



**UNIVERSIDAD PRIVADA
DOMINGO SAVIO**

LABORATORIOS 2

Docente: Ing.Jimmy Nataniel Requena Llorentty

Materia: Programación 2

Estudiante: LORA COLODRO FABRIZZIO

Ejercicio N°12 – Búsqueda Lineal y Binaria. –

Descripción:

El código define dos funciones de búsqueda: `busqueda_lineal` y `busqueda_binaria`. Luego, realiza pruebas sobre estas funciones usando `assert` para verificar que funcionan correctamente. Finalmente, realiza un experimento llamado "experimento del caos" que intenta usar la búsqueda binaria en una lista desordenada. Muestra dos algoritmos de búsqueda fundamentales. La búsqueda lineal es versátil y funciona en cualquier lista, pero es lenta en listas grandes. La búsqueda binaria es rápida, pero exige que la lista esté ordenada. El experimento del caos ilustra la importancia de cumplir con los requisitos previos de cada algoritmo.

The screenshot shows a GitHub Codespace interface with two tabs open: 'busqueda.py' and 'busqueda_lineal.py'. The 'busqueda.py' tab contains the following code:

```
#!/usr/bin/python
# 3. Prueba la función con assert
mi_lista_desordenada = [10, 5, 42, 8, 17, 30, 25]
print("Probando busqueda_lineal...")

assert busqueda_lineal(mi_lista_desordenada, 42) == 2
assert busqueda_lineal(mi_lista_desordenada, 10) == 0 # Al inicio
assert busqueda_lineal(mi_lista_desordenada, 25) == 6 # Al final
assert busqueda_lineal(mi_lista_desordenada, 99) == -1 # No existe
assert busqueda_lineal([], 5) == -1 # Lista vacía
print("Pruebas para busqueda_lineal pasaron! ✅")
```

The 'busqueda_lineal.py' tab contains the following code:

```
#!/usr/bin/python
# 2. Definir la función busqueda_binaria
def busqueda_binaria(lista_ordenada, clave):
    izquierda = 0
    derecha = len(lista_ordenada) - 1
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        if lista_ordenada[medio] == clave:
            return medio
        elif clave > lista_ordenada[medio]:
            izquierda = medio + 1
        else:
            derecha = medio - 1
    return -1

# 3. Prueba de la función
lista_ordenada = [2, 5, 8, 12, 16, 23, 38, 56, 72, 91]
print("\nProbando busqueda_binaria...")
```

On the right side of the interface, there is a terminal window showing the output of running the 'busqueda.py' script:

```
~/workspace$ python busqueda.py
Probando busqueda_lineal...
Pruebas para busqueda_lineal pasaron! ✅

Probando busqueda_binaria...
Pruebas para busqueda_binaria pasaron! ✅
fin del programa

Realizando el experimento del caos...
Búsqueda binaria de '5' en lista desordenada devolvió: 1
Fin del programa
Fabrizio Lora (FaXx0)
~/workspace$
```

Ejercicio N°13 – Ordenamiento Burbuja. –

Descripción:

Este código implementa el **algoritmo de ordenamiento burbuja (bubble sort)**, una técnica sencilla y clásica utilizada para organizar listas de números en orden ascendente. El propósito del código es demostrar cómo funciona este método paso a paso y validar su correcto funcionamiento mediante pruebas automáticas. Se basa en comparar elementos adyacentes de una lista e intercambiarlos si están en el orden incorrecto. Este proceso se repite varias veces hasta que toda la lista está ordenada. Aunque no es el método más eficiente para grandes volúmenes de datos, el ordenamiento burbuja es útil en contextos educativos y cuando se necesita un algoritmo fácil de entender y de implementar.

En este código, además de la función principal, se incluye una función de prueba que verifica distintos casos (listas desordenadas, ordenadas, con repetidos, negativas o vacías) para asegurar que el algoritmo funcione correctamente en diversas situaciones. También se muestra una ejecución visual del algoritmo con una lista de ejemplo para observar el antes y el después del ordenamiento.

The screenshot shows a code editor interface with several tabs open. The main tab contains the Python code for bubble sort and its test cases. The code defines a function `ordenamiento_de_burbuja` that takes a list as input and returns it sorted. It uses nested loops to compare adjacent elements and swap them if they are in the wrong order. The `probar_ordenamiento_burbuja` function then tests this implementation against various scenarios, including lists with 1 element, lists in reverse order, lists with duplicates, lists with negative numbers, and an empty list. The terminal tab shows the execution of the test script, displaying the initial and final states of the list, and confirming that all tests passed.

```
def ordenamiento_de_burbuja(lista):
    n = len(lista)
    for i in range(n):
        hubo_intercambio = False
        for j in range(n - 1 - i):
            if lista[j] > lista[j + 1]:
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
                hubo_intercambio = True
        if not hubo_intercambio:
            break
    return lista

def probar_ordenamiento_burbuja():
    # Caso 1: lista pequeña sin orden
    assert ordenamiento_de_burbuja([4, 2, 7, 1]) == [1, 2, 4, 7]

    # Caso 2: lista ya ordenada
    assert ordenamiento_de_burbuja([1, 2, 3, 4, 5]) == [1, 2, 3, 4, 5]

    # Caso 3: lista con números repetidos
    assert ordenamiento_de_burbuja([3, 1, 3, 2, 1]) == [1, 1, 2, 3, 3]

    # Caso 4: lista con números negativos
    assert ordenamiento_de_burbuja([-2, 5, 0, -10, 3]) == [-10, -2, 0, 3, 5]

    # Caso Borte: lista vacía
    assert ordenamiento_de_burbuja([]) == []

    print("Todas las pruebas pasaron correctamente.")

if __name__ == "__main__":
    # Prueba visual
    numeros = [64, 34, 25, 12, 22, 11, 90]
    print("Antes:", numeros)
    ordenamiento_de_burbuja(numeros)
    print("Después:", numeros)
```

~/workspace\$ python 07Lab_ordenamientos_B_Insercion.py
Antes: [64, 34, 25, 12, 22, 11, 90]
Después: [11, 12, 22, 25, 34, 64, 90]
Todas las pruebas pasaron correctamente.
¡Todas las pruebas del ordenamiento por inserción pasaron! ✓
~/workspace\$

Ejercicio N°14 – Ordenamiento Inserción. –

Descripción:

Este código implementa el **algoritmo de ordenamiento por inserción (insertion sort)**, una técnica eficiente para ordenar listas pequeñas o parcialmente ordenadas. El ordenamiento por inserción funciona recorriendo la lista desde el segundo elemento, e insertando cada uno en su posición correcta con respecto a los elementos anteriores. Para ello, desplaza los elementos mayores hacia la derecha hasta encontrar la ubicación adecuada para insertar el valor actual. Es similar a cómo se ordenan las cartas en la mano durante un juego. Aunque no es el más rápido para listas grandes, este método es muy útil por su simplicidad y su buen rendimiento en casos donde la lista ya está casi ordenada.

El código incluye una serie de pruebas automáticas (assert) que validan su comportamiento con diferentes tipos de listas: desordenadas, ya ordenadas, inversamente ordenadas, con elementos duplicados, vacías y con un solo elemento. Estas pruebas aseguran que la función se comporte correctamente en diversos escenarios y resalten errores si ocurren. Al finalizar, se imprime un mensaje confirmando que todas las pruebas fueron exitosas.

The screenshot shows a code editor interface with several tabs open. The main tab contains Python code for an insertion sort algorithm and four test cases. The code is as follows:

```
#!/usr/bin/python
# Ordenamiento por insercion
def ordenamiento_por_insercion(lista):
    for i in range(1, len(lista)):
        valor_actual = lista[i]
        posicion_actual = i
        # Desplazar elementos mayores hacia la derecha
        while posicion_actual > 0 and lista[posicion_actual - 1] > valor_actual:
            lista[posicion_actual] = lista[posicion_actual - 1] #Desplazamiento
            posicion_actual -= 1
        # Insertar el valor actual en su posicion correcta
        lista[posicion_actual] = valor_actual
    return lista

# Caso 1: Lista desordenada
lista1 = [6, 3, 8, 2, 5]
ordenamiento_por_insercion(lista1)
assert lista1 == [2, 3, 5, 6, 8], "Fallo en Caso 1"

# Caso 2: Lista ya ordenada
lista2 = [1, 2, 3, 4, 5]
ordenamiento_por_insercion(lista2)
assert lista2 == [1, 2, 3, 4, 5], "Fallo en Caso 2"

# Caso 3: Lista ordenada a la inversa (peor caso)
lista3 = [5, 4, 3, 2, 1]
ordenamiento_por_insercion(lista3)
assert lista3 == [1, 2, 3, 4, 5], "Fallo en Caso 3"

# Caso 4: Lista con duplicados
lista4 = [1, 1, 2, 2, 3, 3, 4, 4, 5, 5]
```

The terminal window shows the execution of the script and its output:

```
~/workspace$ python 07Lab_ordenamientos_B_Insercion.py
Antes: [64, 34, 25, 12, 22, 11, 99]
Después: [11, 12, 22, 25, 34, 64, 99]
Todas las pruebas pasaron correctamente.
¡Todas las pruebas del ordenamiento por insercion pasaron!
```

Ejercicio N°15 – Ordenamiento Avanzado. –

Descripción:

Este código implementa el **algoritmo de ordenamiento Merge Sort (ordenamiento por mezcla)**, una técnica eficiente basada en el paradigma “divide y vencerás”. Su objetivo es ordenar listas de cualquier tipo de elementos comparables (enteros, flotantes, etc.) dividiéndolas recursivamente en mitades hasta llegar a sublistas de un solo elemento (que están naturalmente ordenadas), y luego combinándolas de forma ordenada.

La función merge_sort se encarga de dividir la lista y aplicar recursivamente el ordenamiento, mientras que la función auxiliar merge une dos sublistas ya ordenadas en una sola lista ordenada. Este algoritmo tiene una eficiencia de tiempo $O(n \log n)$, por lo que es ideal para ordenar grandes volúmenes de datos. Además, el código incluye una serie de pruebas automáticas (assert) que validan su funcionamiento correcto en diferentes casos: listas vacías, con un solo elemento, ordenadas, desordenadas, con duplicados, números negativos y flotantes. También imprime los pasos intermedios de combinación para visualizar cómo se arma la lista final. Esta implementación es útil para comprender cómo funciona el proceso de ordenamiento eficiente y estable en programación.

The screenshot shows a code editor interface with two panes. The left pane displays the source code for `merge_sort.py`. The right pane shows the terminal output of running the script, which includes the steps of dividing the list and merging the sorted halves, along with a summary of successful assertions.

```
1 v def merge_sort(lista):
2     # Paso Vencer (Condición Base de la Recursividad):
3     if len(lista) <= 1:
4         return lista
5     # Paso 1: DIVIDIR
6     medio = len(lista) // 2
7     mitad_izquierda = lista[:medio]
8     mitad_derecha = lista[medio:]
9     # Paso 2: RECURSIVO (recurrir)
10    izquierda_ordenada = merge_sort(mitad_izquierda)
11    derecha_ordenada = merge_sort(mitad_derecha)
12    # Paso 3: COMBINAR
13    print(f"Mezclaría {izquierda_ordenada} y {derecha_ordenada}")
14    return merge(izquierda_ordenada, derecha_ordenada)
15 v def merge(izquierda, derecha):
16     resultado = []
17     i = j = 0
18     # Comparar elementos de izquierda y derecha uno por uno
19     while i < len(izquierda) and j < len(derecha):
20        if izquierda[i] < derecha[j]:
21            resultado.append(izquierda[i])
22            i += 1
23        else:
24            resultado.append(derecha[j])
25            j += 1
26    # Añadir cualquier elemento restante
27    resultado.extend(izquierda[i:])
28    resultado.extend(derecha[j:])
29    return resultado
30
31 # ---- Prueba ---
32 lista_prueba = [8, 3, 5, 1]
33 print("Lista original:", lista_prueba)
34 resultado = merge_sort(lista_prueba)
35
~/workspace$ python class08_ordenamientosAvanzados.py
Lista original: [8, 3, 5, 1]
Mezclaría [8] y [3]
Mezclaría [5] y [1]
Mezclaría [3, 8] y [1, 5]
Lista ordenada: [1, 3, 5, 8]
Mezclaría [5] y [2]
Mezclaría [3] y [2]
Mezclaría [3, 5] y [2]
Mezclaría [5] y [3]
Mezclaría [10] y [3, 5]
Mezclaría [6] y [2]
Mezclaría [8] y [2, 6]
Mezclaría [3, 5, 10] y [2, 6, 8]
Mezclaría [4] y [1]
Mezclaría [5] y [1, 3]
Mezclaría [7, 9] y [1, 3, 5]
Mezclaría [1] y [2]
Mezclaría [4] y [5]
Mezclaría [3, 4, 5]
Mezclaría [2] y [3, 4, 5]
Mezclaría [4] y [2]
Mezclaría [4] y [1]
Mezclaría [2] y [1, 4]
Mezclaría [2, 4] y [1, 2, 4]
Mezclaría [0, 80] y [-100]
Mezclaría [0] y [-100, 50]
Mezclaría [-50, 100] y [-100, 0, 50]
Mezclaría [1, 2] y [3, 8]
Mezclaría [2, 5] y [1, 2, 3, 8]
iTodas las pruebas con assert pasaron correctamente!
```

Ejercicio N°16 – Recorriendo una matriz. –

Descripción:

Este código muestra dos formas básicas de **recorrer una matriz (lista de listas)** en Python, utilizando una matriz de 3x3 como ejemplo. En la primera parte, se emplea un recorrido **por índices**, usando range y la longitud de filas y columnas para acceder a cada elemento individualmente mediante matriz[i][j]. Esto es útil cuando se necesita conocer la posición exacta de cada elemento (índice de fila y columna). En la segunda parte, se utiliza un recorrido **directo por filas**, donde se itera primero por cada fila y luego por cada elemento de esa fila, lo cual es más simple y legible cuando solo se necesita acceder a los valores sin preocuparse por sus posiciones. Este tipo de recorrido es común en operaciones sobre matrices como sumas, transposiciones, o búsqueda de elementos. El código también utiliza print(end=" ") para mostrar los elementos de cada fila en una sola línea, y print() para saltar a la siguiente línea al final de cada fila. Es una forma clara y eficiente de trabajar con estructuras bidimensionales en Python.

```
matriz.py > ...
2  matriz = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
3  # Ahora si podemos obtener dimensiones y recorrer
4  num_filas = len(matriz) # 3 filas
5  num_columnas = len(matriz[0]) # 3 columnas
6  # Recorremos usando indices
7  for i in range(num_filas): # i = 0, 1, 2
8    for j in range(num_columnas): # j = 0, 1, 2
9      elemento = matriz[i][j]
10     print("Elemento en ({}, {}) es {}".format(i, j, elemento))
11   # Recorremos cada fila
12  for fila_actual in matriz:
13    # Recorremos cada elemento dentro de la fila actual
14    for elemento in fila_actual:
15      print(elemento, end=" ")
16    print() # Salto de linea al final de cada fila
17  print("Fabrizio Lora (Faxx@)")
18  # -----
19  # Laboratorio de matrices
20

AI (Python) Diff
Shell + +
~> workspace$ python matriz.py
Elemento en (0,0) es 1
Elemento en (0,1) es 2
Elemento en (0,2) es 3
Elemento en (1,0) es 4
Elemento en (1,1) es 5
Elemento en (1,2) es 6
Elemento en (2,0) es 7
Elemento en (2,1) es 8
Elemento en (2,2) es 9
1
2
3
4 5 6
7 8 9
Fabrizio Lora (Faxx@)
```

Ejercicio N°17 – Simulación con matrices Teclado numérico. –

Descripción:

Este código trabaja con una **matriz 3x3 que simula un teclado numérico**, donde cada (sublista) representa una fila del teclado. En la primera parte, se crea la matriz inicial con los números del 1 al 9 organizados en tres filas. Luego, se imprime la matriz completa utilizando un ciclo (for), mostrando cada fila por separado. A continuación, se accede e imprime directamente el número del **centro del teclado** (el número 5, que se encuentra en la posición [1][1]) y el número en la **esquina inferior derecha** (el 9, en la posición [2][2]). Después, se realiza una modificación en la matriz: se cambia el valor 1 (ubicado en la esquina superior izquierda [0][0]) por un 0. Finalmente, se imprime nuevamente la matriz para mostrar el cambio realizado.

Este ejemplo es útil para comprender cómo trabajar con matrices bidimensionales en Python, acceder a elementos específicos mediante índices, y modificar valores directamente en estructuras tipo lista. Además, ilustra cómo representar visualmente estructuras como teclados o tableros.

The screenshot shows a Jupyter Notebook workspace with several files listed in the top bar: 'matriz.py', '07Lab_ordenamientos_B_In...', 'class08_ordenamientosAvan...', 'proyecto.py', 'C_E=G_A_CINE.py', 'Batalla_Naval.py', 'BatallaNaval_mejorada.py', and 'class12_Diccionar...'. The 'matriz.py' file is open in the code editor. The code defines a 3x3 matrix 'teclado' with values [1, 2, 3], [4, 5, 6], and [7, 8, 9]. It prints the matrix, accesses the center element (5), and the bottom-right element (9). It then changes the top-left element (1) to 0 and prints the modified matrix. The output cell shows the original matrix, the accessed elements, the modified matrix, and a final message from the developer.

```
matriz.py > ...
18 print("Fabrizio Lora (FaXx0)")
19 #
20 # Laboratorio de matrices
21 # Ejercicio 1: Imprimir una matriz
22 #
23 teclado = [[1, 2, 3], [4, 5, 6], [7, 8, 9]] # 1. Crear la matriz 3x3 que representa el teclado numérico
24 print("Matriz completa:") # 2. Imprimir la matriz completa
25 for fila in teclado:
26     print(fila)
27 # 3. Acceder e imprimir elementos específicos
28 print("\nNúmero en el centro:", teclado[1][1]) # El 5 (fila 1, columna 1)
29 print("Número en la esquina inferior derecha:", 
30      teclado[2][2]) # El 9 (fila 2, columna 2) # 4. Modificar el número en la esquina superior izquierda (1 por 0)
31 teclado[0][0] = 0
32 print("\nMatriz después de la modificación:")
33 for fila in teclado:
34     print(fila)
35 print("Fabrizio Lora (FaXx0)")
36 #
```

```
Shell | + ~ /workspace: python matriz.py
Matriz completa:
[1, 2, 3]
[4, 5, 6]
[7, 8, 9]
Número en el centro: 5
Número en la esquina inferior derecha: 9
Matriz después de la modificación:
[0, 2, 3]
[4, 5, 6]
[7, 8, 9]
Fabrizio Lora (FaXx0)
? Probando_sumar_total_matriz...
```

Ejercicio N°18 – Suma total de una matriz. –

Descripción:

Este código define una función llamada **sumar_total_matriz** que calcula la **suma total de todos los elementos numéricos en una matriz** (lista de listas) en Python. La función recorre cada fila de la matriz y luego cada elemento dentro de esa fila, acumulando su valor en una variable total, que se devuelve al final. Esta técnica es útil cuando se desea realizar operaciones sobre toda la estructura matricial, como sumar valores en una tabla, un tablero o un conjunto de datos bidimensional.

Además, se incluye una función de prueba llamada **probar_suma_total**, que utiliza assert para verificar que la función principal funciona correctamente bajo diferentes escenarios: una matriz normal, una con números negativos y ceros, una fila vacía, una matriz completamente vacía y una matriz con un solo valor. Estas pruebas aseguran que la función sea robusta y capaz de manejar tanto casos comunes como bordes. Finalmente, se imprime un mensaje confirmando que todas las pruebas pasaron correctamente. Este tipo de validación es clave al desarrollar funciones reutilizables y confiables en programación.

The screenshot shows a Jupyter Notebook interface with the following code in a cell:

```
matriz.py > ...
38 #-----#
39 # Definimos la función que suma todos los elementos de una matriz
40 v def sumar_total_matriz(matriz):
41     total = 0
42     for fila in matriz:
43         for elemento in fila:
44             total += elemento
45     return total
46 # Función para probar que sumar_total_matriz funciona correctamente
47 v def probar_suma_total():
48     print("Probando sumar_total_matriz...")
49     # Caso 1: matriz normal
50     m1 = [[1, 2, 3], [4, 5, 6]]
51     assert sumar_total_matriz(m1) == 21 # 1+2+3+4+5+6 = 21
52     # Caso 2: matriz con negativos y ceros
53     m2 = [[-1, 0, 1], [10, -5, 5]]
54     assert sumar_total_matriz(m2) == 10 # -1+0+1+10-5+5 = 10
55     # Caso 3: matriz con límites
56     assert sumar_total_matriz([]) == 0 # Matriz con una fila vacía
57     assert sumar_total_matriz([[]]) == 0 # Matriz completamente vacía
58     assert sumar_total_matriz([[42]]) == 42 # Matriz de un solo elemento
59     print("Pruebas para sumar_total_matriz pasaron! (matricial) ✅")
60 # Llamadas a la función de pruebas
61 probar_suma_total()
62 print("Fabrizio Lora (FaXx0)")
```

Below the code cell, the notebook's shell shows the command run and its output:

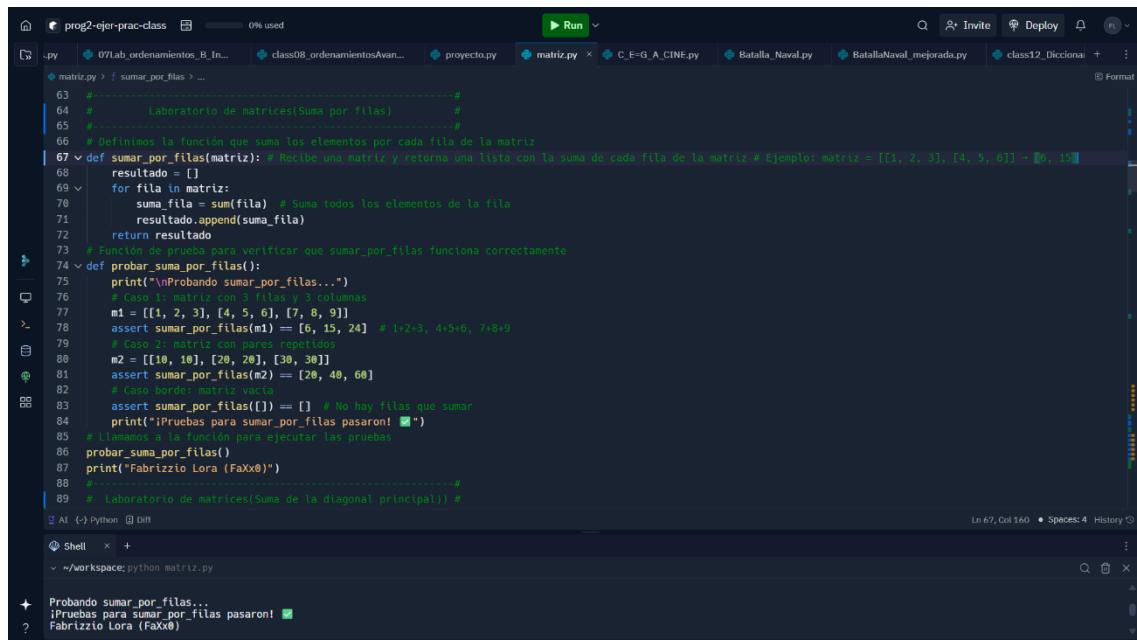
```
~/workspace$ python matriz.py
Fabrizio Lora (FaXx0)
Probando sumar_total_matriz...
Pruebas para sumar_total_matriz pasaron! (matricial) ✅
Fabrizio Lora (FaXx0)
```

Ejercicio N°19 - Suma por filas de una matriz. –

Descripción:

Este código define la función **sumar_por_filas**, que toma como entrada una **matriz (lista de listas)** y devuelve una **lista con la suma de los elementos de cada fila**. Para cada fila en la matriz, se utiliza la función incorporada `sum()` para calcular su total, y ese resultado se agrega a una lista llamada `resultado`. Este tipo de operación es útil cuando se necesita obtener estadísticas por fila, como totales por categoría, puntuaciones o registros.

Además, se incluye una función de prueba llamada **probar_suma_por_filas**, la cual valida el funcionamiento correcto de la función principal mediante casos de prueba. Se verifican: una matriz 3x3 con valores secuenciales, una matriz con pares repetidos, y un caso borde con una matriz vacía (sin filas). Las afirmaciones con `assert` garantizan que la función devuelva los resultados esperados, y si todas las pruebas pasan, se muestra un mensaje de éxito. Este enfoque asegura la fiabilidad del código ante distintos escenarios, demostrando cómo aplicar pruebas automáticas simples en funciones que procesan matrices.



The screenshot shows a Jupyter Notebook interface with several open files. The current file is `matriz.py`, which contains the following code:

```
63 # -----
64 #   Laboratorio de matrices(Suma por filas)
65 #
66 # Definimos la función que suma los elementos por cada fila de la matriz
67 def sumar_por_filas(matriz): # Recibe una matriz y retorna una lista con la suma de cada fila de la matriz
68     resultado = []
69     for fila in matriz:
70         suma_fila = sum(fila) # Soma todos los elementos de la fila
71         resultado.append(suma_fila)
72     return resultado
73 # Función de prueba para verificar que sumar_por_filas funciona correctamente
74 def probar_suma_por_filas():
75     print("\nProbando sumar_por_filas...")
76     # Caso 1: matriz con 3 filas y 3 columnas
77     m1 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
78     assert sumar_por_filas(m1) == [6, 15, 24] # 1+2+3, 4+5+6, 7+8+9
79     # Caso 2: matriz con pares repetidos
80     m2 = [[10, 10], [20, 20], [30, 30]]
81     assert sumar_por_filas(m2) == [20, 40, 60]
82     # Caso borde: matriz vacía
83     assert sumar_por_filas([]) == [] # No hay filas que sumar
84     print("Pruebas para sumar_por_filas pasaron! ✅")
85     # Llamamos a la función para ejecutar las pruebas
86     probar_suma_por_filas()
87     print("Fabrizio Lora (FaXx0)")
88 # -----
89 # Laboratorio de matrices(Suma de la diagonal principal) #
```

Below the code cell, there is a shell terminal window showing the output of running the script:

```
Probando sumar_por_filas...
Pruebas para sumar_por_filas pasaron! ✅
Fabrizio Lora (FaXx0)
```

Ejercicios N°20 – suma diagonal principal de una matriz. –

Descripción:

Este código implementa la función **sumar_diagonal_principal**, que recibe una **matriz cuadrada** (es decir, con el mismo número de filas y columnas) y retorna la suma de los elementos ubicados en su **diagonal principal**. La diagonal principal de una matriz contiene los elementos que están en la misma posición de fila y columna, es decir, en las coordenadas (0,0), (1,1), (2,2), etc. Dentro de la función, se utiliza un bucle (for) que recorre los índices de la matriz y va sumando los valores matriz[i][i] a una variable suma.

El código también incluye la función **probar_suma_diagonal_principal**, que contiene varios casos de prueba para asegurarse de que la función funciona correctamente. Se prueba con matrices de diferentes tamaños: una matriz 3x3 con números consecutivos, una 2x2 con ceros y valores definidos, y una matriz 1x1 como caso borde. Los assert aseguran que el resultado sea el esperado, y si todas las pruebas se ejecutan sin errores, se imprime un mensaje indicando que las pruebas pasaron correctamente. Este ejercicio es útil para trabajar con índices en matrices y para entender cómo acceder a elementos diagonales en estructuras bidimensionales.

```
prog2-ejer-prac-class 0% used
Run ▾
Q Invite Deploy ⌂ Format
matrix.py > / sumar_diagonal_principal > ...
85 # Llamamos a la función para ejecutar las pruebas
86 probar_suma_por_filas()
87 print("Fabrizio Lora (FaXx0)" )
88 #-----#
89 # Laboratorio de matrices(Suma de la diagonal principal) #
90 #-----#
91 def sumar_diagonal_principal(matriz): Definimos la función que suma los elementos de la diagonal principal de una matriz cuadrada (misma cantidad de filas y columnas) y retorna la suma de los elementos en su diagonal principal. Ejemplo: matriz = [[1, 2], [3, 4]] diagonal principal: 1 + 4 = suma = 5
92     suma = 0
93     for i in range(len(matriz)):
94         suma += matriz[i][i] # Accede al elemento en la posición (i, i)
95     return suma
96 # Función de prueba para verificar que sumar_diagonal_principal funciona correctamente
97 def probar_suma_diagonal_principal():
98     print("\nProbando: sumar_diagonal_principal...")
99     # Caso 1: matriz 3x3 con números consecutivos
100    m1 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
101    assert sumar_diagonal_principal(m1) == 15 # 1 + 5 + 9
102    # Caso 2: matriz 2x2 con ceros y valores definidos
103    m2 = [[10, 0], [0, 20]]
104    assert sumar_diagonal_principal(m2) == 30 # 10 + 20
105    # Caso borde: matriz 1x1
106    m3 = [[5]]
107    assert sumar_diagonal_principal(m3) == 5 # Solo un elemento en la diagonal
108    print("¡Pruebas para sumar_diagonal_principal pasaron! ✅")
109 # Llamamos a la función para ejecutar las pruebas
110 probar_suma_diagonal_principal()

At <Python 3.8.5>
Shell + 
~/workspace:python matrix.py
Fabrizio Lora (FaXx0)
+ Probando sumar_diagonal_principal...
? ¡Pruebas para sumar_diagonal_principal pasaron! ✅

```

Ejercicio N°21 – Suma diagonal secundaria de una matriz. –

Descripción:

Este código define la función **sumar_diagonal_secundaria**, que calcula la **suma de los elementos ubicados en la diagonal secundaria** de una **matriz cuadrada** (es decir, con igual número de filas y columnas). La **diagonal secundaria** está formada por los elementos que van desde la esquina superior derecha hasta la esquina inferior izquierda, cuyas posiciones siguen el patrón $(i, n - 1 - i)$, donde n es el tamaño de la matriz. La función recorre la matriz con un ciclo (for) y va sumando los valores correspondientes a esa diagonal.

Se incluye también una función de prueba llamada **probar_suma_diagonal_secundaria**, que valida el funcionamiento correcto mediante varios casos de prueba. Se consideran una matriz 3x3 con números del 1 al 9, una 2x2 con ceros y valores definidos, y un caso borde con una matriz 1x1. Cada prueba usa assert para asegurar que el valor retornado sea el esperado. Si todas las pruebas son correctas, se imprime un mensaje de confirmación. Este código es útil para aprender cómo acceder y operar sobre diagonales de una matriz, aplicando lógica de índices inversos y validación mediante pruebas automatizadas.

```
prog2-ejer-prac-class 0% used
Run
Q Invite Deploy ⚙️
matriz.py > ...
117 v def sumar_diagonal_secundaria(matriz): #Esta función recibe una matriz cuadrada (misma cantidad de filas y columnas) y retorna la suma de los elementos en su diagonal secundaria.Ejemplo: matriz = [[1, 2, 3], [4, 5, 6], [7, 8, 9]] diagonal secundaria: 2 + 3 + 5 = 10
118     # Definimos la función que suma los elementos de la diagonal secundaria de una matriz cuadrada
119     suma = 0 # Inicializamos la suma en 0
120     n = len(
121         matriz
122     ) # Obtenemos la dimensión de la matriz (número de filas o columnas)
123     for i in range(n): # Iteramos sobre el rango de la dimensión de la matriz
124         suma += matriz[i][
125             n - 1 - i] # Accedemos al elemento en la posición (i, n - 1 - i)
126     return suma # Retornamos la suma de los elementos de la diagonal secundaria
127     # Función de prueba para verificar que sumar_diagonal_secundaria funciona correctamente
128 v def probar_suma_diagonal_secundaria():
129     print("\nProbando sumar_diagonal_secundaria...")
130     # Caso 1: matriz 3x3 con números consecutivos
131     m1 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
132     assert sumar_diagonal_secundaria(m1) == 15 # 3 + 5 + 7
133     # Caso 2: matriz 2x2 con ceros y valores definidos
134     m2 = [[10, 0], [0, 20]]
135     assert sumar_diagonal_secundaria(m2) == 20 # 0 + 20
136     # Caso 3: matriz 1x1
137     m3 = [[5]]
138     assert sumar_diagonal_secundaria(
139         m3) == 5 # Solo un elemento en la diagonal
140     print("Pruebas para sumar_diagonal_secundaria pasaron! ✅")
141     # Llamamos a la función para ejecutar las pruebas
142     print("Fabrizio Lora (FaXx0)")
143
AI ↗ Python DIFF
Shell + ~workspace/python matriz.py
Fabrizio Lora (FaXx0)
Pruebas para sumar_diagonal_secundaria pasaron! ✅
Fabrizio Lora (FaXx0)
Suma de la diagonal secundaria2: 15
```

Opción 2 - de suma diagonal secundaria de una matriz. –

Descripción:

Este código define la función **sumar_diagonal_secundaria2**, que calcula la **suma de los elementos de la diagonal secundaria** de una **matriz cuadrada** (una matriz con igual número de filas y columnas). Antes de realizar la suma, la función **verifica que la matriz sea cuadrada**, recorriendo cada fila y comprobando que su longitud coincida con el número total de filas. Si no se cumple esta condición, se lanza un **ValueError** para evitar errores de índice.

Luego, la función recorre la matriz utilizando un bucle for que va desde la primera hasta la última fila. En cada iteración, accede al elemento de la **diagonal secundaria**, cuya posición corresponde a la columna $n - 1 - i$, donde i es el índice de la fila actual y n es el tamaño de la matriz. Los valores de esa diagonal se van sumando en una variable suma, que se retorna al final.

En la parte final del código, se muestra un **ejemplo de uso**: se define una matriz 3x3 con valores del 1 al 9, se llama a la función con esa matriz y se imprime el resultado. La suma esperada es 15 (por los elementos 3 + 5 + 7). Este código es un buen ejemplo de buenas prácticas, ya que incluye validación de entrada, documentación clara en los comentarios, uso de condiciones defensivas (ValueError) y una demostración directa de cómo utilizar la función.

The screenshot shows a Jupyter Notebook environment with the following code in a cell:

```
def sumar_diagonal_secundaria2(matriz_cuadrada):
    """Calcula la suma de los elementos en la diagonal secundaria de una matriz cuadrada. La diagonal secundaria es la que va desde la esquina superior derecha hasta la esquina inferior izquierda. #Parámetros: #matriz_cuadrada (list(list)): Matriz cuadrada (N x N) de números
    #Retorna: int/float: Suma de los elementos en la diagonal secundaria
    #Lanza: ValueError: si la matriz no es cuadrada # Verificar si la matriz es cuadrada
    n = len(matriz_cuadrada)
    for fila in matriz_cuadrada:
        if len(fila) != n:
            raise ValueError(
                "La matriz debe ser cuadrada (mismo número de filas y columnas)"
            )
    # Calcular suma de diagonal secundaria
    suma = 0
    for i in range(n):
        j = n - 1 - i # Índice de columna para la diagonal secundaria
        suma += matriz_cuadrada[i][j]
    return suma
# Ejemplo de uso
if __name__ == "__main__":
    matriz = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
    resultado = sumar_diagonal_secundaria2(matriz)
    print(
        f"Suma de la diagonal secundaria: {resultado}"
    ) # Salida: 15 (3 + 5 + 7) # Llamamos a la función para ejecutar las pruebas y ejecutámos las pruebas de la función sumar_diagonal_secundaria2 con la matriz de ejemplo
    print("Fabrizio Lora (FaXx0)")
```

The cell output shows the results of running the code:

```
Fabrizio Lora (FaXx0)
Suma de la diagonal secundaria: 15
Fabrizio Lora (FaXx0)
¡Pruebas para transponer_matriz pasaron!
```

Ejercicio N°22 – Suma de una columna. –

Descripción:

Este código implementa la función **transponer_matriz**, que recibe una **matriz (lista de listas)** y devuelve su **matriz transpuesta**. Transponer una matriz significa **intercambiar sus filas por columnas**, es decir, que el elemento que estaba en la posición (i, j) pasa a estar en la posición (j, i) . Por ejemplo, una matriz de 2×3 (2 filas y 3 columnas) se convierte en una de 3×2 (3 filas y 2 columnas).

El código primero verifica si la matriz está vacía (o si la primera fila está vacía) para evitar errores. Luego calcula el número de filas y columnas, y construye dinámicamente la matriz transpuesta. Esto se logra recorriendo las columnas originales (j) y, dentro de cada una, recorriendo las filas (i), extrayendo los elementos $\text{matriz}[i][j]$ y agregándolos en una nueva fila.

También se incluye una función de prueba llamada **probar_transponer_matriz**, que verifica el correcto funcionamiento de la transposición en distintos casos: una matriz rectangular 2×3 , una matriz cuadrada 2×2 , y una matriz 1×1 . Cada prueba usa `assert` para confirmar que el resultado es el esperado, e imprime un mensaje si pasa correctamente. Al final, se imprime que todas las pruebas han sido exitosas. Este ejemplo es ideal para entender cómo manipular matrices bidimensionales en Python, y cómo implementar la transposición manualmente, sin usar librerías externas.

The screenshot shows a code editor interface with the following details:

- File:** The current file is `matriz.py`.
- Code Content:**

```
172 v def transponer_matriz(matriz):
173 v     if not matriz or not matriz[0]:
174 v         return []
175 v     num_filas = len(matriz)
176 v     num_columnas = len(matriz[0])
177 v     # Construimos la matriz transpuesta con la estructura correcta(o la construimos dinamicamente)
178 v     matriz_transpuesta = []
179 v     for j in range(num_columnas):
180 v         nueva_fila = []
181 v         for i in range(num_filas):
182 v             nueva_fila.append(matriz[i][j])
183 v         matriz_transpuesta.append(nueva_fila)
184 v     return matriz_transpuesta
185 #
186 v def probar_transponer_matriz():
187 v     print("\nProbando transponer_matriz...")
188 v     # Caso 1: matriz 3x3 con numeros consecutivos
189 v     m1 = [[1, 2, 3], [4, 5, 6]] #2x3
190 v     t1 = transponer_matriz(m1)
191 v     assert t1 == [[1, 4], [2, 5], [3, 6]] #debe ser 3x2
192 v     print("Prueba 1 pasada!")
193 v     # Caso 2: matriz 2x2 con ceros y valores definidos
194 v     m2 = [[10, 0], [0, 20]] #2x2
195 v     t2 = transponer_matriz(m2)
196 v     assert t2 == [[10, 0], [0, 20]] #debe ser 2x2
197 v     print("Prueba 2 pasada!")
198 v     # Caso 3: matriz 1x1
```
- Output:**

```
!Pruebas para transponer_matriz pasaron! ✓
Fabrizio Lora (Fxx0)
!Pruebas para es_Identidad pasaron! ✓
Fabrizio Lora (Fxx0)
```

Ejercicio N°23 – Matriz identidad. –

Descripción:

Este código define la función `es_identidad`, que evalúa si una **matriz cuadrada** es una **matriz identidad**. Una matriz identidad es una matriz en la que todos los elementos de la **diagonal principal** son iguales a 1, y todos los demás elementos (fuera de la diagonal) son 0. La función comienza comprobando si la matriz es cuadrada (es decir, si el número de columnas en cada fila coincide con el número total de filas). Si no se cumple esta condición, retorna False.

Luego, la función recorre la matriz con dos bucles anidados: uno para las filas (i) y otro para las columnas (j). Para cada elemento, verifica que si está en la diagonal principal ($i == j$), su valor debe ser 1, y si está fuera de la diagonal ($i \neq j$), su valor debe ser 0. Si alguna de estas condiciones falla, la función retorna False. Si todas se cumplen, retorna True.

La función `probar_es_identidad` ejecuta varios casos de prueba usando assert para validar diferentes matrices: matrices identidad de tamaños 3x3, 2x2 y 1x1; y matrices que no cumplen con las condiciones necesarias para ser identidad. Cada prueba imprime un mensaje si se ejecuta correctamente. Al final, se confirma que todas las pruebas han sido superadas. Este código es útil para entender cómo se validan propiedades específicas en matrices, y cómo usar condiciones lógicas combinadas con recorridos matriciales en Python.

The screenshot shows a Jupyter Notebook interface with the following details:

- File Explorer:** Shows files like `matriz.py`, `C_E=G_A_CINE.py`, `Batalla_Naval.py`, etc.
- Code Cell:** Displays the `matriz.py` code:

```
200 > #def es_identidad(matriz):
201     #Requisito 1:debe ser cuadrada
202     num_filas = len(matriz)
203     for fila in matriz:
204         if len(fila) != num_filas:
205             return False
206         #Requisito 2:diagonal principal debe ser 1 y el resto 0
207         for i in range(num_filas): #Recorremos la matriz
208             for j in range(num_filas): #Recorremos la matriz
209                 if i == j: #i es la diagonal principal
210                     if matriz[i][j] != 1: #Si no es 1, no es identidad
211                         return False #Si no es 1, no es identidad
212                 else: #Si no es la diagonal principal
213                     if matriz[i][j] != 0: #Si no es 0, no es la diagonal principal
214                         return False #Si no cumple con alguno de los requisitos, no es identidad
215         return True #Si pasa ambos requisitos, es identidad
216     #Prueba
217 > def probar_es_identidad():
218     print("\nProbando es_identidad...")
219     #Caso 1: matriz 3x3 identidad
220     m1 = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
221     assert es_identidad(m1) == True
222     print("Prueba 1 pasada!")
223     #Caso 2: matriz 2x2 identidad
224     m2 = [[1, 0], [0, 1]]
225     assert es_identidad(m2) == True
226     print("Prueba 2 pasada!")
```
- Shell:** Shows the command `python matriz.py` and its output:

```
!Pruebas para transponer_matriz pasaron! ✓
Fabrizio Lora (Fxx0)
!Pruebas para es_identidad pasaron! ✓
Fabrizio Lora (Fxx0)
```

Ejercicio N°24 – Matriz simétrica. –

Descripción:

Este código define la función **es_simetrica**, que determina si una **matriz cuadrada es simétrica**. Una matriz es simétrica si es **igual a su transpuesta**, es decir, si el elemento en la posición (i, j) es igual al elemento en (j, i) para todas las posiciones posibles. Para verificarlo, primero se comprueba si la matriz es cuadrada (mismo número de filas y columnas). Si no lo es, la función retorna False inmediatamente.

Luego, se recorre la matriz utilizando dos bucles anidados: uno para las filas (i) y otro para las columnas (j). En cada iteración, se comparan los elementos simétricos respecto a la diagonal principal ($\text{matriz}[i][j]$ con $\text{matriz}[j][i]$). Si alguno de estos pares no coincide, la función retorna False. Si todas las comparaciones son correctas, retorna True.

La función **probar_es_simetrica** contiene una serie de **pruebas automáticas con assert** para verificar distintos escenarios: matrices simétricas de tamaño 3x3, 2x2, 1x1 y 1x1 con ceros; así como matrices que no son simétricas. Cada prueba imprime un mensaje indicando que ha sido superada, y al final se confirma que todas pasaron correctamente. Este código es útil para entender cómo identificar matrices simétricas en programación, trabajar con posiciones espejadas en una estructura bidimensional y validar con pruebas automatizadas.

The screenshot shows a code editor interface with a Python file named 'matriz.py' open. The code defines a function `es_simetrica` that checks if a matrix is symmetric. It first checks if the matrix is square. If not, it returns False. Then it iterates through all elements in the matrix. For each element at position (i, j) , it checks if the element at (j, i) is equal. If any pair does not match, it returns False. Otherwise, it returns True. Below this, there is a section for tests using the `assert` statement. It defines several test matrices `m1`, `m2`, and `m3` and uses `assert` to check if `es_simetrica(m1)`, `es_simetrica(m2)`, and `es_simetrica(m3)` return True. The output window at the bottom shows the results of running the script, indicating that all tests passed successfully.

```
matriz.py > ./probar_es_simetrica...
256     def es_simetrica(matriz): # Función para verificar si una matriz es simétrica
257         num_filas = len(matriz)
258         num_columnas = len(matriz[0])
259         if num_filas != num_columnas: # Requisito 1: matriz cuadrada
260             return False
261         for i in range(num_filas): # Requisito 2: igual a su transpuesta
262             for j in range(num_columnas):
263                 if matriz[i][j] != matriz[j][i]:
264                     return False
265         return True
266
267     # Pruebas
268     def probar_es_simetrica():
269         print("\nProbando es_simetrica...")
270         m1 = [[1, 2, 3], [2, 4, 5], [3, 5, 6]]
271         assert es_simetrica(m1) == True
272         print("Prueba 1 pasada!")
273         m2 = [[1, 2], [2, 3]]
274         assert es_simetrica(m2) == True
275         print("Prueba 2 pasada!")
276         m3 = [[11]]
277         assert es_simetrica(m3) == True
278         print("Prueba 3 pasada!")

Probando es_simetrica...
Prueba 1 pasada!
Prueba 2 pasada!
Prueba 3 pasada!
Prueba 4 pasada!
Prueba 5 pasada!
Prueba 6 pasada!
¡Pruebas para es_simetrica pasaron!
```

Ejercicio N°25 – Sala de cine usando matrices. –

Descripción:

Este código crea un **sistema básico de reservas para una sala de cine** usando una **matriz bidimensional**, donde 'L' representa un asiento libre y 'O' uno ocupado. El usuario puede ver la sala, seleccionar asientos y saber cuántos quedan disponibles, todo desde la consola.

Las funciones principales permiten **crear la sala, mostrarla visualmente con colores, ocupar asientos y contar los libres**. El programa valida que las posiciones sean válidas y que el asiento esté disponible antes de marcarlo como ocupado.

Es un ejemplo claro y práctico para aprender a manejar **matrices, validación de entradas, y visualización en consola** en Python, ideal para estudiantes o proyectos educativos.

The screenshot shows a Jupyter Notebook interface with two panes. The left pane displays the source code for `C_E=6_A_CINE.py`. The right pane shows the terminal output of running the script.

Code Content:

```
def crear_sala(filas, columnas): # Función para crear una matriz de asientos inicialmente libres
    # === Crear matriz de asientos inicialmente libres ===
    return [[L' for _ in range(columnas)] for _ in range(filas)] # Retorna una matriz de asientos inicialmente libres

def mostrar_sala(sala): # Función para mostrar la sala de cine de forma visual
    # === Muestra la sala de cine de forma visual ===
    print("\n" * 50) # Imprime una línea de separación
    print("SALA DE CINE".center(50)) # Imprime el título del programa
    print("\n" * 50) # Imprime una línea de separación
    # Encabezado numérico para columnas #
    print(" ", end="") # Espacio para alinear con filas
    for j in range(len(sala[0])): # Bucle para imprimir el encabezado numérico para columnas
        print(f"\t{j+1}", end="") # Imprime el encabezado numérico para columnas
    print("\n" + "\n" * (4 + 4*len(sala[0]))) # Imprime una línea de separación
    # Filas con asientos #
    for i, fila in enumerate(sala): # Bucle para imprimir las filas con asientos
        print(f"\t{i+1}: ", end="") # Imprime el número de fila
        for asiento in fila: # Bucle para imprimir los asientos de la fila
            color = "\u033[92m" if asiento == 'L' else "\u033[91m" # Verde libre / Rojo ocupado
            print(f"\t{color}{asiento}\u033[0m", end="") # Imprime el asiento con color
        print() # Salto de linea
    print("Leyenda: L = Libre, O = Ocupado") # Imprime la leyenda de colores

def ocupar_asiento(sala, fila, columna): # Función para ocupar un asiento si está disponible
    # === Intenta ocupar un asiento si está disponible ===
    # Validar rango de filas y columnas
    if not (1 <= fila <= len(sala)) or not (1 <= columna <= len(sala[0])): # Validación de rango de filas y columnas
        print("\u033[91mError: Asiento fuera de rango\u033[0m") # Mensaje de error
        return False # Retorna falso si el asiento está fuera de rango
    fila_idx = fila - 1 # Índice de fila
    col_idx = columna - 1 # Índice de columna
    if sala[fila_idx][col_idx] == 'L': # Si el asiento está libre
        sala[fila_idx][col_idx] = 'O' # Ocupa el asiento
    else:
        print("\u033[91mError: Asiento ya ocupado\u033[0m") # Mensaje de error
        return False # Retorna falso si el asiento ya está ocupado
    return True # Retorna verdadero si se ocupó el asiento
```

Terminal Output:

```
~/workspace$ python C_E=6_A_CINE.py
=====
SALA DE CINE
=====
1 2 3 4 5 6 7 8
1 | L L L L L L L
2 | L L L L L L L
3 | L L L L L L L
4 | L L L L L L L
5 | L L L L L L L
Leyenda: L = Libre, O = Ocupado

===== MENÚ PRINCIPAL =====
Asientos disponibles: 40

Opciones:
1: Ocupar asiento
0: Salir

Seleccione una opción: 1
Fila (1-5): 4
Columna (1-8): 4
Asiento 4,4 ocupado con éxito!

Presione Enter para continuar...
===== SALA DE CINE =====
1 2 3 4 5 6 7 8
1 | L L L L L L L
2 | L L L L L L L
3 | L L L L L L L
4 | L L L O L L L
5 | L L L L L L L
Leyenda: L = Libre, O = Ocupado

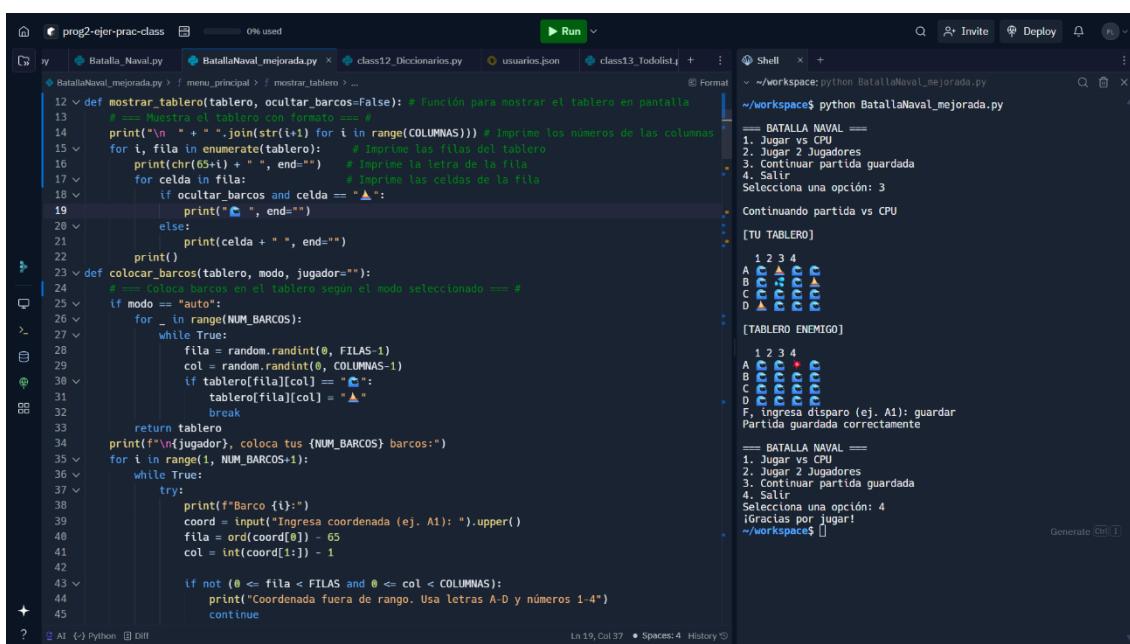
===== MENÚ PRINCIPAL =====
Asientos disponibles: 39
```

Ejercicio N°26 – juego de Batalla naval. –

Descripción:

Este programa en Python implementa el popular juego de **Batalla Naval**, con funcionalidad completa como colocación de barcos, disparos por turnos, juego contra la CPU o entre dos jugadores, y la capacidad de **guardar y continuar partidas**. El tablero es representado por listas de listas (matrices), y las funciones como crear tablero, mostrar tablero, colocar barcos, realizar disparo y guardar partida permiten estructurar el programa de forma modular y clara. Estas funciones son llamadas desde bucles (while) que controlan el flujo del juego, permitiendo repetir acciones como disparar hasta que uno de los jugadores gane.

Aunque este código no aplica directamente el **ordenamiento burbuja (bubble sort)**, dicho algoritmo puede resultar útil si se desea agregar funcionalidades adicionales como **ordenar una tabla de puntuaciones, tiempos de partida o rankings de jugadores**. El ordenamiento burbuja es uno de los algoritmos de ordenamiento más sencillos y se basa en el uso de **bucles anidados** para comparar pares de elementos y reorganizarlos en orden. Esto lo hace ideal para explicar conceptos básicos de programación como ciclos (for), condicionales (if), e intercambio de valores. Además, su implementación en Python es didáctica para quienes están aprendiendo cómo manipular listas, recorrer estructuras y usar funciones correctamente dentro de un programa más grande.



The screenshot shows a code editor with several files open on the left and a terminal window on the right. The terminal output is as follows:

```
~/workspace$ python BatallaNaval_mejorada.py
== BATALLA NAVAL ==
1. Jugar vs CPU
2. Jugar 2 Jugadores
3. Continuar partida guardada
4. Salir
Selecciona una opción: 3
Continuando partida vs CPU
[TU TABLERO]
 1 2 3 4
A ⚡ ⚡ ⚡ 
B ⚡ ⚡ ⚡ 
C ⚡ ⚡ ⚡ 
D ⚡ ⚡ ⚡ 
[TABLERO ENEMIGO]
 1 2 3 4
A ⚡ ⚡ ⚡ 
B ⚡ ⚡ ⚡ 
C ⚡ ⚡ ⚡ 
D ⚡ ⚡ ⚡ 
F, ingresa disparo (ej. A1): guardar
Partida guardada correctamente
== BATALLA NAVAL ==
1. Jugar vs CPU
2. Jugar 2 Jugadores
3. Continuar partida guardada
4. Salir
Selecciona una opción: 4
¡Gracias por jugar!
~/workspace$
```

The code editor shows the source code for the game, which includes functions for displaying the board, placing ships, and handling player input. The terminal shows the game running, selecting a saved game, displaying both players' boards, and then saving the game.

Ejercicio N°27 – Diccionarios en Python de un producto. -

Description:

Este código muestra cómo recorrer un **diccionario en Python** que representa un producto con atributos como código, nombre, precio y cantidad. Primero, se imprimen las **claves del diccionario** usando un bucle (for), lo que permite ver solo los nombres de los campos del producto. Luego, se utiliza el método `ítems()` para iterar sobre **pares clave-valor**, mostrando tanto el campo como su contenido, con un formato legible (Nombre: IPhone, por ejemplo). El uso de funciones como `capitalize()` mejora la presentación de las salidas, y el bucle (for) es esencial para procesar estructuras de datos tipo diccionario, útiles en aplicaciones como inventarios, bases de datos o catálogos de productos. Este ejemplo ayuda a comprender cómo acceder, manipular y mostrar información almacenada de forma estructurada y legible.

The screenshot shows a Python code editor interface with the following details:

- Code Editor:** The main window displays the `class12_Diccionarios.py` file. The code defines a product dictionary, iterates over its keys, and then its items (key-value pairs) using `items()`. It uses `capitalize()` to format the output. The code also creates an inventory list and appends various product dictionaries to it.
- Terminal:** A terminal window to the right shows the execution of the script: `~/workspace$ python class12_Diccionarios.py`. The output is:
 - Claves del producto ---
 - Código
 - Nombre
 - Precio
 - Cantidad
 -
 - Clave y valor del producto ---
 - Código: 100
 - Nombre: IPhone
 - Precio: 1200.0
 - Cantidad: 10
 - Fabrizio Lora (FaXx@)
- File Explorer:** On the left, there's a file explorer showing other files like `C_E-G_A CINE.py`, `Batalla_Naval.py`, and `BatallaNaval_mejorada.py`.

Ejercicio N°28 – Diccionario de Python con varios productos sistema de inventario pequeño. -

Descripción:

Este código en Python simula un pequeño **sistema de inventario** utilizando listas y diccionarios. Cada producto (como un pantalón, detergente o iPhone) está representado mediante un diccionario que contiene campos como código, nombre, precio y cantidad. Los productos se agregan a una lista llamada inventario, que funciona como una colección de todos los artículos. Luego, mediante un bucle (for), el programa recorre cada producto de la lista y, dentro de otro bucle, muestra sus claves y valores de forma ordenada y legible. Este tipo de estructura es útil para gestionar catálogos, listar productos o manejar almacenes de manera sencilla en Python. Es importante notar que algunas variables como comida, limpieza y ropa se sobrescriben, lo que significa que solo el último valor definido se conserva.

The screenshot shows a code editor interface with several files open. The main file is 'class12_Diccionarios.py'. The code defines a dictionary 'inventario' containing various products like comida, tecnología, celulares, limpieza, ropa, and canciones. It then iterates over the inventory to print each item's key-value pairs. The output is shown in a terminal window, displaying four items: Pan (precio: 5.00, cantidad: 40), Iphone (precio: 1200.00, cantidad: 10), detergente (precio: 30.00, cantidad: 10), and zapatos (precio: 270.00, cantidad: 10). The code editor also shows imports for 'C_E=G_A CINE.py', 'Batalla_Naval.py', 'BatallaNaval_mejorada.py', 'usuarios.json', and 'class12_Diccionarios.py'.

```
10 print ("Fabrizio Lora (FaXx0)")
11 #-----#
12 #-----#
13 #-----#
14 # Diccionario de productos
15 inventario =[]
16 comida = {'codigo' : 123, 'nombre' : 'frutas', 'precio' : 15.00, 'cantidad' : 10}
17 comida = {'codigo' : 124, 'nombre' : 'Pan', 'precio' : 5.00, 'cantidad' : 40}
18 tecnologia = {'codigo' : 456, 'nombre' : 'Laptop', 'precio' : 6500.00, 'cantidad' : 10}
19 celulares = {'codigo': 457, 'nombre': 'Iphone', 'precio': 4200.00, 'cantidad': 5}
20 limpieza = {'codigo' : 789, 'nombre' : 'jabon', 'precio' : 18.00, 'cantidad' : 10}
21 ropa = {'codigo' : 101112, 'nombre' : 'detergente', 'precio' : 30.00, 'cantidad' : 10}
22 ropa = {'codigo' : 101113, 'nombre' : 'camisa', 'precio' : 150.00, 'cantidad' : 10}
23 ropa = {'codigo' : 101114, 'nombre' : 'pantalon', 'precio' : 150.00, 'cantidad' : 10}
24 ropa = {'codigo' : 101114, 'nombre' : 'zapatos', 'precio' : 270.00, 'cantidad' : 10}
25 # Diccionario de producto
26 inventario.append(comida) # Agrega el diccionario de comida al inventario
27 inventario.append(producto) # Agrega el diccionario de producto al inventario
28 inventario.append(limpieza) # Agrega el diccionario de limpieza al inventario
29 inventario.append(ropa) # Agrega el diccionario de ropa al inventario
30 for producto in inventario: # Iteración sobre los productos del inventario
31     print("\n--- Clave y valor del producto ---")
32     for clave, valor in producto.items(): # Iteración sobre las claves y valores del diccionario
33         valor = producto [clave] # Obtiene el valor de la clave
34         print(f'{clave.capitalize()}: {valor}')
35     # Fin del producto
36 print ("Fabrizio Lora (FaXx0)")
37
38 #-----#
39 #-----#
40
41 # Diccionario para una Canción
42 canción = {
43     "título": "Shape of You",
44 }
```

Ejercicio N°29 – Diccionario de Python con una canción, coche y una red social. –

Descripción:

Este código en Python presenta tres ejemplos de estructuras de datos utilizando **diccionarios** para representar distintos objetos del mundo real: una canción, un coche y una publicación en una red social. Cada diccionario almacena información clave-valor, permitiendo organizar datos como el título, artista y colaboradores de una canción; los detalles técnicos de un automóvil como tipo de motor y transmisión; y los metadatos de una publicación en redes sociales como autor, contenido, fecha y número de likes. El uso de diccionarios permite estructurar la información de forma clara y accesible, y se imprime cada uno de ellos por separado para su visualización. Este tipo de estructuras es muy útil para representar datos complejos de forma organizada y reutilizable en programas más grandes.

```
class12_Diccionarios.py > ...
42     "cancion" = {
43         "titulo": "Shape of You",
44         "artista": "Ed Sheeran",
45         "album": "Divide",
46         "duracion_segundos": 233,
47         "genero": "Pop",
48         "fecha_lanzamiento": ["2017-01-06"],
49         "colaboradores": ["Johnny McDaid", "Steve Mac"]
50     }
51     # Diccionario para un Coche
52     "coche" = {
53         "marca": "Toyota",
54         "modelo": "Corolla",
55         "año": 2020,
56         "color": "Negro",
57         "placa": "ABC-1234",
58         "caracteristicas": {
59             "tipo_motor": "Gasolina",
60             "potencia_hp": 132,
61             "transmision": "Automática"
62         }
63     }
64     # Diccionario para un Post de Red Social
65     "post_red_social" = {
66         "id_post": 987654321,
67         "autor": "user.py",
68         "contenido_texto": "¡Learning python!",
69         "fecha_publicacion": "2025-06-23 15:30:00",
70         "likes": 23,
71     }
72     # Imprimir
73     print("Canción:")
74     print(cancion)
75     print("\nCoche:")
76     print(coche)
77     print("\nPost de Red Social:")
78     print(post_red_social)

--- Clave y valor del producto ---
Codigo: 101114
Nombre: zapatos
Precio: 279.0
Cantidad: 10
Imagen: Lora (FaXx0)
Cancion:
{
    "titulo": "Shape of You",
    "artista": "Ed Sheeran",
    "album": "Divide",
    "duracion_segundos": 233,
    "genero": "Pop",
    "fecha_lanzamiento": ["2017-01-06"],
    "colaboradores": ["Johnny McDaid", "Steve Mac"]
}
Coche:
{
    "marca": "Toyota",
    "modelo": "Corolla",
    "año": 2020,
    "color": "Negro",
    "placa": "ABC-1234",
    "caracteristicas": {
        "tipo_motor": "Gasolina",
        "potencia_hp": 132,
        "transmision": "Automática"
    }
}
Post de Red Social:
{
    "id_post": 987654321,
    "autor": "user.py",
    "contenido_texto": "¡Learning python!",
    "fecha_publicacion": "2025-06-23 15:30:00",
    "likes": 23
}

New tab >
Search for files & tools...
Files Find a file >
Q Search Search through your files >
+ New file Create a new file >
Files
edad-cine-completo.py Recently opened
tabla_multiplicar.py Recently opened
Adivina_el_numero.py Recently opened
Refactorizar.py Recently opened
```

Ejercicio N°30 – Gestión de Tareas. –

Descripción:

Este programa en Python implementa una aplicación básica de **gestión de tareas (To-Do List)** usando listas y diccionarios. Cada tarea se representa como un diccionario con campos como ID, título, descripción, prioridad y estado de finalización. El código incluye funciones para **agregar, mostrar, buscar, completar y eliminar tareas**, lo cual permite mantener organizada la lista de pendientes. También se utiliza un bucle (while) que presenta un **menú interactivo** en consola, facilitando la interacción con el usuario. La identificación única de cada tarea se maneja con un ID incremental, y se aprovechan estructuras como condicionales y bucles (for) para recorrer la lista. Esta solución es útil para aprender a gestionar colecciones de datos y aplicar lógica secuencial en Python.

The screenshot shows a Python development environment with several tabs open:

- File: prog-ejer-prac-class
- Code: arada.py (active tab)
- Code: class12_Diccionarios.py
- Code: usuarios.json
- Code: class13_Todolist.py
- Code: bat1

The arada.py file contains code for a todo list application:

```
#!/usr/bin/python3
# Ejercicio Práctico Clase 13: Manejo de diccionarios y listas

# Función para agregar una tarea a la lista de tareas
def agregar_tarea(descripción, prioridad="media"):
    global proximo_id_tarea # Declaramos la variable global
    tarea = { # Creamos un diccionario con los datos de la tarea
        "id": proximo_id_tarea,
        "título": descripción, # Usamos la descripción como título
        "descripción": descripción,
        "prioridad": prioridad,
        "completada": False
    }
    lista_tareas.append(tarea)
    proximo_id_tarea += 1
    print(f"La tarea '{descripción}' agregada con éxito.")

# Función para mostrar las tareas
def mostrar_tareas():
    print("\n--- Lista de tareas ---")
    if not lista_tareas:
        print("No hay tareas en la lista.")
        return
    for tarea in lista_tareas:
        estado = "Completada" if tarea["completada"] else "Pendiente"
        print(f"ID: {tarea['id']} | {tarea['título']}\n{tarea['prioridad']}")

# Agregar tarea
agregar_tarea("Estudiar para el examen de Cálculo")
agregar_tarea("Hacer la compra de supermercado", "alta")
agregar_tarea("Llamar al médico", "baja")
# Mostrar tareas
mostrar_tareas()

# Buscar tarea por id
def buscar_tarea_por_id(id_buscado):
    for tarea in lista_tareas:
        if tarea["id"] == id_buscado:
            return tarea
    return None
```

The terminal window shows the application's output:

```
Arada@Arada-OptiPlex-5090:~/Desktop$ python arada.py
--- Lista de tareas ---
No hay tareas en la lista.
ID: 1 | Estudiar para el examen de Cálculo (media) - Pendiente
ID: 2 | Hacer la compra de supermercado (alta) - Pendiente
ID: 3 | Llamar al médico (baja) - Pendiente
Tarea 'Estudiar para el examen de Cálculo' agregada con éxito.
Tarea 'Hacer la compra de supermercado' agregada con éxito.
Tarea 'Llamar al médico' agregada con éxito.

--- Lista de tareas ---
ID: 1 | Estudiar para el examen de Cálculo (media) - Pendiente
ID: 2 | Hacer la compra de supermercado (alta) - Pendiente
ID: 3 | Llamar al médico (baja) - Pendiente
Tarea 'Estudiar para el examen de Cálculo' marcada como completada.

--- Lista de tareas ---
ID: 1 | Estudiar para el examen de Cálculo (media) - Completada
ID: 2 | Hacer la compra de supermercado (alta) - Pendiente
ID: 3 | Llamar al médico (baja) - Pendiente
Tarea 'Hacer la compra de supermercado' eliminada.

--- Lista de tareas ---
ID: 1 | Estudiar para el examen de Cálculo (media) - Completada
ID: 3 | Llamar al médico (baja) - Pendiente
Error: No se encontró la tarea con ID 99.

===== MENU 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: 0
```

Ejercicio N°31 – Agenda Contactos. –

Descripción:

Este programa en Python implementa un **Gestor de Contactos con Interfaz Gráfica (GUI)** utilizando el módulo tkinter, junto con estructuras de datos orientadas a objetos mediante clases. Se definen dos clases principales: Contacto, que representa a cada persona con atributos como ID, nombre, teléfonos y correo electrónico; y Gestor Contactos, que se encarga de administrar la lista de contactos, incluyendo funciones para agregar, editar, eliminar, buscar, guardar y cargar datos en formato JSON. La interfaz gráfica permite al usuario realizar estas acciones de forma intuitiva, ingresando información mediante campos de texto y botones. Las operaciones se reflejan en una lista visual (Listbox), y se actualizan automáticamente cada vez que el usuario modifica los contactos. Este tipo de aplicación es útil para aprender conceptos como programación orientada a objetos, manejo de archivos JSON y creación de interfaces gráficas básicas en Python.

The screenshot shows the PyCharm IDE interface. On the left, the code editor displays Python code for a contact manager application. The code includes methods for adding contacts, saving them to a JSON file, and deleting them. On the right, a separate window titled "Gestor de Contactos 2.0" is running, showing a simple GUI with fields for Nombre, Email, and Teléfonos (separated by comma), and buttons for Agregar, Editar, and Eliminar.

```
13.Todolist.py  batalla.naval.save.json  permisos ssh.md  AgendaContacto2.0.py  +  Run  +  Shell  VNC  +  Invite  Deploy  ...  
AgendaContacto2.0.py > Contacto > f_to_dict  
Format  
• Gestor de Contactos 2.0  
Nombre  
Email  
Teléfonos (separados por coma)  
Agregar  
Editar  
Eliminar  
+  At (-) Python Diff  In 18, Col 18 • Spaces:4 History
```

The screenshot shows a Python development environment with the following details:

- Code Editor:** The main pane displays the `AgendaContacto2.0.py` script. It defines a `Contacto` class with methods like `agregar_telefono` and `to_dict`, and a `GestorContactos` class that interacts with a JSON file and provides a GUI interface.
- Graphical Application:** A window titled "Gestor de Contactos 2.0" is open, showing a form with fields for Nombre, Email, and Teléfonos (separados por coma). Below the form is a list of contacts, with the first entry being "1 - Faxxo | Tels: 67846463, 5768898 | Email: snxwsxulles@outlook.com". Buttons for Agregar, Editar, and Eliminar are visible at the bottom.
- Status Bar:** The status bar at the bottom indicates "Ln 18, Col 18" and "Spaces: 4".

The screenshot shows the same Python development environment after changes have been made to the code:

- Code Editor:** The `AgendaContacto2.0.py` script now includes additional methods like `guardar` and `cargar` for interacting with a file named `ARCHIVO`. The `actualizar_lista` method is also present.
- Graphical Application:** The same "Gestor de Contactos 2.0" window is shown, displaying the same contact list and interface elements.
- Status Bar:** The status bar at the bottom indicates "Ln 18, Col 18" and "Spaces: 4".