

MATERIA: PROGRAMACIÓN II



ACTIVIDAD 02

ING: JIM REQUENA

ESTUDIANTE: MARCO AURELIO BARRIGA

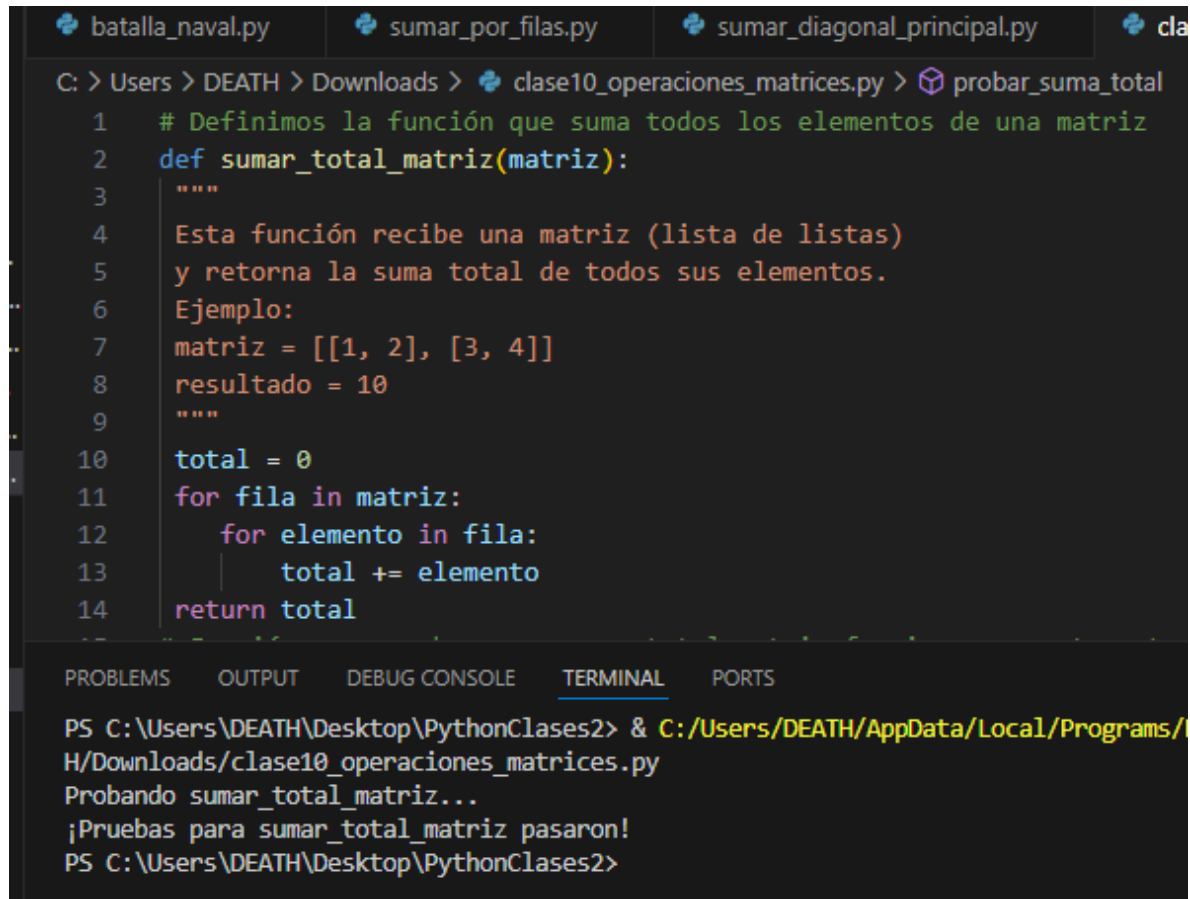
BOLIVIA - SANTA CRUZ

2025

ACTIVIDAD #2- PROG II

Clase 10: Operaciones Básicas con Matrices

sumar_total_matriz



```
batalla_naval.py  sumar_por_filas.py  sumar_diagonal_principal.py  cla
C: > Users > DEATH > Downloads > clase10_operaciones_matrices.py > probar_suma_total
1  # Definimos la función que suma todos los elementos de una matriz
2  def sumar_total_matriz(matriz):
3      """
4      Esta función recibe una matriz (lista de listas)
5      y retorna la suma total de todos sus elementos.
6      Ejemplo:
7      matriz = [[1, 2], [3, 4]]
8      resultado = 10
9      """
10     total = 0
11     for fila in matriz:
12         for elemento in fila:
13             total += elemento
14     return total

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS C:\Users\DEATH\Desktop\PythonClases2> & C:/Users/DEATH/AppData/Local/Programs/Python/Python39-64/Python.exe C:/Users/DEATH/Desktop/PythonClases2/Downloads/clase10_operaciones_matrices.py
Probando sumar_total_matriz...
¡Pruebas para sumar_total_matriz pasaron!
PS C:\Users\DEATH\Desktop\PythonClases2>
```

Este código en Python está diseñado para resolver una tarea común en programación, calcular la suma de todos los elementos dentro de una matriz. La función principal, llamada `sumar_total_matriz`, recibe como parámetro una matriz (es decir, una lista que contiene otras listas). El objetivo es recorrer todos los elementos que contiene, sin importar cuántas filas o columnas tenga, y sumarlos en una variable acumuladora llamada `total`.

sumar_por_filas

```
C: > Users > DEATH > Downloads > sumar_por_filas.py > ...
1  # Definimos la función que suma los elementos por cada fila de la matriz
2  def sumar_por_filas(matriz):
3      """
4      Esta función recibe una matriz (lista de listas)
5      y devuelve una lista con la suma de cada fila.
6      Ejemplo:
7      matriz = [[1, 2, 3], [4, 5, 6]]
8      resultado = [6, 15]
9      """
10     resultado = []
11     for fila in matriz:
12         suma_fila = sum(fila) # Suma todos los elementos de la fila
13         resultado.append(suma_fila)
14     return resultado

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\DEATH\Desktop\PythonClases2> & C:/Users/DEATH/AppData/Local/Programs/Python/
H/Downloads/sumar_por_filas.py

Probando sumar_por_filas...
¡Pruebas para sumar_por_filas pasaron!
PS C:\Users\DEATH\Desktop\PythonClases2>
```

Este código define una función llamada `sumar_por_filas`, cuyo propósito es calcular la suma de los elementos de cada fila dentro de una **matriz**, la función devuelve una nueva lista que contiene la suma de cada fila por separado. Por ejemplo, si la matriz es `[[1, 2, 3], [4, 5, 6]]`, el resultado será `[6, 15]`, ya que se suman los elementos fila por fila. incluye una función de prueba llamada `probar_suma_por_filas`, que verifica si la función principal funciona correctamente en distintos casos, incluyendo matrices comunes, con datos repetidos y el caso especial de una matriz vacía. Usar funciones como esta permite validar automáticamente que el código se comporta como se espera.

sumar_diagonal_principal

```
C: > Users > DEATH > Downloads > sumar_diagonal_principal.py
1  # Definimos la función que suma los elementos de la diagonal principal
2  def sumar_diagonal_principal(matriz):
3      """
4      Esta función recibe una matriz cuadrada (n x n) y retorna la suma de los elementos en su diagonal principal.
5      Ejemplo:
6      matriz = [[1, 2],
7               [3, 4]]
8      diagonal principal: 1 y 4 → suma = 5
9      """
10
11     suma = 0
12     for i in range(len(matriz)):
13         suma += matriz[i][i] # Accede al elemento en la diagonal principal
14     return suma

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\DEATH\Desktop\PythonClases2> & C:/Users/DEATH/Desktop/PythonClases2/Downloads/sumar_diagonal_principal.py

Probando sumar_diagonal_principal...
¡Pruebas para sumar_diagonal_principal pasaron!
PS C:\Users\DEATH\Desktop\PythonClases2>
```

Tiene como propósito sumar los elementos ubicados en la **diagonal principal** de una matriz cuadrada. La función utiliza un bucle for que recorre los índices de la matriz y suma los valores que se encuentran en esas posiciones diagonales. Además, se incluye una función de prueba llamada probar_suma_diagonal_principal que verifica si la función principal funciona correctamente en distintos escenarios: matrices de 3x3, 2x2 y el caso más simple de una matriz 1x1.

Teclado numérico

```
H/Downloads/matriz_teclado.py
```

```
▸ MATRIZ DEL TECLADO:
```

```
1      2      3
4      5      6
7      8      9
*      0      #
```

```
▸ MATRIZ 5x5 CON CEROS (usando bucles):
```

```
0      0      0      0      0
0      0      0      0      0
0      0      0      0      0
0      0      0      0      0
0      0      0      0      0
```

```
▸ MATRIZ 5x5 CON CEROS (usando comprensión de listas):
```

```
0      0      0      0      0
0      0      0      0      0
0      0      0      0      0
0      0      0      0      0
0      0      0      0      0
```

```
EXPLICACIÓN:
```

```
matriz_compression = [[0 for j in range(5)] for i in range(5)]
```

```
↳ Parte interna: [0 for j in range(5)] → crea una fila con cinco ceros
```

```
↳ Parte externa: for i in range(5) → repite esa fila cinco veces
```

```
Resultado: Una matriz de 5x5 completamente llena de ceros.
```

Se presentan ejemplos claros y prácticos como la representación de un teclado numérico, la construcción de una matriz 5x5 llena de ceros usando bucles, enseña cómo imprimir matrices de forma ordenada, lo que ayuda a visualizar mejor los datos. Gracias a sus explicaciones paso a paso, Su propósito no es solo mostrar cómo construir matrices, sino también cómo entender su lógica interna, lo cual permite aplicar estos conceptos en proyectos más grandes como simulaciones, tableros interactivos o análisis de datos.

Clase 11 transformaciones

Matriz simétrica

```
1 def es_simetrica(matriz):
2     # Requisito 1: Debe ser cuadrada
3     num_filas = len(matriz)
4     if num_filas == 0:
5         return True # Una matriz vacía es trivialmente simétrica
6
7     for i in range(num_filas):
8         if len(matriz[i]) != num_filas:
9             return False # No es cuadrada
10
11     # Requisito 2: Comparar matriz[i][j] con matriz[j][i]
12     for i in range(num_filas):
13         for j in range(i + 1, num_filas): # Solo la mitad superior
14             if matriz[i][j] != matriz[j][i]:
15                 return False # ¡Con una diferencia no es simétrica!
16
17     return True # Si nunca encontramos diferencias, es simétrica
18
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\DEATH\Desktop\PythonClases2> & C:/Users/DEATH/Desktop/PythonClases2/es_simetrica.py
¡Pruebas para es_simetrica pasaron! ✓
PS C:\Users\DEATH\Desktop\PythonClases2>
```

Este código en Python define una función llamada `es_simetrica`, cuyo propósito es determinar si una matriz cuadrada es simétrica respecto a su diagonal principal. Una matriz es simétrica cuando el valor en la posición $[i][j]$ es igual al valor en $[j][i]$, es decir, cuando es un reflejo de sí misma en la diagonal. La función primero comprueba que la matriz sea cuadrada, es decir, que tenga el mismo número de filas que de columnas, lo cual es una condición necesaria para que pueda ser simétrica, compara cada par de elementos opuestos en la diagonal sin repetir comprobaciones innecesarias, lo cual la hace eficiente. El código incluye también varios casos de prueba para validar que la función funciona correctamente en distintos escenarios (simétrica, no simétrica y no cuadrada).

Matriz identidad

```
1  def es_identidad(matriz):
2      # Requisito 1: Debe ser cuadrada
3      num_filas = len(matriz)
4      if num_filas == 0:
5          return True # Una matriz vacía
6
7      for i in range(num_filas):
8          if len(matriz[i]) != num_filas:
9              return False # No es cuadrada
10
11     # Requisito 2: Verificar la diagonal
12     for i in range(num_filas):
13         for j in range(num_filas):
14             if i == j:
15                 if matriz[i][j] != 1:
16                     return False # La d
17             else:
18                 if matriz[i][j] != 0:
19                     return False # Elem
20
21     return True # Cumple con todas las
22
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORT

```
PS C:\Users\DEATH\Desktop\PythonClases2> & C:/Users
es_identidad.py
¡Pruebas para es_identidad pasaron! ✓
PS C:\Users\DEATH\Desktop\PythonClases2>
```

Este código en Python evalúa si una matriz es una matriz identidad. La función primero verifica si la matriz es cuadrada, condición necesaria para ser identidad. Luego recorre todos los elementos: si está en la diagonal ($i == j$), debe ser igual a 1; si está fuera de la diagonal, debe ser 0. Si encuentra alguna excepción a esta regla, devuelve False. Finalmente, se incluyen varios **casos de prueba** que validan el funcionamiento correcto en distintas situaciones, incluyendo una matriz identidad válida, una con errores en la diagonal y una que ni siquiera es cuadrada.

Matriz traspuesta

```
1  def transponer_matriz(matriz):
2      if not matriz or not matriz[0]:
3          return []
4
5      num_filas = len(matriz)
6      num_columnas = len(matriz[0])
7
8      # Inicializamos la traspuesta con la estructura
9      matriz_transpuesta = []
10
11     for j in range(num_columnas): # Itera sobre las columnas
12         nueva_fila = []
13         for i in range(num_filas): # Itera sobre las filas
14             nueva_fila.append(matriz[i][j])
15         matriz_transpuesta.append(nueva_fila)
16
17     return matriz_transpuesta
18
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\DEATH\Desktop\PythonClases2> & C:/Users/DEATH/Desktop/PythonClases2/transpuesta.py
¡Prueba 1 (2x3) pasada! ✓
PS C:\Users\DEATH\Desktop\PythonClases2>
```

La función está diseñada para manejar tanto matrices cuadradas como no cuadradas, e incluso casos especiales, como una matriz vacía o con una sola fila/columna, usa dos bucles anidados: el externo recorre las columnas originales, y el interno recorre las filas, tomando los elementos en orden invertido para crear las nuevas filas de la matriz traspuesta. La transposición convierte una matriz de dimensión $m \times n$ en otra de dimensión $n \times m$, el código también incluye una prueba automatizada usando `assert`, lo que ayuda a confirmar que el resultado es correcto.

Batalla Naval

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL F
1 2 3 4 5
A 0 0 0 0 0
B 0 0 0 0 0
C 0 0 0 0 0
D 0 0 0 0 0
E 0 0 0 0 0
Ingresa coordenada para disparar (ej. A3): d3
Agua...
La CPU dispara a E1
Agua...

--- Turno 2 ---
Tu tablero:
1 2 3 4 5
A 0 0 0 0 0
B 0 0 0 1 0
C 0 0 0 0 0
D 1 0 1 0 0
E * 0 0 0 0
Tus disparos:
1 2 3 4 5
A 0 0 0 0 0
B 0 0 0 0 0
C 0 0 0 0 0
D 0 0 * 0 0
E 0 0 0 0 0
Ingresa coordenada para disparar (ej. A3): a1
Agua...
La CPU dispara a A1
Agua...
```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
C 0 0 0 * 0
D 1 0 1 * *
E * 0 * 0 *
Tus disparos:
1 2 3 4 5
A * * 0 0 0
B 0 0 * 0 *
C 0 0 0 0 *
D 0 0 * 0 *
E * 0 0 0 0
Ingresa coordenada para disparar (ej. A3): e3
Agua...
La CPU dispara a C5
Agua...

--- Turno 10 ---
Tu tablero:
1 2 3 4 5
A * 0 0 * 0
B 0 0 0 1 0
C 0 0 0 * *
D 1 0 1 * *
E * 0 * 0 *
Tus disparos:
1 2 3 4 5
A * * 0 0 0
B 0 0 * 0 *
C 0 0 0 0 *
D 0 0 * 0 *
E * 0 * 0 0
Ingresa coordenada para disparar (ej. A3):
```

En Python representa un **minijuego por turnos** donde se colocan barcos aleatoriamente en un tablero de 5x5, y el jugador y la CPU se turnan para disparar tratando de hundir los barcos enemigos. La matriz se usa para simular el tablero, donde cada celda puede contener agua (0), un barco (1), un impacto (2) o un disparo fallido (3). Se utilizan funciones bien estructuradas para mostrar el tablero, traducir coordenadas (como "B2"), colocar barcos, disparar, y verificar si aún quedan barcos flotando. La CPU ataca con coordenadas aleatorias válidas.

Clase 12 Registros de diccionario

Producto

```
C: > Users > DEATH > Downloads > clase12_diccionarios.py > ...
1  # 1. Crear el diccionario 'producto'
2  producto = {
3      "codigo": "P001",
4      "nombre": "Chocolate para Taza 'El Ceibo'",
5      "precio_unitario": 15.50,
6      "stock": 50,
7      "proveedor": "El Ceibo Ltda."
8  }

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\DEATH\Desktop\PythonClases2> & C:/Users/DEATH/Ag
ionarios.py
Producto: Chocolate para Taza 'El Ceibo'
Precio unitario: Bs. 15.5

Estado final del producto:
codigo: P001
nombre: Chocolate para Taza 'El Ceibo'
precio_unitario: 15.5
stock: 45
proveedor: El Ceibo Ltda.
en_oferta: True
PS C:\Users\DEATH\Desktop\PythonClases2>
```

Utiliza un **diccionario** para representar un producto con varias características como código, nombre, precio unitario, stock y proveedor. Los diccionarios son estructuras de datos muy útiles para almacenar información relacionada en pares clave-valor, aquí se muestra cómo acceder a valores específicos para imprimirlos, modificar el stock simulando una venta y agregar un nuevo atributo que indica si el producto está en oferta. El uso de bucles para imprimir el diccionario completo ayuda a mostrar de forma ordenada toda la información disponible.

Lista de inventario

```
C: > Users > DEATH > Downloads > clase12_diccionarios2.py > ...
1  # 1. Crear una lista vacía llamada inventario
2  inventario = []
3
4  # 2. Crear al menos tres diccionarios de productos
5  producto1 = {
6      "nombre": "Chocolate para Taza 'El Ceibo'",
7      "stock": 50
8  }
9
10 producto2 = {
11     "nombre": "Café de los Yungas",
12     "stock": 100
13 }
14
15 producto3 = {
16     "nombre": "Quinua Real en Grano",
17     "stock": 80
18 }
19
20 inventario.append(producto1)
21 inventario.append(producto2)
22 inventario.append(producto3)
23
24 # 3. Mostrar el inventario
25
26 # 3.1. Cantidad de productos en inventario
27 cantidad = len(inventario)
28 print(f"Cantidad de productos en inventario: {cantidad}")
29
30 # 3.2. Resumen visual del inventario
31 print("\n--- Inventario Actual ---")
32 for producto in inventario:
33     print(f"- {producto['nombre']}: {producto['stock']} unidades en stock.")
34
35 PS C:\Users\DEATH\Desktop\PythonClases2> & C:/Users/DEATH/Desktop/PythonClases2/clase12_diccionarios2.py
Cantidad de productos en inventario: 3
--- Inventario Actual ---
- Chocolate para Taza 'El Ceibo': 50 unidades en stock.
- Café de los Yungas: 100 unidades en stock.
- Quinua Real en Grano: 80 unidades en stock.
PS C:\Users\DEATH\Desktop\PythonClases2>
```

La utilidad de este enfoque es que permite **gestionar múltiples productos de forma escalable**. Es posible acceder, modificar o expandir la información de cada producto de forma individual. Además, al recorrer la lista con un bucle for, se genera un resumen visual claro del estado del inventario. Cada producto se modela como un diccionario con atributos clave como "nombre" y "stock", lo que permite almacenar información específica de manera clara y organizada. Luego, estos diccionarios se agregan a una lista llamada inventario, que actúa como una colección general de productos.

Uso de keys

```
PS C:\Users\DEATH\Desktop\PythonClases2> & C:/Us
ionarios3.py
Claves del diccionario producto:
→ codigo
→ nombre
→ precio_unitario
→ stock
→ proveedor

Valores del diccionario producto:
→ P001
→ Chocolate para Taza 'El Ceibo'
→ 15.5
→ 50
→ El Ceibo Ltda.

Contenido completo del diccionario producto:
codigo: P001
nombre: Chocolate para Taza 'El Ceibo'
precio_unitario: 15.5
stock: 50
proveedor: El Ceibo Ltda.

❌ La clave 'en_oferta' no existe.
Stock disponible: 50 unidades

--- Detalle de productos usando .items() ---
nombre → Chocolate para Taza 'El Ceibo'
stock → 50
---
nombre → Café de los Yungas
stock → 100
---
nombre → Quinoa Real en Grano
stock → 80
```

```
---
🎵 Canción:
titulo: Bohemian Rhapsody
artista: Queen
album: A Night at the Opera
duracion_segundos: 354
genero: Rock Progresivo
es_explicita: False
reproducciones: 275000000
```

```
🚗 Coche:
marca: Toyota
modelo: Corolla Cross
año: 2023
color: Gris Metálico
placa: 5923-LLT
kilometraje: 17450.6
en_venta: True
```

```
📱 Post en Red Social:
id_post: POST-20250622-001
autor: jimmy.requena
contenido_texto: ¡Hoy lanzamos nuestro nuevo ERP con IA! 🚀
lista_de_likes: ['ana_123', 'dev.mario', 'claudia77']
fecha_publicacion: 2025-06-22
es_publico: True
hashtags: ['#IA', '#ERP', '#ProductLaunch']
```

```
📱 Post en Red Social:
id_post: POST-20250622-001
autor: Marco Aurelio
contenido_texto: ¡Hoy lanzamos nuestro nuevo ERP con IA! 🚀
lista_de_likes: ['ana_123', 'dev.mario', 'claudia77']
fecha_publicacion: 2025-06-22
es_publico: True
hashtags: ['#IA', '#ERP', '#ProductLaunch']
```

Se hace uso de funciones útiles como. `keys()`, `values()` y `items()` para acceder a los datos, y se incluye verificación de existencia de claves usando `in`. Cada bloque de código demuestra cómo los diccionarios permiten representar objetos complejos como productos, canciones, autos o publicaciones en redes sociales mediante pares clave-valor. Este tipo de estructura es ampliamente utilizada en bases de datos, APIs, almacenamiento en archivos JSON y desarrollo web

Todo_list

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

==== MENÚ TO-DO LIST ====
1. Agregar nueva tarea
2. Mostrar todas las tareas
3. Marcar tarea como completada
4. Eliminar tarea
0. Salir
Elige una opción: 1
Descripción de la nueva tarea: Realizar la 2da actividad
Prioridad (alta, media, baja): media
✅ Tarea 'Realizar la 2da actividad' añadida con éxito.

==== MENÚ TO-DO LIST ====
1. Agregar nueva tarea
2. Mostrar todas las tareas
3. Marcar tarea como completada
4. Eliminar tarea
0. Salir
Elige una opción: 4
ID de la tarea a eliminar: Realizar la 2da actividad
Traceback (most recent call last):
  File "c:\Users\DEATH\Downloads\clase13_todolist.py", line 77, in <module>
    id_t = int(input("ID de la tarea a eliminar: "))
ValueError: invalid literal for int() with base 10: 'Realizar la 2da actividad'
PS C:\Users\DEATH\Desktop\PythonClases2> |
```

Este código en Python implementa una **aplicación de lista de tareas (to-do list)** completamente funcional desde la consola. Usa una lista de diccionarios para representar cada tarea, y permite al usuario **agregar, ver, marcar como completada y eliminar tareas** mediante un menú interactivo. Cada tarea tiene un id único, una descripción, un estado de completado y una prioridad (alta, media o baja). Se definen funciones específicas para cada acción (como `agregar_tarea`, `mostrar_tareas`, `buscar_tarea_por_id`, etc.), lo cual facilita la lectura y futura ampliación del programa.

Sala de cine

```
🎬 Estado actual de la sala de cine:
  0 1 2 3 4 5 6 7
0 L L L L L L L L
1 L L L L L L L L
2 L L L L L L L L
3 L L L L L L L L
4 L L L L L L L L
Asientos libres: 40

Menú:
1. Ocupar asiento
0. Salir
Elige una opción: 1
Ingresa el número de fila: 2
Ingresa el número de columna: 4
Asiento (2, 4) ocupado exitosamente.

🎬 Estado actual de la sala de cine:
  0 1 2 3 4 5 6 7
0 L L L L L L L L
1 L L L L L L L L
2 L L L L O L L L
3 L L L L L L L L
4 L L L L L L L L
Asientos libres: 39

Menú:
1. Ocupar asiento
0. Salir
Elige una opción: 0
👋 Gracias por usar el sistema de reserva. ¡Hasta luego!
PS C:\Users\DEATH\Desktop\PythonClases2> |
```

Implementa un sistema básico de reserva de asientos para una sala de cine, utilizando matrices (listas de listas) para representar la disposición de los asientos. Cada asiento puede estar libre ('L') u ocupado ('O'), y el usuario puede visualizarlos en pantalla, elegir uno por coordenadas (fila, columna), y marcarlo como ocupado. El programa principal, main, muestra un menú simple e interactivo que se repite hasta que el usuario decide salir. crear_sala genera la matriz de asientos, mostrar_sala imprime la sala visualmente, incluyendo índices de filas y columnas, ocupar_asiento valida la elección del usuario y actualiza el estado del asiento, contar_asientos_libres resume cuántos lugares siguen disponible