Fundamentos de Python para el Análisis de Datos

Caso-1: Aplicación del Lenguaje Python en el Análisis de Datos

Mariana trabaja en el departamento de ventas de una empresa de tecnología y recibe diariamente reportes de ventas en formato de texto con datos de diferentes clientes, montos de compra y fechas de transacción. Sin conocimientos avanzados en programación, ha decidido aprender Python para automatizar algunas tareas repetitivas, como calcular el total de ventas del día, identificar los clientes con compras mayores a cierto umbral y convertir los datos a un formato más estructurado.

Escenario: Mariana recibe un archivo con la siguiente información:
clientes = ["Juan", "Ana", "Luis", "Carlos"]
ventas = [1500.50, 2300.75, 1850.00, 920.30]
Ella necesita escribir un script en Python que le permita:

- Calcular el total de ventas del día.
- Determinar cuántos clientes realizaron compras mayores a $2,000.
- Convertir los valores numéricos a texto para generar un informe.
- Imprimir los resultados en consola.

NOTA: Al final del documento se anexa cuaderno Google_Colab con resultados

Preguntas Clave:

1. ¿Qué tipo de datos se están utilizando en la lista de clientes y en la lista de ventas?
   - clientes es una lista de strings (cadenas de texto). Cada elemento representa el nombre de un cliente.
   - ventas es una lista de floats (números decimales). Cada elemento indica el monto de la compra realizada por cada cliente.

2. ¿Cómo podría calcular el total de ventas usando una función de Python?
   Se puede usar la función incorporada sum):
   total_ventas = sum(ventas)
   sum) recorre automáticamente la lista y suma todos los valores.

3.- ¿Qué estructura de control puede utilizar para identificar a los clientes con compras mayores a $2,000?
   Usar un bucle for con una condicional if:
   For i in range(len(ventas)):
      if ventas[i] > 2000:
      print(f"{clientes[i]} hizo una compra mayor a $2,000.")

4. ¿Cómo puede convertir los datos numéricos en texto antes de imprimirlos en el informe?
   - print("3.1 Conversion simple de valores a texto")
     print()
     for i in range(len(clientes)):

```
        cliente= str(clientes[i]) venta = "${:,.2f}".format(ventas[i]) print(cliente + "
        realizó una compra de " + venta)

print(f"\nTotal de ventas del día: ${total_ventas}")
```

- En el caso de imprimir las cifras en palabras

```
def numero_a_palabras_decimal(numero):
    def numero_a_palabras(n):
        unidades    = ["", "uno", "dos", "tres", "cuatro", "cinco",
                       "seis", "siete", "ocho", "nueve"]
        especiales = ["diez", "once", "doce", "trece", "catorce", "quince",
                      "dieciséis", "diecisiete", "dieciocho", "diecinueve"]
        decenas    = ["", "", "veinte", "treinta", "cuarenta", "cincuenta",
                      "sesenta", "setenta", "ochenta", "noventa"]
        centenas  = ["", "ciento", "doscientos", "trescientos", "cuatrocientos",
                     "quinientos", "seiscientos", "setecientos", "ochocientos", "novecientos"]
        if n == 0:
            return "cero"
        elif n == 100:
            return "cien"
        texto = ""
        if n >= 1000:
            mil = n // 1000
            texto += "mil " if mil == 1 else numero_a_palabras(mil) + " mil "
            n %= 1000
        if n >= 100:
            texto += centenas[n // 100] + " "
            n %= 100
        if 10 <= n < 20:
            texto += especiales[n - 10]
            n = 0
        elif n >= 20:
            texto += decenas[n // 10]
            if n % 10 != 0:
                texto += " y " + unidades[n % 10]
            n = 0
        if 0 < n < 10:
            texto += unidades[n]
        return texto.strip()
    entero          = int(numero)
    centavos        = int(round((numero - entero) * 100))
    palabras_entero  = numero_a_palabras(entero)
    palabras_centavos = numero_a_palabras(centavos)
    return f"{palabras_entero} pesos con {palabras_centavos} centavos"
```

5.- ¿Cuáles son los entornos de desarrollo más adecuados para ejecutar este tipo de script y por qué?

Entorno	Ventajas principales
Jupyter Notebook	Perfecto para pruebas rápidas, visualización y documentación interactiva.
VS Code	Ligero, con autocompletado, depuración y control de versiones. Muy útil para proyectos organizados.
PyCharm	IDE profesional con análisis de código, refactorización y gestión avanzada de proyectos.
Thonny	Ideal para principiantes: interfaz simple y explicaciones paso a paso.
IDLE (oficial)	Viene incluido con Python. Básico, pero funcional para scripts simples.
Google Colab	Basado en la nube, solo se necesita un navegador y conexión a internet, integrado con Google Drive, tiene un entorno interactivo con ejecución celdas una a una, visualizar salidas intermedias y depurar fácilmente, puede instalar librerías, permite probar, documentar y mantener código organizado por secciones.

6.- Cuaderno Google-Colab

```
# Datos iniciales
clientes = ["Juan", "Ana", "Luis", "Carlos"]
ventas = [1500.50, 2300.75, 1850.00, 920.30]

# 1. Total de ventas
total_ventas = sum(ventas)

# 2. Compras mayores a $2000
compras_altas = sum(1 for venta in ventas if venta > 2000)

# 3. Conversión simple de valores a texto
print()
print("3.1 Conversion simple de valores a texto")
print()
for i in range(len(clientes)):
    cliente = str(clientes[i])
    venta = "${:,.2f}".format(ventas[i])
    print(cliente + " realizó una compra de " + venta)
print(f"\nTotal de ventas del día: ${total_ventas}")

# 3.1 Conversion valores a palabras
def numero_a_palabras_decimal(numero):
```

```
    def numero_a_palabras(n):
        unidades    = ["", "uno", "dos", "tres", "cuatro", "cinco",
                       "seis", "siete", "ocho", "nueve"]
        especiales = ["diez", "once", "doce", "trece", "catorce", "quince",
                      "dieciséis", "diecisiete", "dieciocho", "diecinueve"]
        decenas    = ["", "", "veinte", "treinta", "cuarenta", "cincuenta",
                      "sesenta", "setenta", "ochenta", "noventa"]
        centenas   = ["", "ciento", "doscientos", "trescientos", "cuatrocientos",
                      "quinientos", "seiscientos", "setecientos", "ochocientos", "novecientos"]
        if n == 0:
            return "cero"
        elif n == 100:
            return "cien"
        texto = ""
        if n >= 1000:
            mil = n // 1000
            texto += "mil " if mil == 1 else numero_a_palabras(mil) + " mil "
            n %= 1000
        if n >= 100:
            texto += centenas[n // 100] + " "
            n %= 100
        if 10 <= n < 20:
            texto += especiales[n - 10]
            n = 0
        elif n >= 20:
            texto += decenas[n // 10]
            if n % 10 != 0:
                texto += " y " + unidades[n % 10]
            n = 0
        if 0 < n < 10:
            texto += unidades[n]
        return texto.strip()
    entero           = int(numero)
    centavos         = int(round((numero - entero) * 100))
    palabras_entero  = numero_a_palabras(entero)
    palabras_centavos = numero_a_palabras(centavos)
    return f"{palabras_entero} pesos con {palabras_centavos} centavos"
print()
print("3.1 Conversion simple de valores a palabras")
print()
for i in range(len(clientes)):
cliente = str(clientes[i])
venta = numero_a_palabras_decimal(ventas[i])
print(cliente + " realizó una compra de " + venta)
print(f"\nTotal de ventas del día:" + numero_a_palabras_decimal(total_ventas))
print()
```

```
# 4. Imprimir resultados
print()
print("=== INFORME DE VENTAS ===")
for i in range(len(clientes)):
cliente = str(clientes[i])
venta = numero_a_palabras_decimal(ventas[i])
print(cliente + " realizó una compra de " + venta)
print(f"\nTotal de ventas del día: ${total_ventas}")
print(f"\nTotal de ventas del día:" + numero_a_palabras_decimal(total_ventas))
print()
print(f"Clientes con compras superiores a $2,000: {compras_altas}")
```

3.1 Conversion simple de valores a texto

Juan realizó una compra de $1,500.50
Ana realizó una compra de $2,300.75
Luis realizó una compra de $1,850.00
Carlos realizó una compra de $920.30
Total de ventas del día: $6571.55

3.1 Conversion simple de valores a palabras

Juan realizó una compra de mil quinientos pesos con cincuenta centavos
Ana realizó una compra de dos mil trescientos pesos con setenta y cinco centavos
Luis realizó una compra de mil ochocientos cincuenta pesos con cero centavos
Carlos realizó una compra de novecientos veinte pesos con treinta centavos
Total de ventas del día: seis mil quinientos setenta y uno pesos con cincuenta y cinco centavos

=== INFORME DE VENTAS ===

Juan realizó una compra de mil quinientos pesos con cincuenta centavos
Ana realizó una compra de dos mil trescientos pesos con setenta y cinco centavos
Luis realizó una compra de mil ochocientos cincuenta pesos con cero centavos
Carlos realizó una compra de novecientos veinte pesos con treinta centavos

Total de ventas del día: $6571.55
Total de ventas del día: seis mil quinientos setenta y uno pesos con cincuenta y cinco centavos
Clientes con compras superiores a $2,000: 1

Caso_2: Aplicación de Estructuras Condicionales en Análisis de Datos

La empresa "DataSmart Analytics" se dedica a analizar datos de clientes para mejorar estrategias de marketing. Han recopilado información sobre el número de compras de los clientes en los últimos seis meses y quieren clasificarlos en tres categorías:

- Clientes Premium: Más de 50 compras.
- Clientes Regulares: Entre 20 y 50 compras.
- Clientes Ocasionales: Menos de 20 compras.

El equipo de análisis de datos debe crear un programa en Python que clasifique automáticamente a cada cliente según su número de compras y emita un mensaje personalizado para cada categoría. Adicionalmente, si un cliente tiene exactamente 0 compras, debe recibir un mensaje indicando que se le enviará una promoción especial para incentivar su primera compra.

Preguntas Clave:

1. ¿Qué tipo de estructura condicional es más adecuada para clasificar a los clientes? ¿Por qué?

La mejor opción es usar una estructura if-elif-else. Es ideal porque:

- Permite evaluar varios rangos excluyentes en orden de prioridad.
- Evita múltiples if que podrían evaluarse innecesariamente.
- Mejora la claridad al asociar un bloque único con cada condición.

2. ¿Cómo usarías operadores de comparación y booleanos para definir los rangos de clientes?

Se puede usar operadores como >, <, >=, <=, y == para definir los rangos claramente:

```
if compras == 0:
    mensaje = "Recibirás una promoción especial por ser nuevo cliente."
elif compras > 50:
    mensaje = "¡Gracias por tu fidelidad! Eres un cliente Premium."
elif compras >= 20:
    mensaje = "Eres un cliente Regular. ¡Nos encanta verte seguido!"
else:
    mensaje = "Eres un cliente Ocasional. ¡Esperamos verte más a menudo!"
```

Esto define claramente las condiciones de cada grupo, de forma mutuamente excluyente.

3. ¿Cómo podrías mejorar la legibilidad del código al usar estructuras condicionales?

Se puede aplicar prácticas como:

- Dar nombres descriptivos a las variables, como numero_compras en lugar de solo n.
- Usar funciones: encapsular la lógica en una función como clasificar_cliente(compras).
- Incluir comentarios y evitar anidar condiciones innecesariamente.

Por ejemplo:

```
def clasificar_cliente(nombre, compras):
    if compras == 0:
        return f"{nombre}: promoción especial por tu primera compra."
    elif compras > 50:
        return f"{nombre}: Cliente Premium "
    elif compras >= 20:
        return f"{nombre}: Cliente Regular "
    else:
        return f"{nombre}: Cliente Ocasional "
```

4. ¿Qué ventajas ofrece el uso de sentencias condicionales en este caso respecto a un enfoque manual de clasificación?

- Escalabilidad: se puede procesar miles de clientes automáticamente.
- Precisión: elimina errores humanos al clasificar uno por uno.
- Flexibilidad: se puede modificar las reglas fácilmente (por ejemplo, cambiar el umbral de Premium de 50 a 60).
- Automatización: el sistema se integra fácilmente.

5. ¿Cómo podrías adaptar este código para que funcione con una base de datos en lugar de valores fijos?
Se puede usar archivos CSV o Excel, conectarse a la base de datos,hacer una consulta (SELECT nombre, compras FROM clientes).

- **Utilizar pandas y data frame y archivo CSV**

```
import pandas as pd

# Cargar datos desde archivo CSV
df = pd.read_csv("clientes.csv")
```

- **Utilizar pandas y data frame y archivo CSV**

import pandas as pd y archivo Excel

```
# Cargar datos desde Excel
df = pd.read_excel("clientes.xlsx", sheet_name="Sheet1")
```

- **Sin utilizar pandas y archivo CSV**

```
#Leer el archivo CSV
clientes = []
compras = []
```

```
with open("clientes.csv", "r", encoding="utf-8") as archivo:
    next(archivo)  # Saltar encabezado
    for linea in archivo:
        nombre, cantidad = linea.strip().split(",")
        clientes.append(nombre)
        compras.append(int(cantidad))
```

Cuaderno Google_Colab Ejemplo con resultados

```
# Datos de ejemplo M3_Sesion2_Caso

clientes = ["Juan", "Ana", "Luis", "Carlos", "Sofía"]
compras = [12, 0, 53, 27, 51]

# Función para clasificar y generar mensaje personalizado
def clasificar_cliente(nombre, cantidad_compras):
    if cantidad_compras == 0:
        return f"{nombre}: promoción especial por tu primera compra."
    elif cantidad_compras > 50:
        return f"{nombre}: Cliente Premium. ¡Gracias por tu fidelidad!"
    elif cantidad_compras >= 20:
        return f"{nombre}: Cliente Regular. ¡Nos encanta verte seguido!"
    else:
        return f"{nombre}: Cliente Ocasional. ¡Esperamos verte más seguido!"

# Procesar todos los clientes
for nombre, compras_realizadas in zip(clientes, compras):
    mensaje = clasificar_cliente(nombre, compras_realizadas)
    print(mensaje)
```

```
Juan: Cliente Ocasional. ¡Esperamos verte más seguido!
Ana: promoción especial por tu primera compra.
Luis: Cliente Premium. ¡Gracias por tu fidelidad!
Carlos: Cliente Regular. ¡Nos encanta verte seguido!
Sofía: Cliente Premium. ¡Gracias por tu fidelidad
```

Caso_3: Aplicación de Fundamentos de Python en una Empresa de Retail

La empresa RetailData se enfrenta a un problema en su departamento de análisis de datos. Diariamente, reciben informes con las ventas de cientos de productos y necesitan calcular indicadores clave para la toma de decisiones. Actualmente, este análisis se realiza de manera manual en hojas de cálculo, lo que consume mucho tiempo y puede generar errores. El gerente de datos ha solicitado automatizar este proceso utilizando Python. Se espera que el nuevo sistema realice las siguientes tareas de manera eficiente:

1. Calcular el total de ventas del día.
2. Determinar el producto con la mayor y la menor cantidad de ventas.
3. Calcular el promedio de ventas por producto.
4. Implementar una función que permita obtener estos indicadores de manera sencilla para cualquier conjunto de datos.

El equipo de análisis ha proporcionado el siguiente conjunto de datos de ejemplo:
python
ventas = [150, 200, 50, 400, 300, 120, 80]
productos = ["A", "B", "C", "D", "E", "F", "G"]
Se necesita una solución en Python que use **funciones preconstruidas, funciones personalizadas y módulos estándar** para realizar los cálculos y optimizar el análisis.

Preguntas Clave:
1. ¿Cómo las funciones pueden ayudar a mejorar la eficiencia del análisis de datos en este caso?

Las funciones permiten:
- Reutilizar código sin repetir lógica.
- Organizar tareas complejas en bloques comprensibles y modulares.
- Reducir errores al encapsular comportamientos.
- Facilitar pruebas y mantenimiento, lo que mejora la escalabilidad.

2. ¿Cuáles son las ventajas de utilizar funciones personalizadas en este problema?

Las funciones personalizadas permiten:
- Adaptar la lógica a requisitos específicos del negocio.
- Crear una interfaz clara para usuarios técnicos o no técnicos.
- Permitir la integración con otros sistemas o flujos de trabajo.
- Separar la lógica de presentación de la lógica de análisis.

3. ¿Qué módulo de Python sería útil para calcular el promedio de ventas y por qué?

El módulo statistics, ya que es parte de la biblioteca estándar y ofrece:
- statistics.mean(lista) para calcular promedios de forma directa.
- Mayor legibilidad y precisión que usar sum()/len() manualmente.

4. ¿Cómo se puede modificar la solución para que funcione con distintos conjuntos de datos sin necesidad de cambiar el código principal?

Encapsulando la lógica en una función que reciba listas de productos y ventas como argumentos:

```
def analizar_ventas(productos, ventas):
```

Así se puede utilizar cualquier conjunto de datos sin modificar el código principal.

5. Escribe un código en Python que implemente la solución solicitada.

```
import statistics

def analizar_ventas(productos, ventas):
    total    = sum(ventas)
    promedio = statistics.mean(ventas)

    mayor_ventas  = ventas.index(max(ventas))  # Indice de posicio de mayor venta
    minimo_ventas = ventas.index(min(ventas))  # Indice de posición de menor venta

    mayor_producto = productos[mayor_ventas]
    menor_producto = productos[minimo_ventas]
    print("<<<<< Indicadores del día: >>>>>>")
    print(f"1.- Total de ventas                  : {total}")
    print(f"2.- Promedio por producto       : {promedio:.2f}")
    print(f"3.- Producto con más ventas    : {mayor_producto} ({ventas[mayor_ventas]})")
    print(f"4.- Producto con menos ventas: {menor_producto} ({ventas[minimo_ventas]})")

# Progema principal - datos de ejemplo
productos = ["A", "B", "C", "D", "E", "F", "G"]
ventas = [150, 200, 50, 400, 300, 120, 80]

analizar_ventas(productos, ventas)
```

```
<<<<< Indicadores del día: >>>>>>
 1.- Total de ventas                  1300
 2.- Promedio por producto       : 185.71
 3.- Producto con más ventas     : D (400)
 4.- Producto con menos ventas: C (50
```

Caso_4: Gestión de Clientes en una Empresa de Tecnología

La empresa Tech Solutions se especializa en ofrecer soluciones digitales a clientes de diversas industrias. A medida que su base de clientes ha crecido, la gestión de la información se ha vuelto más compleja. Actualmente, los empleados manejan datos en hojas de cálculo, lo que ha generado dificultades para acceder rápidamente a información relevante y realizar análisis de datos eficaces.

El equipo de desarrollo ha decidido migrar la gestión de clientes a Python utilizando estructuras de datos eficientes. Se presentan los siguientes desafíos:

1. **Almacenamiento de Clientes**: Se necesita una lista con los nombres de los clientes para hacer un seguimiento ordenado de los nuevos ingresos.
2. **Información Detallada de Cada Cliente**: Cada cliente debe tener asociada su información de contacto, incluyendo número de teléfono y dirección de correo electrónico.
3. **Clasificación de Clientes por Ubicación:** La empresa quiere agrupar clientes por ciudad para diseñar estrategias de marketing específicas.
4. **Análisis de Servicios Contratados:** Se requiere un registro de los servicios contratados por cada cliente sin permitir duplicados.

Preguntas Clave

1. ¿Qué estructura de datos en Python recomendarías para almacenar la lista de nombres de los clientes? Explica por qué es la mejor opción.

- Una lista (list) permite mantener el orden de ingreso de los clientes, lo que es ideal para registrar nuevos ingresos cronológicamente.
- Se puede recorrer fácilmente para búsquedas, visualización o exportación.

2. Para asociar la información de contacto de cada cliente a su nombre, ¿qué estructura de datos es más adecuada? Justifica tu respuesta con un ejemplo de código.

- Diccionario para asociar un cliente a su información mediante llaves únicas (nombre) y valores como otro diccionario con contacto.

```
contactos = {
    "Juan Pérez": {
        "teléfono": "+56911111111",
        "email": "juanperez@email.com"
    },
    "Ana Rojas": {
        "teléfono": "+56922222222",
        "email": "anarojas@email.com"
    }
}
```

Esto facilita acceder a datos de un cliente de forma directa:

```
print(contactos["Ana Rojas"]["email"])
```

Salida : anarojas@email.com

3. ¿Cómo organizarías los clientes por ciudad para que la empresa pueda acceder rápidamente a los clientes de una ubicación específica? Proporciona un ejemplo práctico en Python.
- Se pueden agrupar fácilmente múltiples clientes bajo una misma ciudad

```
from collections import defaultdict

clientes_por_ciudad = defaultdict(list)
clientes_por_ciudad["Santiago"].extend(["Juan Pérez", "Ana Rojas"])
clientes_por_ciudad["Valparaíso"].append("Carlos Díaz")
```

Consulta rápida por ubicación:

```
print(clientes_por_ciudad["Santiago"])
```

4. ¿Qué estructura de datos utilizarías para garantizar que un cliente no tenga servicios duplicados en su historial? Explica su utilidad con un fragmento de código.

Los sets garantizan que no se repitan valores.

```
servicios = {
   "Juan Pérez": {"Hosting", "Desarrollo Web"},
   "Ana Rojas": {"SEO", "Análisis de Datos"}
}

# Agregar sin duplicar
servicios["Juan Pérez"].add("Desarrollo Web")   # No se duplicará
```

5. Considerando todas las estructuras de datos mencionadas, ¿cómo las integrarías en un solo sistema eficiente de gestión de clientes?

```
from collections import defaultdict

# Inicialización de estructuras
nombres_clientes = []
contactos = {}
clientes_por_ciudad = defaultdict(list)
servicios = {}

# Función para registrar nuevo cliente
def registrar_cliente(nombre, ciudad, teléfono, email, lista_servicios):
   print(">>>>> Registro de Clientes <<<<<<")
   if nombre not in nombres_clientes:
      nombres_clientes.append(nombre)
      contactos[nombre] = {"teléfono": teléfono, "email": email}
      clientes_por_ciudad[ciudad].append(nombre)
      servicios[nombre] = set(lista_servicios)
```

```
        print(f"Cliente '{nombre}' registrado correctamente.\n")
    else:
        print(f"El cliente '{nombre}' ya existe.\n")

# Función para mostrar resumen
def mostrar_resumen():
    for nombre in nombres_clientes:
        info = contactos[nombre]
        print(f"{nombre} ({info['teléfono']}, {info['email']})")
        print(f"    Servicios: {', '.join(servicios[nombre])}")
        ciudad = next((c for c, lista in clientes_por_ciudad.items() if nombre in lista), "Desconocida")
        print(f"    Ciudad: {ciudad}\n")

def mostrar_clientes_por_ciudad(clientes_por_ciudad):
    print()
    print("Clientes organizados por ciudad:")
    for ciudad, clientes in clientes_por_ciudad.items():
        print(f"{ciudad}:")
        for cliente in clientes:
            print(f"   - {cliente}")
    print()

# Prueba
registrar_cliente("Juan Pérez", "Santiago", "+56911111111", "juanperez@email.com", ["Hosting", "Desarrollo
Web"])
registrar_cliente("Ana Rojas", "Santiago", "+56922222222", "anarojas@email.com", ["SEO", "Análisis de
Datos"])
registrar_cliente("Carlos Díaz", "Valparaíso", "+56933333333", "carlosdiaz@email.com", ["Consultoría"])
registrar_cliente("Juan Pérez", "Santiago", "+56911111111", "juanperez@email.com", ["Hosting", "Desarrollo
Web"])
print(">>>>> Resumen de Clientes <<<<<<")
mostrar_resumen()
print("Listado de Clientes:", nombres_clientes)
mostrar_clientes_por_ciudad(clientes_por_ciudad)
```

#Salida

```
>>>>> Registro de Clientes <<<<<<
Cliente 'Juan Pérez' registrado correctamente.

>>>>> Registro de Clientes <<<<<<
Cliente 'Ana Rojas' registrado correctamente.

>>>>> Registro de Clientes <<<<<<
Cliente 'Carlos Díaz' registrado correctamente.

>>>>> Registro de Clientes <<<<<<
El cliente 'Juan Pérez' ya existe.

>>>>> Resumen de Clientes <<<<<<
Juan Pérez (+56911111111, juanperez@email.com)
    Servicios: Desarrollo Web, Hosting
    Ciudad: Santiago
```

Ana Rojas (+56922222222, anarojas@email.com)
Servicios: Análisis de Datos, SEO
Ciudad: Santiago

Carlos Díaz (+56933333333, carlosdiaz@email.com)
Servicios: Consultoría
Ciudad: Valparaíso

Listado de Clientes: ['Juan Pérez', 'Ana Rojas', 'Carlos Díaz']

Clientes organizados por ciudad:
Santiago:
- Juan Pérez
- Ana Rojas

Valparaíso:
- Carlos Díaz

Caso_5: Automatización del Cálculo de Ventas en una Empresa

La empresa TecnoVentas S.A. se dedica a la comercialización de productos tecnológicos. Cada día, su equipo de ventas registra cientos de transacciones en un sistema, generando una lista de precios de los productos vendidos. Actualmente, el equipo de administración necesita calcular el total de ingresos generados al final de cada jornada, pero este proceso se realiza manualmente, lo que consume demasiado tiempo y es propenso a errores humanos.
El gerente de tecnología ha decidido automatizar este proceso mediante un programa en Python que recorra la lista de precios de los productos vendidos y calcule el total de ventas del día. Además, se requiere generar un informe donde se identifiquen aquellos productos cuyo precio supere un umbral determinado (por ejemplo, productos que cuesten más de 500 dólares). Para ello, es necesario implementar sentencias iterativas que permitan recorrer y procesar la información de manera eficiente.

Preguntas Clave
1. ¿Qué tipo de estructura iterativa (While o For) es más adecuada para calcular el total de ventas? Justifica tu elección con base en el contenido estudiado.

La estructura for es más adecuada aquí porque:
- Se tiene una lista de precios de longitud conocida.
- Se busca iterar secuencialmente sobre cada elemento para acumular el total.
- Es más clara y legible cuando procesamos listas completas.

La estructura while se reserva más para bucles controlados por condición, donde no sabemos cuántas veces debemos repetir.

2. Escribe un fragmento de código en Python que recorra una lista de precios y calcule el total de ventas del día.

```
precios = [120, 300, 650, 820, 450, 150]
total = 0

for precio in precios:
    total += precio

print(f"Total de ventas del día: ${total}")
```

Salida:
Total de ventas del día: $2490

3. ¿Cómo podría usarse la función Range en este escenario? Proporciona un ejemplo.

```
precios = [120, 300, 650, 820, 450, 150]

for i in range(len(precios)):
    print(f"Transacción {i+1}: ${precios[i]}")
```

Salida:

Transacción 1: $120
Transacción 2: $300
Transacción 3: $650
Transacción 4: $820
Transacción 5: $450
Transacción 6: $150

4. Escribe un código en Python que identifique y muestre los productos cuyo precio sea mayor a 500 dólares.

```
productos = ["Monitor", "Teclado", "Notebook", "Smartphone", "Mouse", "Cámara"]
precios = [120, 300, 650, 820, 450, 150]

print("Productos con precio mayor a $500:\n")
for i in range(len(precios)):
    if precios[i] > 500:
        print(f"{productos[i]}: ${precios[i]}")
```

Salida:

Productos con precio mayor a $500:

Notebook: $650
Smartphone: $820

5. Explica cómo la automatización de este proceso puede mejorar la eficiencia en la gestión de datos de la empresa.

Automatizar este proceso:

- Reduce errores humanos por cálculos manuales.
- Ahorra tiempo: resultados en segundos en vez de horas.
- Genera reportes reproducibles y personalizables.
- Permite análisis en tiempo real para tomar decisiones mejor informadas.
- Escalable: puede adaptarse para miles de registros sin perder rendimiento

Caso_6: Aplicación de la Programación Orientada a Objetos en un Entorno Laboral

La empresa TechData Solutions se especializa en la gestión y análisis de datos financieros. Actualmente, los analistas de la empresa registran manualmente la información de clientes y sus transacciones, lo que ha generado errores y retrasos en los reportes financieros.
Para mejorar este proceso, la gerencia ha decidido desarrollar un sistema que gestione la información de los clientes y permita registrar sus transacciones de manera eficiente. El equipo de desarrollo ha propuesto utilizar Programación Orientada a Objetos para estructurar la solución de forma modular y escalable.
Se requiere diseñar una clase **Cliente** con los siguientes atributos y métodos:

- **Atributos:** nombre, email, saldo.
- **Métodos**: depositar(monto), retirar(monto), mostrar_saldo().

También se necesita una clase **Transaccion** que registre cada operación realizada por un cliente.

- **Atributos**: tipo (depósito o retiro), monto, fecha.

La solución debe garantizar que los clientes puedan realizar depósitos y retiros, y que cada transacción quede registrada correctamente.

Preguntas Clave

1. ¿Cuál es la importancia de utilizar la Programación Orientada a Objetos en este escenario?

La POO permite estructurar el sistema de forma modular, mantenible y escalable, facilitando:

- Agrupación lógica de datos y operaciones (clientes y sus transacciones).
- Reutilización de código a través de clases reutilizables.
- Encapsulamiento para proteger y controlar el acceso a los datos.
- Extensibilidad: puedes añadir nuevos comportamientos (intereses, validaciones, etc.) sin modificar lo existente.

2. ¿Cuál es la diferencia entre una clase y un objeto en este caso de estudio?

- Clase: es el molde o definición. Por ejemplo, la clase Cliente define qué atributos y métodos debe tener todo cliente.
- Objeto: es una instancia específica de esa clase. Por ejemplo, cliente1 = Cliente(...) crea un cliente real a partir del molde.

3. ¿Cómo se aplicaría el principio de encapsulamiento en la clase Cliente?

El encapsulamiento se logra:
- Declarando atributos privados o protegidos (self._saldo).
- Usando métodos públicos como depositar() o retirar() para controlar cómo se accede o modifica el saldo.
- Esto evita modificaciones directas no deseadas al estado interno del cliente.

4. ¿Cómo podría utilizarse la herencia para mejorar este sistema en futuras versiones?

Se puede crear subclases para extender comportamientos:
- ClienteVIP(Cliente) con límites mayores o servicios preferentes.
- ClienteEmpresa(Cliente) con múltiples representantes o cuentas compartidas.
- Esto permite especialización sin reescribir la lógica base.

5. Implementa en Python la clase Cliente y la clase Transaccion, asegurando que se registren correctamente las operaciones realizadas.

```
from datetime import datetime

class Transaccion:
    def __init__(self, tipo, monto):
        self.tipo = tipo
        self.monto = monto
        self.fecha = datetime.now()

    def __str__(self):
            return  f"{self.fecha.strftime('%Y-%m-%d  %H:%M:%S')}  |  {self.tipo.upper()}  |
${self.monto}"

class Cliente:
    def __init__(self, nombre, email):
        self.nombre = nombre
        self.email = email
        self._saldo = 0
        self.transacciones = []

    def depositar(self, monto):
        if monto > 0:
            self._saldo += monto
            self.transacciones.append(Transaccion("depósito", monto))
            print(f"Depósito de ${monto} realizado.")
        else:
            print("Monto inválido para depósito.")

    def retirar(self, monto):
        if 0 < monto <= self._saldo:
            self._saldo -= monto
            self.transacciones.append(Transaccion("retiro", monto))
            print(f"Retiro de ${monto} realizado.")
        else:
            print("Fondos insuficientes o monto inválido.")
```

```
    def mostrar_saldo(self):
        print(f"Saldo actual de {self.nombre}: ${self._saldo}")

    def historial_transacciones(self):
        print(f"Historial de transacciones para {self.nombre}:")
        for trans in self.transacciones:
            print("  ", trans)

# Ejemplo
cliente1 = Cliente("Mauricio Tapia", "mauricio@email.com")
cliente1.depositar(1000)
cliente1.retirar(300)
cliente1.mostrar_saldo()
cliente1.historial_transacciones()

#Salida

Depósito de $1000 realizado.
Retiro de $300 realizado.
Saldo actual de Mauricio Tapia: $700
Historial de transacciones para Mauricio Tapia:
  2025-07-03 21:15:52 | DEPÓSITO | $1000
  2025-07-03 21:15:52 | RETIRO | $300
```