):
        for j in range(len(matriz[0])):
            if matriz[i][j]> 80:
                criticos.append((i,j,matriz[i][j]))
    return criticos

def mostrar_matriz(matriz):
    print("\nmatriz de sensores: ")
    for fila in matriz:
        print(fila)

def mostrar_criticos(criticos):
    print("\nTempeeraturas criticas(> 80 grados): ")
    if len(criticos) == 0:
        print("no se detectaron valores criticos")
    else:
        for sensor in criticos:
            print(f"sensor en [{sensor[0]}][{sensor[1]}]={sensor[2]}grados")

# programa principal
n= int(input("numero de filas: "))
m= int(input("numero de columnas: "))
```

Fig 6: código red de sensores

D. Sistema de gestión de productos.

Para este ejercicio se construyó una clase `Producto` que modela elementos de inventario mediante atributos como nombre, precio y cantidad. La clase incluye métodos para aumentar y disminuir el stock, lo que permite simular operaciones de compra y venta. Se creó una lista de productos que actúa como inventario, acompañada de una función encargada de mostrar la información actualizada. El diseño orientado a objetos permitió organizar claramente el comportamiento de cada producto y facilitar la modificación del inventario de forma controlada.

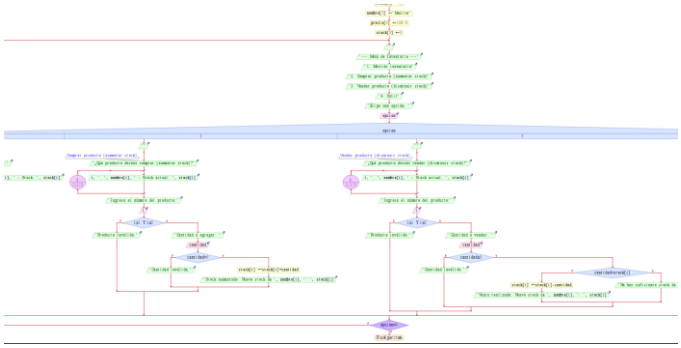

Fig 7: diagrama de flujo de gestion de productos

```python
#sistemas de gestion de productos
class Producto:
    def __init__(self, nombre, precio, cantidad):
        self.nombre = nombre
        self.precio = precio
        self.cantidad = cantidad

    def aumentar_stock(self,cantidad):
        self.cantidad += cantidad
    def disminuir_stock(self,cantidad):
        if cantidad > self.cantidad:
            print(f"No hay suficientes stock de {self.nombre}.")
        else:
            self.cantidad -= cantidad
    def mostrar_info(self):
        return f"{self.nombre}-Precio: {self.precio:.2f}-Stock: {self.cantidad}"

# Funciones
def crear_inventario():
    return [
        Producto("Mouse", 20.0, 15),
        Producto("Teclado", 35.0, 10),
        Producto("Monitor", 150.0, 5)
    ]
def mostrar_inventario(inventario):
    print("\nInventario Actual: ")
    for i, producto in enumerate(inventario):
        print(f"{i+1}. {producto.mostrar_info()}")
def seleccionar_producto(inventario):
    mostrar_inventario(inventario)
    try:
```

Fig 8: codigo de programa gestion de productos

E. Sistema bancario.

Este programa implementa un sistema básico de cuentas bancarias mediante la creación de una clase CuentaBancaria. La clase permite realizar depósitos, retiros y consultas de saldo, garantizando que las operaciones se ajusten a las reglas definidas (por ejemplo, evitar retiros mayores al saldo disponible). Además, se simuló la atención de clientes mediante una lista que funciona como cola, donde cada cliente espera su turno para realizar una operación. El sistema muestra el flujo de atención y la actualización del estado de cada cuenta conforme se procesan las transacciones.

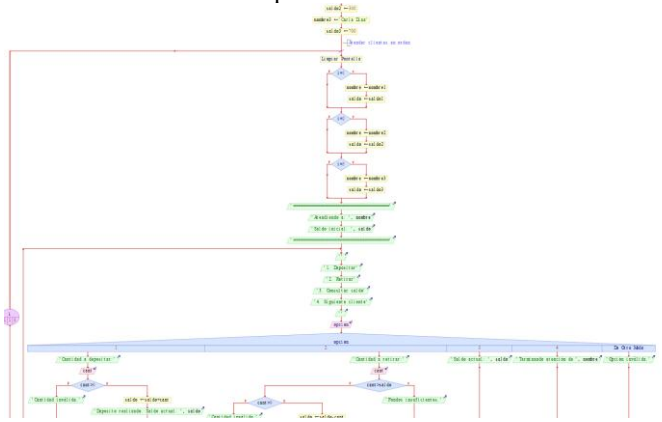

Fig 9: diagrama de flujo sistema de banco

```python
# Simulación de un sistema bancario

# Definición de la clase CuentaBancaria
class CuentaBancaria:
    def __init__(self, nombre_cliente, saldo_inicial=0):
        self.nombre_cliente = nombre_cliente
        self.saldo = saldo_inicial

    def depositar(self, cantidad):
        if cantidad > 0:
            self.saldo += cantidad
            print(f"Depósito de {cantidad} realizado con éxito. Nuevo saldo: {self.saldo}")
        else:
            print("La cantidad a depositar debe ser positiva.")

    def retirar(self, cantidad):
        if cantidad > self.saldo:
            print("Fondos insuficientes para realizar el retiro.")
        elif cantidad > 0:
            self.saldo -= cantidad
            print(f"Retiro de {cantidad} realizado con éxito. Nuevo saldo: {self.saldo}")
        else:
            print("La cantidad a retirar debe ser positiva.")

    def consultar_saldo(self):
        return self.saldo


# Función para simular el procesamiento de clientes en la cola
def proceso_clientes(cola):
    while cola:
        cliente = cola.pop(0)  # atender al primer cliente de la cola
```

Fig 10: codigo de sitema bancario

IV. Resultados

Los resultados obtenidos corresponden a la ejecución de cada uno de los programas desarrollados en el laboratorio. Para cada ejercicio se presenta la evidencia visual del funcionamiento del código, mostrando las salidas generadas, la estructura final de los datos procesados o la interacción del usuario con el sistema, según corresponda. Las capturas de pantalla permiten verificar que los módulos implementados operan de acuerdo con el diseño planteado y que los scripts desarrollados cumplen con los requisitos funcionales de cada problema.

A continuación, se presentan las evidencias correspondientes a cada ejercicio:

A. Calculadora de Figuras Geométricas

```
elcione una figura para calcular su area:
. cuadrado
. circulo
. triangulo
. rectangulo
. rombo
. trapecio
. pentangono
. romboide
.. salir
elecione una opcion: 4
ngrese la base del rectangulo:4
ngrese la altura del rectangulo:2
l area del rectangulo es: 8.0
ienvenido a la calculadora de figuras geometricas
elcione una figura para calcular su area:
```

Fig. 1. Captura de la ejecución del programa de cálculo de áreas.

B. Procesamiento de Datos a Gran Escala

```
Número de accesos por usuario:
juan: 3
maria: 2
ana: 1

Lista resultados (para el informe):
[['juan', 3], ['maria', 2], ['ana', 1]]
```

Fig. 2. Evidencia del procesamiento de registros y conteo de

accesos por usuario.*)*

C. Simulación de Red de Sensores

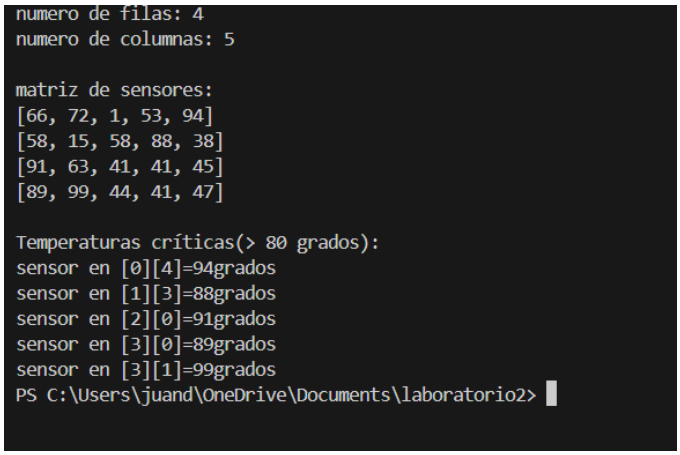

Fig. 3. Captura del recorrido de la matriz y detección de valores críticos.

D. Sistema de Gestión de Productos

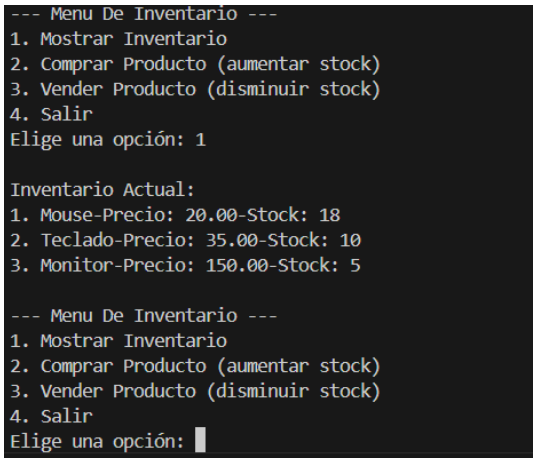

Fig. 4. Salida del inventario y operaciones de modificación de stock.

E. Simulador Bancario

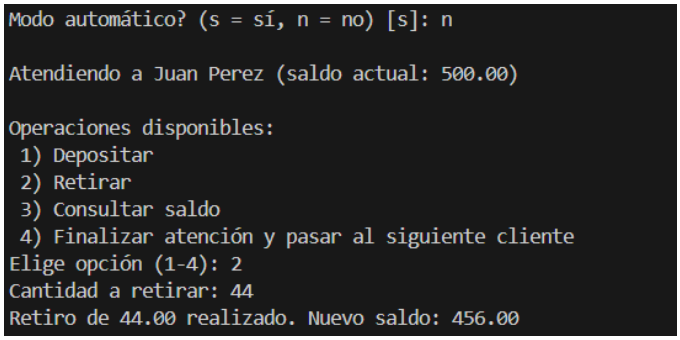

Fig. 5. Ejecución de operaciones bancarias y manejo de la cola de clientes.

V. Conclusión

El desarrollo de los cinco programas presentados demuestra la efectividad de aplicar la metodología de resolución de problemas en sus fases de análisis, diseño e implementación. La adopción de la programación modular permitió organizar la lógica de cada solución en funciones, módulos y clases, facilitando su comprensión, mantenimiento y escalabilidad. Cada caso específico, desde el cálculo de figuras geométricas hasta la simulación de sistemas bancarios y redes de sensores, evidenció la importancia de la estructuración de algoritmos y del uso adecuado de estructuras de control y recorridos de datos. Además, la implementación orientada a objetos en los sistemas más complejos contribuyó a modelar comportamientos y relaciones de manera intuitiva. En conjunto, el proyecto reafirma que un enfoque modular y bien planificado optimiza la eficiencia en la resolución de problemas computacionales y garantiza la claridad y coherencia en el diseño de programas en Python.