Colección de Ejercicios de Python: Manejo de Listas de Listas

A continuación, se presenta una colección de 9 ejercicios diseñados para practicar la manipulación de estructuras de datos anidadas en Python, centrándose exclusivamente en listas, tuplas y tipos de datos básicos.

Ejercicio 1

Cree una función, aplanar(matriz), que reciba una matriz (lista de listas) de números, y la "aplane" convirtiéndola en una lista de una sola dimensión que tendrá todos los elementos de la matriz, en el orden en que aparecen en un recorrido de izquierda a derecha y de arriba abajo si son superiores un umbral pasado por parámetro.

Ejemplo de Uso:

```
# --- Datos de entrada ---
matriz_numeros = [
    [10, 5, 8],
    [25, 2, 16],
    [1, 12]
]
umbral = 9

# --- Implementación del código solicitado ---
# TO DO

# --- Llamada y muestra del resultado ---
resultado = aplanar (matriz_numeros)
print(f"Matriz original: {matriz_numeros}")
print(f"Números mayores que {umbral}: {resultado}")

# Resultado esperado:
# [10, 25, 16, 12]
```

Ejercicio 2

Objetivo: Escribir una función que modifique una lista que representa el estado de varios objetos en un plano 2D. Cada objeto tiene un identificador y una lista de posiciones (coordenadas). La función debe añadir una nueva coordenada al final de la lista de un objeto específico. Si no se encuentra ningún objeto con el identificador de objeto proporcionado, la función no debe hacer nada.

La Función a Implementar:

actualizar_posicion_objeto(estado_objetos, id_objeto, nueva_coordenada)

Parámetros:

estado_objetos: Una lista de listas. Cada lista interna representa un objeto y contiene: En la primera posición, el identificador del objeto (una string). En las posiciones siguientes, tuplas (x,

y) que representan su historial de coordenadas. Formato: [[id_1, (x1, y1), (x2, y2)], [id_2, (x3, y3)], ...]

id_objeto: Una string con el identificador del objeto a modificar.

nueva_coordenada: Una tupla (x, y) que se debe añadir al historial del objeto.

Ejemplo de Uso:

```
# --- Datos de entrada ---
# Formato: [id_objeto, (coord1), (coord2), ...]
objetos_en_mapa = [
    ["Player1", (10, 20), (12, 25)],
    ["EnemyA", (50, 50)],
    ["ItemBox", (100, 100)]
]

# --- Implementación del código solicitado ---
# TO DO

# --- Llamada y muestra del resultado ---
print("Estado antes de la actualización:")
print(objetos_en_mapa)

actualizar_posicion_objeto(objetos_en_mapa, "Player1", (15, 30))
actualizar_posicion_objeto(objetos_en_mapa, "EnemyB", (0, 0))

print("\nEstado después de la actualización:")
print(objetos_en_mapa)

# Resultado esperado:
# Estado antes de la actualización:
#  [['Player1', (10, 20), (12, 25)], ['EnemyA', (50, 50)],
['ItemBox', (100, 100)]]
#
# Estado después de la actualización:
# [['Player1', (10, 20), (12, 25), (15, 30)], ['EnemyA', (50,
50)], ['ItemBox', (100, 100)]]
```

Ejercicio 3:

Objetivo: Crear una función que procese una lista de calificaciones por asignatura y genere un resumen por alumno. Este resumen debe incluir el nombre del alumno, su nota media, y su nota más alta. Las notas se deben redondear a dos decimales.

La Función a Implementar:

generar_resumen_alumnos(calificaciones_por_asignatura)

Parámetros:

calificaciones_por_asignatura: Una lista de listas. Cada lista interna representa una asignatura y contiene: El nombre de la asignatura (string) seguido de una o más tuplas, donde cada tupla es (nombre_alumno, nota). Formato: [[asignatura_1, (alumno_A, 7.5), (alumno_B, 8.0)], [asignatura_2, (alumno_A, 9.0)], ...]

Valor a devolver: Una lista de tuplas. Cada tupla representa el resumen de un alumno y sigue el formato: (nombre_alumno, nota_media, nota_mas_alta) La lista debe estar ordenada alfabéticamente por el nombre del alumno.

Ejemplo de Uso:

```
# --- Datos de entrada ---
calificaciones = [
    ["Matemáticas", ("Ana", 8.5), ("Juan", 7.0), ("Eva", 9.5)],
    ["Historia", ("Juan", 6.5), ("Ana", 9.0)],
    ["Ciencias", ("Eva", 8.0), ("Ana", 8.5), ("Juan", 8.0)]
]

# --- Implementación del código solicitado ---
# TO DO

# --- Llamada y muestra del resultado ---
resumen = generar_resumen_alumnos(calificaciones)
print("Resumen de calificaciones:")
for nombre, media, maxima in resumen:
    print(nombre, media, máxima, sep=", ")

# Resultado esperado:
# Resumen de calificaciones:
# Ana, 8.67, 9.0
# Eva, 8.75, 9.5
# Juan, 7.17, 8.0
```

Ejercicio 4

Objetivo: Crear una función que determine si una ruta, definida por una secuencia de coordenadas, es válida dentro de un mapa. El mapa es una matriz donde ciertos valores representan obstáculos infranqueables.

La Función a Implementar:

validar_ruta_en_mapa(mapa, ruta)

Parámetros:

mapa: Una lista de listas de números. 0 representa un camino libre y 1 representa un obstáculo. ruta: Una lista de tuplas, donde cada tupla es una coordenada (fila, columna) que representa un paso en la ruta.

¿Qué debe hacer la función? 1. Verificar que todas las coordenadas de la ruta están dentro de los límites del mapa. 2. Comprobar que ninguna coordenada de la ruta coincide con la posición de un obstáculo (1) en el mapa. 3. Si la ruta es válida (cumple ambas condiciones), la función debe devolver True. Si alguna coordenada está fuera de los límites o sobre un obstáculo, debe devolver False inmediatamente.

Valor a devolver: Un valor booleano (True o False).

Ejemplo de Uso:

```
# --- Datos de entrada ---
```

```
mapa_juego = [
    [0, 0, 1, 0],
    [0, 0, 0, 0],
    [1, 0, 1, 0],
    [0, 0, 0, 0]
]

ruta_1 = [(0, 0), (1, 0), (1, 1), (2, 1), (3, 1)]
ruta_2 = [(0, 0), (0, 1), (0, 2)] # Tropieza con un obstáculo
ruta_3 = [(2, 3), (3, 3), (4, 3)] # Se sale del mapa

# --- Implementación del código solicitado ---
# TO DO

# --- Llamada y muestra del resultado ---
print(
   "¿Ruta 1 es válida?: ",
   validar_ruta_en_mapa(mapa_juego, ruta_1)}
)
print(
   "¿Ruta 2 es válida?: ",
   validar_ruta_en_mapa(mapa_juego, ruta_2)
)
print(
   "¿Ruta 3 es válida?: ",
   validar_ruta_en_mapa(mapa_juego, ruta_3)
)

# Resultado esperado:
# ¿Ruta 1 es válida?: True
# ¿Ruta 2 es válida?: False
# ¿Ruta 3 es válida?: False
```

Ejercicio 5

Objetivo: Implementar una función que transponga una matriz (lista de listas). La transposición consiste en cambiar las filas por columnas. Se debe asumir que la matriz de entrada no es vacía y es rectangular (todas las filas tienen la misma longitud).

La Función a Implementar:

transponer_matriz(matriz)

Parámetros:

matriz: Una lista de listas de cualquier tipo de dato básico (números, cadenas, etc.).

Valor a devolver: Una nueva lista de listas que es la transposición de la matriz de entrada.

Ejemplo de Uso:

```
# --- Datos de entrada ---
matriz_números = [
    [1, 2, 3],
    [4, 5, 6]
]
```

```
matriz_strings = [
    ['A', 'B'],
    ['C', 'D'],
    ['E', 'F']
]

# --- Implementación del código solicitado ---
# TO DO

# --- Llamada y muestra del resultado ---
matriz_transpuesta_1 = transponer_matriz(matriz_números)
matriz_transpuesta_2 = transponer_matriz(matriz_strings)

print("Matriz original 1:")
for fila in matriz_números:
    print(fila)
print("\nMatriz transpuesta 1:")
for fila in matriz_transpuesta_1:
    print(fila)

print("\nMatriz original 2:")
for fila in matriz_strings:
    print(fila)
print("\nMatriz transpuesta 2:")
for fila in matriz_transpuesta_2:
    print(fila)


# Resultado esperado:
# Matriz original 1:
# [1, 2, 3]
# [4, 5, 6]
#
# Matriz transpuesta 1:
# [1, 4]
# [2, 5]
# [3, 6]
#
# Matriz original 2:
# ['A', 'B']
# ['C', 'D']
# ['E', 'F']
#
# Matriz transpuesta 2:
# ['A', 'C', 'E']
# ['B', 'D', 'F']
```

Ejercicio 6

Objetivo: Escribir una función que analice un lienzo (una matriz de caracteres) y encuentre la "caja delimitadora" de un símbolo específico. La caja delimitadora se define por las coordenadas superior-izquierda e inferior-derecha que encierran todas las apariciones del símbolo. Si el símbolo no se encuentra nunca, la función debe devolver None.

La Función a Implementar:

encontrar_caja_delimitadora(lienzo, simbolo)

Parámetros:

lienzo: Una lista de listas de caracteres.

simbolo: El carácter (una string) a buscar en el lienzo.

Valor a devolver: Una tupla de cuatro elementos (fila_min, col_min, fila_max, col_max) si se encuentra el símbolo. None en caso contrario.

Ejemplo de Uso:

```
# --- Datos de entrada ---
lienzo_ascii = [
    ['.', '.', '.', '.', '.', '.'],
    ['.', '#', '#', '.', '.', '.'],
    ['.', '#', '.', '.', '.', '.'],
    ['.', '.', '.', '#', '#', '.'],
    ['.', '.', '.', '.', '.', '.']
]

# --- Implementación del código solicitado ---
# TO DO

# --- Llamada y muestra del resultado ---
caja1 = encontrar_caja_delimitadora(lienzo_ascii, '#')
caja2 = encontrar_caja_delimitadora(lienzo_ascii, 'X')

print("La caja delimitadora para '#' es:", caja1)
print("La caja delimitadora para 'X' es:", caja2)

# Resultado esperado:
# La caja delimitadora para '#' es: (1, 1, 3, 4)
# La caja delimitadora para 'X' es: None
```

Ejercicio 7

Objetivo: Desarrollar una función que reciba una lista de inventarios de diferentes almacenes y los consolide en un único inventario maestro. Cada inventario es una lista de productos con sus cantidades.

La función debe, para cada id_producto único, sumar las cantidades de todos los almacenes donde aparece y añadir esta información a una nueva lista de inventario consolidado.

La Función a Implementar:

consolidar_inventarios(lista_de_inventarios)

Parámetros:

lista_de_inventarios: Una lista de listas. Cada lista interna es el inventario de un almacén y contiene tuplas (id_producto, cantidad).

Valor a devolver: Una lista de tuplas con el formato (id_producto, cantidad_total). La lista debe estar ordenada alfabéticamente por id_producto.

Ejemplo de Uso:

--- Datos de entrada ---

```
inventarios_almacenes = [
    [('P001', 50), ('P002', 120)], # Almacén Central
    [('P003', 80), ('P001', 45)],  # Almacén Norte
    [('P002', 75), ('P004', 200)]  # Almacén Sur
]

# --- Implementación del código solicitado ---
# TO DO

# --- Llamada y muestra del resultado ---
inventario_maestro                                                    =
consolidar_inventarios(inventarios_almacenes)
print("Inventario Maestro Consolidado:")
print(inventario_maestro)

# Resultado esperado:
# Inventario Maestro Consolidado:
# [('P001', 95), ('P002', 195), ('P003', 80), ('P004', 200)]
```

Ejercicio 8: Intermedio

Objetivo: Convertir una lista de lecturas de sensores, agrupadas por sensor, a una lista "plana" de eventos individuales. Cada evento debe registrar el ID del sensor, la lectura y un índice temporal.

La Función a Implementar:

transformar_lecturas_sensores(lecturas_agrupadas)

Parámetros:

lecturas_agrupadas: Una lista de listas. Cada lista interna contiene, en la primera posición, el ID del sensor (string), seguido de una serie de lecturas numéricas.

La función debe recorrer cada lista de sensor en lecturas_agrupadas. Y, para cada lectura numérica de un sensor, crear una tupla de evento. El índice temporal para las lecturas de un mismo sensor será su posición (0, 1, 2, …).

Valor a devolver: Una lista de tuplas, donde cada tupla es un evento con el formato (id_sensor, lectura, indice_temporal).

Ejemplo de Uso:

```
# --- Datos de entrada ---
lecturas_sensores = [
    ['TEMP-A', 25.1, 25.3, 25.2],
    ['HUM-B', 60.5, 61.0],
    ['PRES-C', 1012.5]
]

# --- Implementación del código solicitado ---
# TO DO

# --- Llamada y muestra del resultado ---
lista_eventos = transformar_lecturas_sensores(lecturas_sensores)
```

```
print("Lista de Eventos de Sensores:")
for evento in lista_eventos:
    print(evento)

# Resultado esperado:
# Lista de Eventos de Sensores:
# ('TEMP-A', 25.1, 0)
# ('TEMP-A', 25.3, 1)
# ('TEMP-A', 25.2, 2)
# ('HUM-B', 60.5, 0)
# ('HUM-B', 61.0, 1)
# ('PRES-C', 1012.5, 0)
```

Ejercicio 9

Objetivo: Escribir una función que detecte conflictos de horarios entre múltiples agendas (por ejemplo, de salas de reuniones). Un conflicto ocurre si dos reservas en la misma sala se solapan en el tiempo. Si un intervalo termina exactamente cuando empieza el siguiente, no hay solape.

La Función a Implementar: detectar_conflictos_agenda(agenda_completa)

Parámetros:

agenda_completa: Una lista de listas. Cada lista interna representa la agenda de una sala de reuniones y contiene tuplas (hora_inicio, hora_fin) para cada reserva.

La función debe recorrer cada agenda de la agenda_completa y, dentro de cada agenda, comparar cada reserva con todas las demás reservas de esa misma agenda para ver si se solapan. Dos reservas (A_inicio, A_fin) y (B_inicio, B_fin) se solapan si A_inicio < B_fin y A_fin > B_inicio. La función debe devolver una lista de todos los conflictos encontrados.

Valor a devolver: Una lista de tuplas. Cada tupla representa un conflicto y contiene (indice_sala, reserva_1, reserva_2), donde reserva_1 y reserva_2 son las tuplas de las reservas en conflicto.

Ejemplo de Uso:

```
# --- Datos de entrada ---
agendas = [
    [(9, 11), (14, 16)], # Sala 0: Sin conflictos internos
    [(10, 12), (11, 13), (15, 16)], # Sala 1: Conflicto entre (10,
12) y (11, 13)
    [(9, 10), (10, 11), (11, 12)] # Sala 2: Sin conflictos
(terminan justo cuando empiezan)
]

# --- Implementación del código solicitado ---
# TO DO

# --- Llamada y muestra del resultado ---
conflictos = detectar_conflictos_agenda(agendas)
print("Conflictos de agenda encontrados:")
print(conflictos)

# Resultado esperado:
# Conflictos de agenda encontrados:
# [(1, (10, 12), (11, 13))]
```