

Practico I

Dead Services

Integrantes:

Jose Alejandro Zabala Romero

Leonardo Montenegro

Junior Pacajes Banegas

Nicodemus Berry Quiroga Gonzales

Luis Antonio Moreno Miranda

Lucas Mateo Diaz Badani

Richard Iver Hurtado Cotocari

Materia: Programacion II

Docente: Jimmy Nataniel Requena

Índice

1. Gestión de Proyecto y Git	4
1.1 Creación de carpeta y el push en repl.it.....	4
2. Ejercicios de Introducción	5
2.1 Hola mundo.....	5
2.2 Saludar persona	5
2.3 clasificacion de peliculas	6
2.4 Bienvenidas de integrantes al grupo.....	6
3. Funciones Simples	7
3.1 Dos Funciones (Saludo y Suma)	7
3.2 Suma de dos valores	7
3.3 Encontrar el mayor.....	8
3.4 Defactorizar (Área de rectángulos)	8
3.5 Cervecería (Generador de nombres)	9
4. Comidas Favoritas (Modularización de Funciones)	10
4.1 Comidas Favoritas.....	10
4.2 Funciones	11
4.3 Main - Módulo Externo	12
5. Tablas de Multiplicación.....	13
5.1 Tabla de Multiplicar.....	13
5.2 Tabla de Multiplicar con Funciones y Pruebas.....	13
5.3 Tabla de Multiplicar con Límite Configurable.....	14
6. Juegos de Adivinanza (Número Secreto).....	14
6.1 Número Secreto	14
6.2 Número Oculto con getpass	15
6.3 Adivinanza múltiple.....	16
7. Operaciones con Notas y Listas	17
7.1 Suma de Notas y Promedio Manual.....	17
7.2 Suma de Elementos.....	18
7.3 Suma Pythónica con sum().....	18
7.4 Append vs Insert	19
8. Contar Elementos	19
8.1 Contar un Elemento Específico	19
8.2 Contar con count().....	20

9. Factorial (Varias Implementaciones)	20
9.1 Factorial Recursivo.....	20
9.2 Factorial con for	21
9.3 Factorial con math.prod.....	21
9.4 Factorial con reduce	22
10. For Tradicional vs Lista por Comprensión	22
10.1 Comparación entre for clásico y forma "pytónica".....	22
11. Invertir Cadenas y Listas	23
11.1 Invertir Texto con for y range.....	23
11.2 Invertir Lista con bucle	23
11.3 Invertir Lista con Slicing	24
12. Búsqueda y Ordenamiento	25
12.1 Búsqueda Lineal y Binaria.....	25
12.2 Ordenamiento Mergesort	26
12.3 Ordenamiento Burbuja e Inserción	28
12.4 Ordenamiento con Heap Sort	29
13. Vector Misterioso (Lista Secreta Interactiva)	30
13.1 Lista Secreta	30
13.2 Lista Secreta con Validación de Longitud	31
13.3 Lista Secreta con getpass.....	33
13.4 Lista Secreta con opción "Lo siento.....	35

1. Gestión de Proyecto y Git

1.1 Creación de carpeta y el push en replit

```

~/workspace/prog$ git push
error: failed to push some refs to *https://github.com/leon4rdmtg/prog2.git*
hint: Updates were rejected because the remote contains work that you do not
hint: have locally. This is usually caused by another repository pushing to
hint: the same ref. If you want to integrate the remote changes, use
hint: git pull --rebase.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
~/workspace/prog$ git pull --rebase
error: cannot pull with rebase: You have unstaged changes.
error: Please commit or stash them.
~/workspace/prog$ git add .
hint: If you wanted to say 'git add .?
hint: Disable this message with "git config advice.addEmptyPathspec false"
~/workspace/prog$ git add .
~/workspace/prog$ git commit -m "Guardo cambios locales antes del pull"
[master 5576f31..474ef31] Guardo cambios locales antes del pull
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 Jose_Alejandro/alejandro.md
~/workspace/prog$ git pull --rebase
remote: Enumerating objects: 61, done.
remote: Counting objects: 100% (61/61), done.
remote: Compressing objects: 100% (38/38), done.
remote: Total 56 (delta 15), reused 35 (delta 7), pack-reused 0 (from 0)
Unpacking objects: 100% (56/56), 13.32 KB | 909.00 KB/s, done.
From https://github.com/leon4rdmtg/prog2
 * branch            main      -> origin/main
Successfully rebased and updated refs/heads/main.
~/workspace/prog$ git push
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 4 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (6/6), 646 bytes | 646.00 KB/s, done.
Total 6 (delta 2), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (2/2), completed with 1 local object.
To https://github.com/leon4rdmtg/prog2.git
 474ef31..20f8fd main -> main
~/workspace/prog$ 

```

Se pudo hacer el push de los cambios correctamente en replit.

Author	Commit Message	Date
Antonio-m2	15-057	17 hours ago
Antonio	15-057	17 hours ago
Junior	xd	18 hours ago
Lucas	xd	17 hours ago
Richard	Agregando carpeta Richard al submodulo	17 hours ago
alejandro	subo ejercicios y carpetas dentro de alejandro	5 days ago
asdfa	Guardo cambios locales antes del pull	18 hours ago
Adivina_numero.py	Guardo cambios locales antes del pull	18 hours ago
LUCAS MATEO DIAZ BADANI	agrega carpeta ComidaFavoritas	5 days ago
README.md	Update README.md	last week
Richard-HEAD	Guardo cambios locales antes del pull	18 hours ago
hurtado.md	Update hurtado.md	last week

Al crear las carpetas y subir los archivos al repositorio, se aprendió lo que es hacer el push correctamente. Permite que los cambios realizados localmente se reflejen en el repositorio remoto, lo que asegura que todos los miembros del equipo tengan acceso a las últimas actualizaciones. Es bueno revisar bien lo que se ha modificado y hacer un commit claro antes de hacer el push, para evitar subir cambios innecesarios o desordenados. Así, el trabajo en equipo se mantiene organizado y fluido, sin confusiones ni conflictos en el código.

2. Ejercicios de Introducción

2.1 Hola mundo

Code Blame 2 lines (2 loc) · 83 Bytes GitHub Copilot

```
1 print ("hola mundo")
2 print ("fin del programa ---- Jose Alejandro Zabala Romero")
```

Prueba

```
~/workspace$ cd proyz/alejanuro/practico_1/Ejercicios
~/.../Practico_I/Ejercicios$ python hola_mundo.py
hola mundo
fin del programa ---- Jose Alejandro Zabala Romero
~/.../Practico_I/Ejercicios$
```

2.2 Saludar persona

Code Blame 5 lines (5 loc) · 152 Bytes GitHub Copilot

```
1 def saludar(nombre):
2     mensaje = f"Hola, {nombre}!"
3     print(mensaje)
4 saludar("Alejandro")
5 print("Fin del programa ---- Jose Alejandro Zabala Romero")
```

Prueba

```
~/.../Practico_I/Ejercicios$ python saludar.py
Hola, Alejandro!
Fin del programa ---- Jose Alejandro Zabala Romero
~/.../Practico_I/Ejercicios$
```

El código define una función llamada saludar que recibe un parámetro nombre. Dentro de la función, se utiliza una **f-string** para construir un mensaje personalizado del tipo "Hola, [nombre]!", el cual luego se imprime en pantalla. A continuación, se llama a la función con el argumento "Alejandro", por lo que el programa muestra el saludo: Hola, Alejandro!.

2.3 clasificacion de peliculas

```

1 def clasificar_peliculas(edad):
2     """
3         Devuelve una recomendación de clasificación de películas según la edad.
4         Clasificaciones:
5             - R: Restricted (Restringida) → Solo mayores de 17 años o con acompañante adulto.
6             - PG-13: Parental Guidance Suggested for under 13 → Guía parental sugerida, no recomendable para menores de 13.
7             - G: General Audience → Apta para todo público.
8             - PG: Parental Guidance → Algunas escenas pueden no ser aptas para niños pequeños.
9         """
10        if edad < 0:
11            return "Edad no válida."
12        elif edad < 13:
13            return "Te recomendamos películas clasificadas G o PG."
14        elif edad < 18:
15            return "Puedes ver películas clasificadas PG-13."
16        else:
17            return "¡Puedes ver películas clasificadas R!"
18    # ===== Pruebas unitarias =====
19    assert clasificar_peliculas(20) == "¡Puedes ver películas clasificadas R!", "Error para edad 20"
20    assert clasificar_peliculas(15) == "Puedes ver películas clasificadas PG-13.", "Error para edad 15"
21    assert clasificar_peliculas(10) == "Te recomendamos películas clasificadas G o PG.", "Error para edad 10"
22    assert clasificar_peliculas(-5) == "Edad no válida.", "Error para edad negativa"
23    print("Pruebas unitarias superadas.")
24    # ===== Verificador interactivo de edades =====
25    while True:
26        try:
27            edad = int(input("Ingresa tu edad: "))
28            resultado = clasificar_peliculas(edad)
29            print(f" {resultado}")
30        except ValueError:
31            print(" Debes ingresar un número entero válido.")
32            continue
33
34        continuar = input("¿Deseas verificar otra edad? (Y/N): ").strip().lower()
35        if continuar != 'y':
36            break
37    print("--- Fin del programa --- Richard Hurtado")
38
39

```

Prueba

```

~/.../Practico_I/Ejercicios$ python clasif_peliculas.py
Pruebas unitarias superadas.
Ingresa tu edad: 20
¡Puedes ver películas clasificadas R!
¿Deseas verificar otra edad? (Y/N): no
--- Fin del programa --- Richard Hurtado
~/.../Practico_I/Ejercicios$ █

```

Este programa clasifica películas según la edad del usuario. Incluye pruebas automáticas para validar el comportamiento y permite verificar edades de forma interactiva.

2.4 Bienvenidas de integrantes al grupo

```

1
2 nombres_estudiantes = ['Leonardo', 'Richard', 'Antonio', 'Junior', 'Nico', 'Mateo']
3 for nombre in nombres_estudiantes:
4     print(f"¡Bienvenido al equipo, {nombre}!")
5
6 print("Fin del programa-----Jose Alejandro Zabala Romero")

```

Este código muestra un mensaje de bienvenida para cada estudiante en una lista. Se utiliza un bucle for para recorrer la lista y una f-string para personalizar el saludo.

Prueba

```
~.../Practico_I/Ejercicios$ python bienbenida_equip.py
¡Bienvenido al equipo, Leonardo!
¡Bienvenido al equipo, Richard!
¡Bienvenido al equipo, Antonio!
¡Bienvenido al equipo, Junior!
¡Bienvenido al equipo, Nico!
¡Bienvenido al equipo, Mateo!
Fin del programa-----Jose Alejandro Zabala Romero
~.../Practico_I/Ejercicios$
```

3. Funciones Simples

3.1 Dos Funciones (Saludo y Suma)

```
1 def saludar(nombre_persona):
2     """
3     Recibe un nombre y devuelve un saludo personalizado (modificada para pruebas).
4     """
5     mensaje = f"¡Hola, {nombre_persona}! ¡Qué bueno tenerte aquí!"
6     return mensaje
7 def sumar(a, b):
8     """
9     Recibe dos números y devuelve su suma.
10    """
11    return a + b
12 # ===== Pruebas unitarias con assert =====
13 assert saludar("Jose") == "¡Hola, Jose! ¡Qué bueno tenerte aquí!", "Error en saludo para Jose"
14 assert saludar("Alejandro") == "¡Hola, Alejandro! ¡Qué bueno tenerte aquí!", "Error en saludo para Alejandro"
15 assert sumar(2, 3) == 5, "Error: 2 + 3 debe ser 5"
16 assert sumar(-4, 4) == 0, "Error: -4 + 4 debe ser 0"
17 print(" Todas las pruebas unitarias pasaron correctamente.\n")
18 # ===== Llamadas a las funciones =====
19 print(saludar("Jose"))
20 print(f"La suma de 7 y 8 es: {sumar(7, 8)}")
21 print("Fin del programa----- Richard Hurtado")
```

Este programa contiene dos funciones: una que devuelve un saludo personalizado y otra que suma dos números. Se validan con pruebas unitarias (assert) y luego se llaman para mostrar sus resultados.

Prueba

```
~.../Practico_I/Ejercicios$ python 2funciones.py
Todas las pruebas unitarias pasaron correctamente.

¡Hola, Jose! ¡Qué bueno tenerte aquí!
La suma de 7 y 8 es: 15
Fin del programa----- Richard Hurtado
~.../Practico_I/Ejercicios$
```

3.2 Suma de dos valores

```
progs / alejandro / Practico_I / Ejercicios / suma_Ayo.py
1 def sumar(a, b):
2     """
3     Recibe dos números y devuelve su suma.
4     """
5     resultado_suma = a + b
6     return resultado_suma
7 # ===== Pruebas unitarias con assert =====
8 assert sumar(2, 3) == 5, "Error: 2 + 3 debe ser 5"
9 assert sumar(-1, 1) == 0, "Error: -1 + 1 debe ser 0"
10 assert sumar(10, 0) == 10, "Error: 10 + 0 debe ser 10"
11 print(" Pruebas unitarias para sumar() superadas.")
12
13 print("Fin del programa-----Leonardo Montenegro ")
```

La función sumar recibe dos números, calcula y devuelve su suma. Se incluyen pruebas unitarias para verificar que la función funciona correctamente.

Prueba

```
~/.../Práctico_I/Ejercicios$ python suma_AyB.py
Pruebas unitarias para sumar() superadas.
Fin del programa-----Leonardo Montenegro
~/.../Práctico_I/Ejercicios$ █
```

3.3 Encontrar el mayor

```
1  # Definición de la función
2  def encontrar_mayor(lista_numeros):
3      # Caso especial: lista vacía
4      if not lista_numeros:
5          return None
6      # Paso 1: El primer "campeón"
7      mayor_temporal = lista_numeros[0]
8      # Paso 2-4: Recorrer la lista para buscar al más grande
9      for elemento in lista_numeros:
10         if elemento > mayor_temporal:
11             mayor_temporal = elemento
12     # Paso 5: Devolver el campeón
13     return mayor_temporal
14 # -----
15 # Casos de prueba con assert
16 #
17 print("Probando encontrar_mayor...")
18 assert encontrar_mayor([1, 5, 3, 9, 2]) == 9
19 assert encontrar_mayor([-10, -5, -3, -20]) == -3
20 assert encontrar_mayor([7, 7, 7, 7]) == 7
21 assert encontrar_mayor([]) == None # lista vacía
22 assert encontrar_mayor([42]) == 42 # un solo elemento
23 print("¡Pruebas para encontrar_mayor pasaron!")
24 print("Fin del programa-----Leonardo Montenegro ")
```

La función encontrar_mayor recibe una lista de números y devuelve el mayor valor. Si la lista está vacía, retorna None. Se incluyen pruebas con assert para validar su correcto funcionamiento.

Prueba

```
~/.../Práctico_I/Ejercicios$ python encontrar-may.py
Probando encontrar_mayor...
¡Pruebas para encontrar_mayor pasaron!
Fin del programa-----Leonardo Montenegro
~/.../Práctico_I/Ejercicios$ █
```

3.4 Defactorizar (Área de rectángulos)

```
1  # Refactorizar el código de la tabla de multiplicar
2  # Función para calcular áreas de rectángulos
3  def calcular_area_rectangulo(base, altura):
4      return base * altura
5      # Fin de la función
6  # Función principal con indentación FIXED
7  def mostrar_area_rectangulo(numero, base, altura): # Función principal
8      area = calcular_area_rectangulo(base, altura) # o area = base * altura
9      # Fin de la función
10     print(f"El área del rectángulo {numero} ({base}x{altura}) es: {area}") # Salida de datos
11     # Fin de la función
12  # Rectángulos
13  # Rectángulo 1 (usando la función) # Llamada a la función principal con los parámetros correspondientes
14  mostrar_area_rectangulo(1, 10, 5)
15  # Rectángulo 2 (usando la función) # Llamada a la función principal con los parámetros correspondientes
16  mostrar_area_rectangulo(2, 7, 3)
17  # Rectángulo 3 (usando la función) # Llamada a la función principal con los parámetros correspondientes
18  mostrar_area_rectangulo(3, 15, 8)
19
20 print("Fin del programa-----Richard Hurtado ")
```

El programa calcula y muestra el área de varios rectángulos. Usa una función para calcular el área y otra para mostrar el resultado formateado.

Prueba

```
~/. . . /Práctico_I/Ejercicios$ python defactorizar.py
El área del rectángulo 1 (10x5) es: 50
El área del rectángulo 2 (7x3) es: 21
El área del rectángulo 3 (15x8) es: 120
Fin del programa-----Richard Hurtado
~/. . . /Práctico_I/Ejercicios$
```

3.5 Cervecería (Generador de nombres)

```
1  def crear_nombre_cerveceria():
2      print("🍺 Bienvenido al generador de nombres para tu cervecería 🍺")
3
4      palabra1 = input("🍺 Ingresá la primera palabra: ").strip()
5      palabra2 = input("🍺 Ingresá la segunda palabra: ").strip()
6
7      # Op 1: Palabras unidas
8      nombre_completo = palabra1 + palabra2
9
10     # Op2: Combinar 3 letras de cada palabra
11     parte1 = palabra1[:3] if len(palabra1) >= 3 else palabra1
12     parte2 = palabra2[:3] if len(palabra2) >= 3 else palabra2
13     nombre_corto = parte1 + parte2
14
15     print("\n ¡Felicitaciones! Aquí tienes dos ideas para tu cervecería:\n")
16     print(f" Nombre combinado: {nombre_completo}")
17     print(f" Nombre corto (3+3 letras): {nombre_corto}")
18
19     crear_nombre_cerveceria()
20     print ("----fin del porgrama---- Jose Alejandro Zabala Romero")
```

Prueba

```
~/. . . /Práctico_I/Ejercicios$ python cerveceria.py
🍺 Bienvenido al generador de nombres para tu cervecería 🍺
🍺 Ingresá la primera palabra: Jumper
🍺 Ingresá la segunda palabra: scrit

¡Felicitaciones! Aquí tienes dos ideas para tu cervecería:

Nombre combinado: Jumperscrit
Nombre corto (3+3 letras): Jumscri
----fin del porgrama---- Jose Alejandro Zabala Romero
~/. . . /Práctico_I/Ejercicios$
```

Muestra un mensaje de bienvenida decorado. Luego, solicita al usuario que ingrese dos palabras, las cuales son limpiadas de espacios extra con el método `.strip()` para asegurar un formato limpio. Con esas palabras, se generan dos versiones del posible nombre. La primera une ambas palabras directamente, creando un nombre completo. La segunda opción construye un nombre más corto combinando las primeras tres letras de cada palabra. Si alguna palabra tiene menos de tres letras, se toma lo que esté disponible.

4. Comidas Favoritas (Modularización de Funciones)

4.1 Comidas Favoritas

```

1 comidas_favoritas = ['Pizza', 'hamburgueza', 'broaster']
2 def imprimir_lista(lista):
3     print("\nLista de comidas favoritas:")
4     for i, comida in enumerate(lista):
5         print(f"[{i + 1}]. {comida}")
6     print()
7 def modificar_lista(lista):
8     imprimir_lista(lista)
9     try:
10         indice = int(input("¿Qué número de comida deseas modificar? (1, 2, 3...): ")) - 1
11         if 0 <= indice < len(lista):
12             nuevo_valor = input("¿Cuál es la nueva comida favorita?: ")
13             lista[indice] = nuevo_valor
14             print("¡Comida actualizada con éxito!")
15         else:
16             print(" Índice fuera de rango.")
17     except ValueError:
18         print(" Por favor, ingresa un número válido.")
19 def menu():
20     while True:
21         print("\n--- MENÚ ---")
22         print("1. Ver lista de comidas favoritas")
23         print("2. Modificar una comida")
24         print("3. Salir")
25         opcion = input("Selecciona una opción (1/2/3): ")
26         if opcion == '1':
27             imprimir_lista(comidas_favoritas)
28         elif opcion == '2':
29             modificar_lista(comidas_favoritas)
30         elif opcion == '3':
31             print("👋 ¡Hasta luego!")
32             break
33         else:
34             print(" Opción no válida. Intenta de nuevo.")
35     menu()
36 print ("Lucas Mateo Diaz Badani - FIN DEL PROGRAMA")
37

```

Este código permite gestionar una lista de comidas favoritas de manera interactiva. Al inicio, se define una lista con tres elementos: 'Pizza', 'hamburgueza' y 'broaster'. Luego se crean tres funciones principales para trabajar con esa lista.

La primera función, `imprimir_lista`, recibe una lista como parámetro y muestra su contenido en pantalla con un formato ordenado, usando `enumerate` para numerar cada comida.

La segunda función, `modificar_lista`, también recibe una lista, muestra los elementos actuales llamando a `imprimir_lista`, y luego le pregunta al usuario cuál de las comidas quiere modificar. Si el número ingresado es válido, reemplaza ese elemento por uno nuevo que el usuario escriba. Si el número está fuera de rango o no es válido, muestra un mensaje de error adecuado.

La tercera función, `menu`, es un bucle interactivo que muestra un menú con tres opciones: ver la lista, modificar una comida, o salir del programa. Según lo que elija el usuario, se llama a la función correspondiente o se termina el programa mostrando un mensaje de despedida, finalmente, se llama a `menu()` para iniciar todo el proceso.

Prueba

```

--- MENÚ ---
1. Ver lista de comidas favoritas
2. Modificar una comida
3. Salir
Selecciona una opción (1/2/3): 1

Lista de comidas favoritas:
1. Pizza
2. hamburgueza
3. broaster

--- MENÚ ---
1. Ver lista de comidas favoritas
2. Modificar una comida
3. Salir
Selecciona una opción (1/2/3): 2

Lista de comidas favoritas:
1. Pizza
2. hamburgueza
3. broaster

¿Qué número de comida deseas modificar? (1, 2, 3...): 3
¿Cuál es la nueva comida favorita?: picante
¡Comida actualizada con éxito!

--- MENÚ ---
1. Ver lista de comidas favoritas
2. Modificar una comida
3. Salir
Selecciona una opción (1/2/3): 1

Lista de comidas favoritas:
1. Pizza
2. hamburgueza
3. picante

--- MENÚ ---
1. Ver lista de comidas favoritas
2. Modificar una comida
3. Salir
Selecciona una opción (1/2/3): 3
👋 ¡Hasta luego!
Lucas Mateo Díaz Badani - FIN DEL PROGRAMA
~/.../Práctico_I/comidas_fav$ █

```

4.2 Funciones

```

1 def imprimir_lista(lista):
2     print("\nLista de comidas favoritas:")
3     for i, comida in enumerate(lista):
4         print(f"{i + 1}. {comida}")
5     print()
6
7 def modificar_lista(lista):
8     imprimir_lista(lista)
9     try:
10         indice = int(input("¿Qué número de comida deseas modificar? (1, 2, 3...): ")) - 1
11         if 0 <= indice < len(lista):
12             nuevo_valor = input("¿Cuál es la nueva comida favorita?: ")
13             lista[indice] = nuevo_valor
14             print("¡Comida actualizada con éxito!")
15         else:
16             print(" Índice fuera de rango.")
17     except ValueError:
18         print(" Por favor, ingresa un número válido.")
19
20 print ("Richard Hurtado - FIN DEL PROGRAMA")

```

Este código define dos funciones que trabajan con una lista de comidas favoritas. La función `imprimir_lista` muestra los elementos de la lista numerados de forma clara, utilizando `enumerate` para facilitar la lectura.

La función `modificar_lista` permite al usuario elegir qué comida desea cambiar según su número en la lista. Si el índice es válido, se actualiza con el nuevo valor. Si no, se muestra un mensaje de error, ya sea por un número fuera de rango o por ingresar un valor no numérico. En esta versión no se llama a ninguna, por lo que no se ejecutan a menos que se llamen manualmente.

4.3 Main - Módulo Externo

```

1  from funciones import imprimir_lista, modificar_lista
2  comidas_favoritas = ['Pizza', 'hamburgueza', 'salteñas']
3  def menu():
4      while True:
5          print("\n--- MENÚ ---")
6          print("1. Ver lista de comidas favoritas")
7          print("2. Modificar una comida")
8          print("3. Salir")
9          opcion = input("Selecciona una opción (1/2/3): ")
10
11         if opcion == '1':
12             imprimir_lista(comidas_favoritas)
13         elif opcion == '2':
14             modificar_lista(comidas_favoritas)
15         elif opcion == '3':
16             print("👋 ¡Hasta luego!")
17             break
18         else:
19             print(" Opción no válida. Intenta de nuevo.")
20     menu()
21
22 print ("Junior Pacajes Banegas - FIN DEL PROGRAMA")
23

```

Este programa usa funciones importadas desde un módulo externo (funciones.py) para manejar una lista de comidas favoritas. Muestra un menú interactivo para ver o modificar elementos de la lista.

Prueba:

```

--- MENÚ ---
1. Ver lista de comidas favoritas
2. Modificar una comida
3. Salir
Selecciona una opción (1/2/3): 1
Lista de comidas favoritas:
1. Pizza
2. hamburgueza
3. salteñas

--- MENÚ ---
1. Ver lista de comidas favoritas
2. Modificar una comida
3. Salir
Selecciona una opción (1/2/3): 2
Lista de comidas favoritas:
1. Pizza
2. hamburgueza
3. salteñas
¿Qué número de comida deseas modificar? (1, 2, 3...): 2
¿Cuál es la nueva comida favorita?: sushi
¡Comida actualizada con éxito!

--- MENÚ ---
1. Ver lista de comidas favoritas
2. Modificar una comida
3. Salir
Selecciona una opción (1/2/3): 1
Lista de comidas favoritas:
1. Pizza
2. sushi
3. salteñas

--- MENÚ ---
1. Ver lista de comidas favoritas
2. Modificar una comida
3. Salir
Selecciona una opción (1/2/3): 3
👋 ¡Hasta luego!
Junior Pacajes Banegas - FIN DEL PROGRAMA
~.../Práctico_I/comidas_fav$ █

```

Estos ejercicio enseña a organizar el código en pedacitos (funciones) que hacen tareas específicas, como mostrar la lista o cambiar algo en ella. Además, con el menú, el usuario puede decidir qué quiere hacer y el programa responde según su elección, lo que lo hace más fácil de usar.

5. Tablas de Multiplicación

5.1 Tabla de Multiplicar

```

1 num_tabla = int ( input("introduce un numero : "))
2 print ("tabla del ",num_tabla)
3 for i in range (1,11) :
4     resultado = num_tabla * i
5     print (num_tabla,"*",i,"=",resultado)
6
7 print ("---fin del programa ----- Jose alejandro Zabala romero")

```

Este programa solicita un número al usuario y muestra su tabla de multiplicar del 1 al 10.

Prueba

```

~/.../Practico_I/tabla_multi$ python tabla_mult.py
introduce un numero : 7
tabla del  7
7 * 1 = 7
7 * 2 = 14
7 * 3 = 21
7 * 4 = 28
7 * 5 = 35
7 * 6 = 42
7 * 7 = 49
7 * 8 = 56
7 * 9 = 63
7 * 10 = 70
---fin del programa ----- Jose alejandro Zabala romero
~/.../Practico_I/tabla_multi$ █

```

5.2 Tabla de Multiplicar con Funciones y Pruebas

```

1 def tabla_multiplicar(numero):
2     """
3         Devuelve una lista con la tabla de multiplicar del número dado del 1 al 10.
4     """
5     tabla = []
6     for i in range(1, 11):
7         resultado = numero * i
8         tabla.append(resultado)
9     return tabla
10 # ===== Pruebas unitarias =====
11 assert tabla_multiplicar(2) == [2, 4, 6, 8, 10, 12, 14, 16, 18, 20], "Error en la tabla del 2"
12 assert tabla_multiplicar(5) == [5, 10, 15, 20, 25, 30, 35, 40, 45, 50], "Error en la tabla del 5"
13 assert tabla_multiplicar(0) == [0] * 10, "Error en la tabla del 0"
14 print(" Pruebas unitarias pasadas con éxito..")
15 # ===== Ejecución interactiva =====
16 num_tabla = int(input("Introduce el número de la tabla: "))
17 print(f"--- Tabla del {num_tabla} ---")
18 tabla = tabla_multiplicar(num_tabla)
19 for i, resultado in enumerate(tabla, start=1):
20     print(f"{num_tabla} x {i} = {resultado}")
21
22 print("Fin del programa-----Junior Pacajes Banegas")

```

Este programa genera y muestra la tabla de multiplicar de un número ingresado por el usuario. Incluye una función, pruebas unitarias y una sección interactiva.

Prueba

```

~/.../Practico_I/tabla_multi$ python tabla_multi2.py
Pruebas unitarias pasadas con éxito.
Introduce el número de la tabla: 8
--- Tabla del 8 ---
8 x 1 = 8
8 x 2 = 16
8 x 3 = 24
8 x 4 = 32
8 x 5 = 40
8 x 6 = 48
8 x 7 = 56
8 x 8 = 64
8 x 9 = 72
8 x 10 = 80
Fin del programa-----Junior Pacajes Banegas
~/.../Practico_I/tabla_multi$ █

```

5.3 Tabla de Multiplicar con Límite Configurable

```

1 def tabla_multiplicar(numero, hasta=10):
2     """Devuelve una lista con la tabla de multiplicar de 'numero' hasta 'hasta'."""
3     return [numero * i for i in range(1, hasta + 1)]
4 # ===== Pruebas unitarias =====
5 assert tabla_multiplicar(2) == [2, 4, 6, 8, 10, 12, 14, 16, 18, 20], "Error en la tabla del 2"
6 assert tabla_multiplicar(5) == [5, 10, 15, 20, 25, 30, 35, 40, 45, 50], "Error en la tabla del 5"
7 assert tabla_multiplicar(0) == [0] * 10, "Error en la tabla del 0"
8 print("Pruebas unitarias pasadas con éxito.")
9 # ===== Ejecución interactiva =====
10 num_tabla = int(input("Introduce el número de la tabla: "))
11 print(f"--- Tabla del {num_tabla} ---")
12 for i, resultado in enumerate(tabla_multiplicar(num_tabla), start=1):
13     print(f"{num_tabla} x {i} = {resultado}")
14
15 print("Fin del programa-----Richard Hurtado")

```

Este programa define una función para generar la tabla de multiplicar de un número, hasta un límite configurable (por defecto 10). Incluye pruebas unitarias y una parte interactiva para que el usuario vea la tabla.

Prueba

```

Pruebas unitarias pasadas con éxito.
Introduce el número de la tabla: 13
--- Tabla del 13 ---
13 x 1 = 13
13 x 2 = 26
13 x 3 = 39
13 x 4 = 52
13 x 5 = 65
13 x 6 = 78
13 x 7 = 91
13 x 8 = 104
13 x 9 = 117
13 x 10 = 130
Fin del programa-----Richard Hurtado
~.../Practico_I/tabla_multi$ █

```

6. Juegos de Adivinanza (Número Secreto)

6.1 Número Secreto

```

1 numero_secreto = 7
2 while True:
3     var = int(input("Adivina el número secreto: "))
4     if var < numero_secreto:
5         print("demasiado frío")
6     elif var > numero_secreto:
7         print("demasiado caliente")
8     else:
9         print("¡Acertaste! el número secreto es: ", numero_secreto )
10        break
11 print("Fin del programa----Luis Antonio Moreno Miranda")
12 assert numero_secreto == 7

```

Este programa es un juego de adivinanza. Define un número secreto (7) y luego usa un bucle while True para pedir al usuario que ingrese un número. Según el valor ingresado, el programa indica si el número es “demasiado frío” (menor al secreto) o “demasiado caliente” (mayor al secreto). Cuando el usuario adivina correctamente, muestra un mensaje de felicitación y termina el bucle con break.

Prueba

```
~/workspace$ cd prog/alejandro/Practico_1/numero_secreto
~/. . . /Practico_1/numero_secreto$ python numero_secreto.py
Adivina el número secreto: 8
demasiado caliente
Adivina el número secreto: 3
demasiado frío
Adivina el número secreto: 4
demasiado frío
Adivina el número secreto: 6
demasiado frío
Adivina el número secreto: 10
demasiado caliente
Adivina el número secreto: 11
demasiado caliente
Adivina el número secreto: 6
demasiado frío
Adivina el número secreto: 0
demasiado frío
Adivina el número secreto: 7
¡Acertaste! el número secreto es: 7
Fin del programa---Luis Antonio Moreno Miranda
~/. . . /Practico_1/numero_secreto$
```

6.2 Número Oculto con getpass

```
1 import getpass
2 def verificar_adivinanza(numero_secreto, intento):
3     """Devuelve una pista según la diferencia entre el intento y el número secreto."""
4     if intento < numero_secreto:
5         return "Demasiado bajo"
6     elif intento > numero_secreto:
7         return "Demasiado alto"
8     else:
9         return "Correcto"
10 # ===== Pruebas unitarias =====
11 assert verificar_adivinanza(7, 3) == "Demasiado bajo", "Error: se esperaba 'Demasiado bajo'"
12 assert verificar_adivinanza(7, 10) == "Demasiado alto", "Error: se esperaba 'Demasiado alto'"
13 assert verificar_adivinanza(7, 7) == "Correcto", "Error: se esperaba 'Correcto'"
14 print(" Pruebas unitarias superadas.")
15 # ===== Juego interactivo con múltiples rondas =====
16 while True:
17     try:
18         # Ingreso oculto del número secreto
19         entrada = getpass.getpass(" ⚡ Ingresa el número secreto (oculto, máx 99): ")
20         numero_secreto = int(entrada)
21
22         if numero_secreto > 99 or numero_secreto < 0:
23             print("▲ El número debe estar entre 0 y 99.")
24             continue
25         except ValueError:
26             print("▲ Ingresa un número válido.")
27             continue
28         print(" ¡Adivina el número secreto entre 0 y 99!")
29
30         while True:
31             try:
32                 intento = int(input("Tu intento: "))
33                 resultado = verificar_adivinanza(numero_secreto, intento)
34
35                 if resultado == "Correcto":
36                     print(f" ¡Correcto! El número era {numero_secreto}.")
37                     break
38                 else:
39                     print(f" {resultado}. Intenta de nuevo.")
40             except ValueError:
41                 print(" Ingresa un número válido.")
42
43         continuar = input("¿Deseas jugar otra ronda? (Y/N): ").strip().lower()
44         if continuar != 'y':
45             break
46
47 print(" --- Fin del programa --- Junior Pacajes Banegas")
```

Este programa es un juego interactivo de adivinanza donde un jugador ingresa un número secreto (oculto con `getpass`) y otro intenta adivinarlo. Se dan pistas como "Demasiado bajo" o "Demasiado alto" hasta acertar. Incluye pruebas unitarias y permite jugar múltiples rondas.

Prueba

```
~/.../Practico_I/numero_secreto$ python 2num_ocult.py
Pruebas unitarias superadas.
💡 Ingresá el número secreto (oculto, máx 99):
  Adivina el número secreto entre 0 y 99!
Tu intento: 7
  Demasiado alto. Intenta de nuevo.
Tu intento: -7
  Demasiado bajo. Intenta de nuevo.
Tu intento: 0
  Demasiado bajo. Intenta de nuevo.
Tu intento: 1
  ¡Correcto! El número era 1.
¿Deseas jugar otra ronda? (Y/N): no
--- Fin del programa --- Junior Pacajes Banegas
~/.../Practico_I/numero_secreto$ █
```

6.3 Adivinanza múltiple

```
1  def verificar_adivinanza(numero_secreto, intento):
2      """Devuelve una pista según la diferencia entre el intento y el número secreto."""
3      if intento < numero_secreto:
4          return "Demasiado bajo"
5      elif intento > numero_secreto:
6          return "Demasiado alto"
7      else:
8          return "Correcto"
9  # ===== Pruebas unitarias =====
10 assert verificar_adivinanza(7, 3) == "Demasiado bajo", "Error: se esperaba 'Demasiado bajo'"
11 assert verificar_adivinanza(7, 10) == "Demasiado alto", "Error: se esperaba 'Demasiado alto'"
12 assert verificar_adivinanza(7, 7) == "Correcto", "Error: se esperaba 'Correcto'"
13 print(" Pruebas unitarias superadas.")
14 # ===== Juego interactivo con múltiples rondas =====
15 while True:
16     try:
17         try:
18             import getpass
19             entrada = getpass.getpass(" Ingresa el número secreto (oculto, máx 99): ")
20         except Exception:
21             print(" El entorno no soporta entrada oculta. Se usará entrada visible.")
22             entrada = input("Ingresa el número secreto (visible, máx 99): ")
23
24         numero_secreto = int(entrada)
25
26         if numero_secreto > 99 or numero_secreto < 0:
27             print(" El número debe estar entre 0 y 99.")
28             continue
29     except ValueError:
30         print(" Ingresa un número válido.")
31         continue
32     print(" ¡Adivina el número secreto entre 0 y 99!")
33     while True:
34         try:
35             intento = int(input("Tu intento: "))
36             resultado = verificar_adivinanza(numero_secreto, intento)
37
38             if resultado == "Correcto":
39                 print(f" ¡Correcto! El número era {numero_secreto}.")
40                 break
41             else:
42                 print(f" {resultado}. Intenta de nuevo.")
43         except ValueError:
44             print(" Ingresa un número válido.")
45     continuar = input("¿Deseas jugar otra ronda? (Y/N): ").strip().lower()
46     if continuar != 'y':
47         break
48 print("--- Fin del programa --- Jose Alejandro Zabala Romero")
```

En este programa se debe adivinar un número secreto entre 0 y 99. El número puede ingresar de forma oculta o visible, y se da una pista en cada intento. Además, incluye pruebas unitarias y permite jugar varias rondas.

Prueba

```
~/.../Practico_I/numero_secreto$ python adivinanza_mate.py
Pruebas unitarias superadas.
Ingresa el número secreto (oculto, máx 99):
¡Adivina el número secreto entre 0 y 99!
Tu intento: 4
Demasiado bajo. Intenta de nuevo.
Tu intento: 8
Demasiado alto. Intenta de nuevo.
Tu intento: 88
Demasiado alto. Intenta de nuevo.
Tu intento: 5
¡Correcto! El número era 5.
¿Deseas jugar otra ronda? (Y/N): no
--- Fin del programa --- Jose Alejandro Zabala Romero
~/.../Practico_I/numero_secreto$
```

7. Operaciones con Notas y Listas

7.1 Suma de Notas y Promedio Manual

```
1 # Crear una lista con notas numéricas
2 mis_notas = [85.5, 90, 78, 88.5, 95, 82]
3 # Inicializar variable para la suma
4 suma_total = 0
5 # Usar bucle for para calcular la suma total sin usar sum()
6 for nota in mis_notas:
7     suma_total += nota
8 # Calcular el promedio
9 promedio = suma_total / len(mis_notas)
10 # Imprimir resultados de forma clara
11 print(f"Suma total de las notas: {suma_total}")
12 print(f"Promedio de las notas: {promedio:.2f}")
13 # -----
14 # Validación con pruebas assert
15 # -----
16 # Cálculo esperado usando sum() como referencia
17 suma Esperada = sum(mis_notas)
18 promedio Esperado = suma Esperada / len(mis_notas)
19 # Pruebas
20 assert suma_total == suma Esperada, f" Error: la suma debería ser {suma Esperada}"
21 assert promedio == promedio Esperado, f" Error: el promedio debería ser {promedio Esperado:.2f}"
22 print("OK ¡Todo correcto! Las validaciones pasaron exitosamente.")
23
24 print("--- Fin del programa --- Jose Alejandro Zabala Romero")
25
```

Este programa calcula manualmente la suma total y el promedio de una lista de notas numéricas sin usar sum(), y luego verifica los resultados con pruebas assert usando sum() como referencia.

Prueba

```
~/.../Practico_I/suma_notas$ python suma_notas.py
Suma total de las notas: 519.0
Promedio de las notas: 86.50
OK ¡Todo correcto! Las validaciones pasaron exitosamente.
--- Fin del programa --- Jose Alejandro Zabala Romero
~/.../Practico_I/suma_notas$
```

7.2 Suma de Elementos

```

1 # Definición de la función para sumar elementos
2 def sumar_elementos(lista_numeros):
3     acumulador_suma = 0
4     for elemento_actual in lista_numeros:
5         acumulador_suma += elemento_actual
6     return acumulador_suma
7 # Casos de prueba con assert
8 print("Probando sumar_elementos...")
9 assert sumar_elementos([1, 2, 3, 4, 5]) == 15
10 assert sumar_elementos([10, -2, 5]) == 13
11 assert sumar_elementos([]) == 0 # ¡Importante probar con una lista vacía!
12 assert sumar_elementos([100]) == 100
13 print("OK! ¡Pruebas para sumar_elementos pasaron!")
14
15 print("--- Fin del programa --- Luis Antonio Moreno Miranda")

```

Este programa define una función para sumar todos los elementos de una lista. Incluye pruebas con assert para validar que la función funciona correctamente, incluso con listas vacías.

Prueba

```

~/.../Practico_I/suma_notas$ python suma_elemts.py
Probando sumar_elementos...
OK! ¡Pruebas para sumar_elementos pasaron!
--- Fin del programa --- Luis Antonio Moreno Miranda
~/.../Practico_I/suma_notas$ 

```

7.3 Suma Pythonica con sum()

```

1 # Definición de la función para sumar elementos (versión pythonica)
2 def sumar_elementos(lista_numeros):
3     return sum(lista_numeros)
4     # return sum([x for x in lista_numeros])
5 # Casos de prueba con assert
6 print("Probando sumar_elementos...")
7 assert sumar_elementos([1, 2, 3, 4, 5]) == 15
8 assert sumar_elementos([10, -2, 5]) == 13
9 assert sumar_elementos([]) == 0 # ¡Importante probar con una lista vacía!
10 assert sumar_elementos([100]) == 100
11 print("¡Pruebas para sumar_elementos pasaron! ")
12 print("--- Fin del programa --- Luis Antonio Moreno Miranda")

```

Esta versión de la función sumar_elementos usa una expresión pythonica con sum() y comprensión de listas para sumar los elementos de una lista. Incluye pruebas con assert para validar su correcto funcionamiento.

Prueba

```

~/.../Practico_I/suma_notas$ python suma_pythonica.py
Probando sumar_elementos...
¡Pruebas para sumar_elementos pasaron!
--- Fin del programa --- Luis Antonio Moreno Miranda
~/.../Practico_I/suma_notas$ 

```

7.4 Append vs Insert

```

1 # Creamos una lista vacía
2 numeros = []
3 # append agrega al final
4 numeros.append(10)
5 numeros.append(20)
6 print("Usando append:", numeros) # [10, 20]
7 # insert agrega en una posición específica
8 numeros.insert(1, 15) # insertar en índice 1 el valor 15
9 print("Usando insert:", numeros) # [10, 15, 20]
10 # Queremos duplicar cada número
11 numeros = [1, 2, 3, 4]
12 duplicados = list(map(lambda x: x * 5.5, numeros))
13 print("Usando map:", duplicados) # [5.5, 11.0, 16.5, 22.0]
14
15 print("--- Fin del programa --- Jose Alejandro Zabala Romero")

```

Este programa muestra cómo crear y modificar listas en Python usando métodos como `append` e `insert`. Además, usa `map` con una función `lambda` para transformar los elementos de una lista, multiplicándose por 5.5.

Prueba

```

~/.Practico_I/suma_notas$ python appendVsinsert.py
Usando append: [10, 20]
Usando insert: [10, 15, 20]
Usando map: [5.5, 11.0, 16.5, 22.0]
--- Fin del programa --- Jose Alejandro Zabala Romero
~/.Practico_I/suma_notas$ 

```

8. Contar Elementos

8.1 Contar un Elemento Específico

```

1 # Definición de la función
2 def contar_elemento(lista, elemento_buscado):
3     contador = 0
4     for elemento in lista:
5         if elemento == elemento_buscado:
6             contador += 1
7     return contador
8 # -----
9 # Casos de prueba con assert
10 # -----
11 print("\nProbando contar_elemento...")
12
13 assert contar_elemento([1, 2, 3, 2, 4, 2], 2) == 3
14 assert contar_elemento(["a", "b", "a", "c", "a"], "a") == 3
15 assert contar_elemento(["sol", "luna", "estrella"], "marte") == 0
16 assert contar_elemento([], 5) == 0
17 assert contar_elemento([True, False, True, True], True) == 3
18
19 print("Las pruebas para contar_elemento pasaron!")
20 print("--- Fin del programa --- Leonardo Montenegro ")

```

Este programa define una función que cuenta cuántas veces aparece un elemento específico en una lista. Incluye pruebas con `assert` para validar que funciona correctamente en diferentes casos, incluyendo listas vacías y tipos variados.

Prueba

```

~/.Practico_I/Contar_elemnt$ python contar_elemnt.py
Probando contar_elemento...
Las pruebas para contar_elemento pasaron!
--- Fin del programa --- Leonardo Montenegro
~/.Practico_I/Contar_elemnt$ 

```

8.2 Contar con count()

```

1 def contar_elemento(lista, elemento_buscado):
2     return lista.count(elemento_buscado)
3 print("\nProbando contar_elemento con .count()...")
4
5 assert contar_elemento([1, 2, 3, 2, 4, 2], 2) == 3
6 assert contar_elemento([1, 2, 3, 2, 4, 2], 1) == 1
7 assert contar_elemento(["a", "b", "a", "c", "a"], "a") == 3
8 assert contar_elemento(["sol", "luna", "estrella"], "marte") == 0
9 assert contar_elemento([], 5) == 0
10 assert contar_elemento([True, False, True, True], True) == 3
11
12 print("¡Pruebas con .count() pasaron!")
13 print("--- Fin del programa --- Richard Hurtado")

```

Este código define una función llamada `contar_elemento` que utiliza el método `.count()`, para contar cuántas veces aparece un elemento en una lista. Está estructurado con una función, seguida de pruebas usando `assert` para verificar que funciona correctamente. La parte compleja de entender fue el uso de `assert` para hacer pruebas automáticas.

Prueba

```

~/.../Practico_I/Contar_elemnt$ python contar_elemnt_count.py
Probando contar_elemento con .count()...
¡Pruebas con .count() pasaron!
--- Fin del programa --- Richard Hurtado
~/.../Practico_I/Contar_elemnt$ █

```

9. Factorial (Varias Implementaciones)

9.1 Factorial Recursivo

```

1 # Definición de la función recursiva
2 def factorial(n):
3     if n < 0:
4         raise ValueError("El factorial no está definido para números negativos.")
5     elif n == 0 or n == 1:
6         return 1
7     else:
8         return n * factorial(n - 1)
9 # Pruebas
10 assert factorial(0) == 1
11 assert factorial(1) == 1
12 assert factorial(5) == 120
13 assert factorial(6) == 720
14 print("¡Pruebas de factoriales pasaron!")
15 # Solicita un número al usuario
16 try:
17     numero = int(input("Ingrese un número entero para calcular su factorial: "))
18     resultado = factorial(numero)
19     print(f"El factorial de {numero} es: {resultado}")
20 except ValueError as e:
21     print("Error:", e)
22
23 print("Fin del programa----Jose Alejandro Zabala Romero")
24

```

Este programa define una función recursiva para calcular el factorial de un número entero no negativo. Incluye pruebas con `assert` para validar la función y manejo de errores para números negativos o entradas inválidas.

Prueba

```
~/.Practico_I/factorial_$ python factorial.py
¡Pruebas de factoriales pasaron!
Ingrese un número entero para calcular su factorial: 3
El factorial de 3 es: 6
Fin del programa----Jose Alejandro Zabala Romero
~/.Practico_I/factorial_$
```

9.2 Factorial con for

```
1 # Definición de la función usando un for
2 def factorial(n):
3     if n < 0:
4         raise ValueError("El factorial no está definido para números negativos.")
5     resultado = 1
6     for i in range(2, n + 1):
7         resultado *= i
8     return resultado
9
10 # Solicitud un número al usuario
11 try:
12     numero = int(input("Ingrese un número entero para calcular su factorial: "))
13     resultado = factorial(numero)
14     print(f"El factorial de {numero} es: {resultado}")
15 except ValueError as e:
16     print("Error:", e)
17
18 print("Fin del programa----Nicodemus Berny Quiroga González")
19
```

Esta versión del cálculo del factorial utiliza un bucle for para multiplicar secuencialmente desde 2 hasta n. Incluye manejo de errores para entradas negativas y solicita el número al usuario.

Prueba

```
~/.Practico_I/factorial_$ python factorial_for.py
Ingrese un número entero para calcular su factorial: 6
El factorial de 6 es: 720
Fin del programa----Nicodemus Berny Quiroga González
~/.Practico_I/factorial_$
```

9.3 Factorial con math.prod

```
1 import math
2 def factorial(n):
3     if n < 0:
4         raise ValueError(" El factorial no está definido para números negativos.")
5     return 1 if n == 0 else math.prod(range(1, n + 1))
6 def mostrar_factorial(n):
7     if n < 0:
8         print(" El factorial no está definido para números negativos.")
9     return
10    pasos = ' x '.join(str(i) for i in range(1, n + 1)) if n > 0 else "1"
11    resultado = factorial(n)
12    print(f"{pasos} = {resultado}")
13 #
14 # Ejecución interactiva
15 #
16 try:
17     numero = int(input("Ingrese un número entero para calcular su factorial: "))
18     mostrar_factorial(numero)
19 except ValueError:
20     print(" Entrada inválida. Por favor ingresa un número entero.")
21
22
23 print("Fin del programa----Junior Pacajes Banegas")
```

Este programa calcula el factorial usando la función math.prod para multiplicar rangos de números. Incluye validación de entrada y manejo de errores.

Prueba

```
~/.../Practico_I/factorial_$ python factorial_math.py
Ingrese un número entero para calcular su factorial: 9
1 x 2 x 3 x 4 x 5 x 6 x 7 x 8 x 9 = 362880
Fin del programa-----Junior Pacajes Banegas
~/.../Practico_I/factorial_$
```

9.4 Factorial con reduce

```
1  from functools import reduce
2  def factorial(n):
3      if n < 0:
4          raise ValueError("El factorial no está definido para números negativos.")
5      return 1 if n == 0 else reduce(lambda x, y: x * y, range(1, n + 1))
6  # Pruebas
7  assert factorial(0) == 1
8  assert factorial(1) == 1
9  assert factorial(5) == 120
10 assert factorial(6) == 720
11 print("¡Pruebas de factoriales pasaron!")
12 # Solicita un número al usuario
13 try:
14     numero = int(input("Ingrese un número entero para calcular su factorial: "))
15     resultado = factorial(numero)
16     print(f"El factorial de {numero} es: {resultado}")
17     print("Fin del programa-----Nicodemus Berny Quiroga González")
18 except ValueError as e:
19     print("Error:", e)
20 except Exception:
21     print("Por favor, ingrese un número entero válido.")
```

Este programa calcula el factorial usando `functools.reduce` con una función lambda para multiplicar los números en el rango. Incluye validaciones con `assert` y manejo de errores para números negativos.

Prueba

```
~/.../Practico_I/factorial_$ python factor_reduce.py
¡Pruebas de factoriales pasaron!
Ingrese un número entero para calcular su factorial: 11
El factorial de 11 es: 39916800
Fin del programa-----Nicodemus Berny Quiroga González
```

10. For Tradicional vs Lista por Comprensión

10.1 Comparación entre for clásico y forma "pytónica"

```
1  #CALCULO DEL CUADRADO DE LOS ELEMENTOS DE UNA LISTA DEL 0 AL 4
2  cuadrados = []
3  # TRADICIONAL
4  for x in range(5):
5      cuadrados.append(x * x)
6  print(cuadrados)  # [0, 1, 4, 9, 16]
7  #PYTHONICO
8  cuadrados = [x * x for x in range(5)]
9  print(cuadrados)  # [0, 1, 4, 9, 16]
10
11 print("Fin del programa-----Leonardo Montenegro ")
```

Este código calcula el cuadrado de los números del 0 al 4 usando dos métodos: un bucle tradicional con `.append()` y una lista por comprensión (forma "pytónica").

Prueba

```
~/.../Practico_I/factorial_$ python fortradicionalVS.py
[0, 1, 4, 9, 16]
[0, 1, 4, 9, 16]
Fin del programa ----- Leonardo Montenegro
~/.../Practico_I/factorial_$
```

11. Invertir Cadenas y Listas

11.1 Invertir Texto con for y range

```
prog2 > alejandro > invertir_ > invertir_texto.py > texto

1 texto = "hola"
2 invertido = ''.join(texto[i] for i in range(len(texto) - 1, -1, -1))
3 print(invertido) # Resultado: "aloh"
4
5 print ("fin del programa ---- Jose Alejandro Zabala Romero")
```

Este código invierte una cadena ("hola" → "aloh") utilizando un recorrido inverso con índices en un bucle for. Usa join para construir la nueva cadena .La parte que resultó complejo de entender fue cómo funciona el rango inverso en range(len(texto) - 1, -1, -1).

Prueba

```
~/.../Practico_I/invertir_$ python invertir_texto.py
aloh
fin del programa ---- Jose Alejandro Zabala Romero
~/.../Practico_I/invertir_$
```

11.2 Invertir Lista con bucle

```
1 # Definición de la función
2 def invertir_lista(lista_original):
3     lista_invertida = []
4     # Recorremos la lista desde el final hasta el inicio
5     for i in range(len(lista_original) - 1, -1, -1):
6         lista_invertida.append(lista_original[i])
7
8     return lista_invertida
9
10 # Casos de prueba con assert
11 #
12 print("\nProbando invertir_lista...")
13 lista_prueba = [1, 2, 3, 4, 5]
14 lista_resultante = invertir_lista(lista_prueba)
15 assert lista_resultante == [5, 4, 3, 2, 1], " Error en inversión"
16 assert lista_prueba == [1, 2, 3, 4, 5], " La lista original fue modificada!"
17 assert invertir_lista(["a", "b", "c"]) == ["c", "b", "a"]
18 assert invertir_lista([]) == []
19 print("Pruebas para invertir_lista pasaron! ")
20
21 print ("fin del programa ---- Nicodemus Berny Quiroga González")
22
```

Este código define una función para invertir una lista sin modificar la original, utilizando un bucle for con recorrido inverso y el método append.Las pruebas con assert validan el funcionamiento correctamente en diferentes casos, incluyendo listas vacías y de caracteres.La parte compleja fue entender el control del índice en el range invertido, especialmente al usar -1 como límite final.

Prueba

```
~/.../Practico_I/invertir_$ python invertir_lista.py
Probando invertir_lista...
¡Pruebas para invertir_lista pasaron!
fin del programa ---- Nicodemus Berny Quiroga González
~/.../Practico_I/invertir_$ █
```

11.3 Invertir Lista con Slicing

```
1  def invertir_lista(lista_original):
2      # lista[inicio:fin:paso]
3      return lista_original[::-1]
4  # Esto usa slicing con paso negativo
5  # inicio: índice desde donde empezar (incluye este elemento).
6  # fin: índice hasta donde cortar (no incluye este elemento).
7  # paso: cuántos pasos avanzar (puede ser negativo).
8  # Pruebas
9  print("\nProbando invertir_lista con slicing...")
10 lista_prueba = [1, 2, 3, 4, 5]
11 lista_resultante = invertir_lista(lista_prueba)
12 assert lista_resultante == [5, 4, 3, 2, 1], "Error en inversión con slicing"
13 assert lista_prueba == [1, 2, 3, 4, 5], "La lista original fue modificada!"
14 assert invertir_lista(["a", "b", "c"]) == ["c", "b", "a"]
15 assert invertir_lista([]) == []
16 print("¡Pruebas con slicing pasaron! ")
17
18 print("fin del programa ---- Lucas Mateo Diaz Badani")
19
```

Este código invierte una lista usando slicing (`[::-1]`). Utiliza estructuras de listas y el concepto de rebanado con paso negativo, sin necesidad de bucles ni métodos auxiliares. La parte que fue compleja de comprender fue cómo funciona el slicing con tres valores (`inicio:fin:paso`), especialmente cuando el paso es negativo.

Prueba

```
~/.../Practico_I/invertir_$ python invertir_list_slicing.py
Probando invertir_lista con slicing...
¡Pruebas con slicing pasaron!
fin del programa ---- Lucas Mateo Diaz Badani
~/.../Practico_I/invertir_$ █
```

La prueba(o ejecucion) del programa verifica que la función `invertir_lista` realiza correctamente la inversión de una lista sin modificar la lista original. La función utiliza slicing con paso negativo (`[::-1]`) para crear una copia invertida de la lista recibida.

Primero, se define una lista de prueba `lista_prueba` con los valores `[1, 2, 3, 4, 5]`. Se llama a la función para invertirla y el resultado se guarda en `lista_resultante`. Luego, se usa `assert` para confirmar que la lista resultante es exactamente la inversa de la original: `[5, 4, 3, 2, 1]`. Esto asegura que la inversión es correcta.

Seguidamente, se verifica que la lista original `lista_prueba` no haya sido alterada, garantizando que la función no produce efectos secundarios sobre la entrada, un aspecto clave en la programación funcional y para evitar bugs.

Después, la función se prueba con otros casos: una lista de caracteres `["a", "b", "c"]` y una lista vacía `[]`. Estas pruebas aseguran que la función funciona bien con diferentes tipos de datos y con entradas sin elementos, cubriendo casos comunes y extremos.

12. Búsqueda y Ordenamiento

12.1 Búsqueda Lineal y Binaria

```

1 # 1. Definir la función busqueda_lineal
2 def busqueda_lineal(lista, clave):
3     for i in range(len(lista)):
4         if lista[i] == clave:
5             return i
6     return -1
7 # 2. Definir la función busqueda_binaria
8 def busqueda_binaria(lista_ordenada, clave):
9     izquierda = 0
10    derecha = len(lista_ordenada) - 1
11    while izquierda <= derecha:
12        medio = (izquierda + derecha) // 2
13        if lista_ordenada[medio] == clave:
14            return medio
15        elif clave > lista_ordenada[medio]:
16            izquierda = medio + 1
17        else:
18            derecha = medio - 1
19    return -1
20 # 3. Prueba de la función busqueda_lineal
21 mi_lista_desordenada = [10, 5, 42, 8, 17, 30, 25]
22 print("Probando busqueda_lineal...")
23 assert busqueda_lineal(mi_lista_desordenada, 42) == 2
24 assert busqueda_lineal(mi_lista_desordenada, 10) == 0 # Al inicio
25 assert busqueda_lineal(mi_lista_desordenada, 25) == 6 # Al final
26 assert busqueda_lineal(mi_lista_desordenada, 99) == -1 # No existe
27 assert busqueda_lineal([], 5) == -1 # Lista vacía
28 print("¡Pruebas para busqueda_lineal pasaron! ✅")
29 print("Lucas Mateo Diaz Badani- FIN DEL PROGRAMA")
30 # 4. Prueba de la función busqueda_binaria
31 lista_ordenada = [2, 5, 8, 12, 16, 23, 38, 56, 72, 91]
32 print("\nProbando busqueda_binaria...")
33 assert busqueda_binaria(lista_ordenada, 23) == 5
34 assert busqueda_binaria(lista_ordenada, 91) == 9 # Último
35 assert busqueda_binaria(lista_ordenada, 2) == 0 # Primero
36 assert busqueda_binaria(lista_ordenada, 3) == -1 # No existe
37 assert busqueda_binaria(lista_ordenada, 100) == -1 # Fuera de rango (mayor)
38 print("¡Pruebas para busqueda_binaria pasaron! ✅")
39 print("Lucas Mateo Diaz Badani - FIN DEL PROGRAMA")
40 # 5. Experimento corregido: búsqueda binaria con lista previamente ordenada
41 print("\nRealizando el experimento correctamente (ordenando la lista)...")
42 mi_lista_desordenada = [10, 8, 42, 5, 17, 30, 25]
43 mi_lista_ordenada = sorted(mi_lista_desordenada)
44 print(f"Lista original: {mi_lista_desordenada}")
45 print(f"Lista ordenada: {mi_lista_ordenada}")
46 resultado_ordenado = busqueda_binaria(mi_lista_ordenada, 30)
47 print(f"Búsqueda binaria de '30' en lista ordenada devolvió: {resultado_ordenado}")
48 print("Lucas Mateo Diaz Badani - FIN DEL PROGRAMA")
49

```

Este código implementa dos algoritmos de búsqueda: búsqueda lineal y búsqueda binaria, utilizando estructuras como listas, bucles (for, while), y condicionales (if, else). Las pruebas con assert ayudan a verificar que ambas funciones funcionen correctamente en diferentes casos. La parte que puede resultar más difícil es entender y aplicar bien los límites (izquierda, derecha) y el cálculo del punto medio en la búsqueda binaria.

Prueba

```

~/.Practico_I/busquedas$ python Busqueda.py
Probando busqueda_lineal...
¡Pruebas para busqueda_lineal pasaron! ✅
Lucas Mateo Diaz Badani- FIN DEL PROGRAMA

Probando busqueda_binaria...
¡Pruebas para busqueda_binaria pasaron! ✅
Lucas Mateo Diaz Badani - FIN DEL PROGRAMA

Realizando el experimento correctamente (ordenando la lista)...
Lista original: [10, 8, 42, 5, 17, 30, 25]
Lista ordenada: [5, 8, 10, 17, 25, 30, 42]
Búsqueda binaria de '30' en lista ordenada devolvió: 5
Lucas Mateo Diaz Badani - FIN DEL PROGRAMA
~/.Practico_I/busquedas$ █

```

12.2 Ordenamiento Mergesort

```

1  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: VENCER (Recursión)
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 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     # Agregar cualquier elemento restante
27     resultado.extend(izquierda[i:])
28     resultado.extend(derecha[j:])
29     return resultado
30 # --- Prueba ---
31 lista_prueba = [8, 3, 5, 1]
32 print("Lista original:", lista_prueba)
33 resultado = merge_sort(lista_prueba)
34 print("Lista ordenada:", resultado)
35 # --- Pruebas automatizadas ---
36 assert merge_sort([]) == []                      # Lista vacía
37 assert merge_sort([1]) == [1]                     # Lista con un solo elemento
38 assert merge_sort([5, 2]) == [2, 5]                # Lista con dos elementos
39 assert merge_sort([3, 1, 2]) == [1, 2, 3]          # Lista con tres elementos desordenados
40 assert merge_sort([10, 5, 3, 8, 6, 2]) == [2, 3, 5, 6, 8, 10] # Lista par
41 assert merge_sort([9, 7, 5, 3, 1]) == [1, 3, 5, 7, 9] # Lista en orden descendente
42 assert merge_sort([1, 2, 3, 4, 5]) == [1, 2, 3, 4, 5] # Lista ya ordenada
43 assert merge_sort([4, 2, 2, 4, 1]) == [1, 2, 2, 4, 4] # Lista con elementos repetidos
44 assert merge_sort([-100, -50, 0, 50, -100]) == [-100, -50, 0, 50, 100] # Lista con enteros negativos y positivos
45 assert merge_sort([2.5, 1.2, 3.8]) == [1.2, 2.5, 3.8] # Lista con flotantes
46 print("¡Todas las pruebas con assert pasaron correctamente!")
47 print("Nicodemus Berny Quiroga González - FIN DEL PROGRAMA")

```

El código implementa el algoritmo de ordenamiento **Merge Sort**, que utiliza la estrategia divide y vencerás para ordenar listas. La función principal, `merge_sort`, primero verifica si la lista tiene uno o cero elementos; en ese caso, simplemente la retorna porque ya está ordenada (condición base de la recursión).

Si la lista tiene más de un elemento, se divide en dos mitades. Luego, la función se llama recursivamente sobre cada mitad para ordenarlas por separado. Una vez ordenadas las dos mitades, se combinan en una sola lista ordenada usando la función `merge`.

La función `merge` toma dos listas ordenadas (izquierda y derecha) y las fusiona en una sola lista ordenada. Lo hace comparando elemento a elemento y agregando el menor de cada par a la lista resultado, hasta que se terminan los elementos de alguna de las dos listas. Luego añade los elementos restantes de la lista que aún tenga datos.

El código incluye varias pruebas automatizadas con `assert` que cubren distintos casos: listas vacías, con un solo elemento, con números desordenados, con valores repetidos, negativos y números flotantes.

Prueba

```
~/. . /Practico_I/busquedas$ python advance_orden.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 [1] y [2]
Mezclaría [3] y [1, 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 [9] y [7]
Mezclaría [3] y [1]
Mezclaría [5] y [1, 3]
Mezclaría [7, 9] y [1, 3, 5]
Mezclaría [1] y [2]
Mezclaría [4] y [5]
Mezclaría [3] y [4, 5]
Mezclaría [1, 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 [100] y [-50]
Mezclaría [50] 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]
¡Todas las pruebas con assert pasaron correctamente!
Nicodemus Berny Quiroga González - FIN DEL PROGRAMA
```

Durante la ejecución del programa ,se pone en funcionamiento el algoritmo **Merge Sort** aplicado a una lista de prueba inicial. El programa inicia mostrando la lista original [8, 3, 5, 1], lo que permite identificar claramente cuál será la entrada a ordenar. Inmediatamente después, se visualiza cómo el algoritmo divide esta lista en sublistas más pequeñas, hasta llegar a elementos individuales, lo que representa la fase de **división recursiva** característica del método "divide y vencerás".

A medida que el algoritmo va combinando las sublistas, se muestran en pantalla mensajes como Mezclaría [8] y [3], Mezclaría [5] y [1], y posteriormente Mezclaría [3, 8] y [1, 5]. Estas impresiones reflejan de forma clara la **fase de combinación**, donde se unen sublistas previamente ordenadas para formar nuevas listas también ordenadas. Este proceso es sumamente útil para entender cómo se construye la lista final de manera progresiva y ordenada.

Una vez finalizada la ordenación de la lista inicial, se imprime el resultado: Lista ordenada: [1, 3, 5, 8], lo cual confirma que el algoritmo ha cumplido su objetivo. A partir de allí, el programa realiza una serie de **pruebas automatizadas** utilizando sentencias assert, que validan el correcto funcionamiento del algoritmo en distintos escenarios. Estas pruebas incluyen listas vacías, listas con un solo elemento, listas con números negativos, elementos flotantes, valores repetidos, y listas ya ordenadas o en orden inverso.

Durante cada prueba, también se imprime el proceso de mezcla correspondiente, lo cual brinda una trazabilidad clara de cómo el algoritmo trata cada caso. Finalmente, al superar todas las pruebas , se muestra el mensaje ¡Todas las pruebas con assert pasaron correctamente!

12.3 Ordenamiento Burbuja e Inserción

```

1 # ALGORITMO DE BURBUJA
2 def ordenamiento_burbuja(lista):
3     """
4     Ordena una lista en orden ascendente utilizando el algoritmo de burbuja.
5     Modifica la lista original (in-place) y también la retorna por conveniencia.
6     """
7     n = len(lista)
8     for i in range(n - 1):
9         hubo_intercambio = False
10        for j in range(n - 1 - i):
11            if lista[j] > lista[j + 1]:
12                lista[j], lista[j + 1] = lista[j + 1], lista[j]
13                hubo_intercambio = True
14            if not hubo_intercambio:
15                break
16        return lista
17 if __name__ == "__main__":
18     numeros = [6, 3, 8, 2, 5]
19     print("Antes:", numeros)
20     ordenamiento_burbuja(numeros)
21     print("Después Ordenamiento Burbuja:", numeros)
22 # --- Pruebas burbuja ---
23 lista1 = [6, 3, 8, 2, 5]
24 ordenamiento_burbuja(lista1)
25 assert lista1 == [2, 3, 5, 6, 8]
26 lista2 = [1, 2, 3, 4, 5]
27 ordenamiento_burbuja(lista2)
28 assert lista2 == [1, 2, 3, 4, 5]
29 lista3 = [5, 4, 3, 2, 1]
30 ordenamiento_burbuja(lista3)
31 assert lista3 == [1, 2, 3, 4, 5]
32 lista4 = [5, 1, 4, 2, 8, 5, 2]
33 ordenamiento_burbuja(lista4)
34 assert lista4 == [1, 2, 2, 4, 5, 5, 8]
35 assert ordenamiento_burbuja([]) == []
36 assert ordenamiento_burbuja([42]) == [42]
37 print("¡Todas las pruebas ordenamiento burbuja pasaron! ✅")
38 print("JOSE ALEJANDRO ZABALA ROMERO - FIN PROGRAMA ORDENAMIENTO BURBUJA")
39 # ALGORITMO DE INSERCIÓN
40 def ordenamiento_insercion(lista):
41     """
42     Ordena la lista in-place usando el algoritmo de inserción.
43     Retorna la misma lista por conveniencia.
44     """
45     for i in range(1, len(lista)):
46         valor_actual = lista[i]
47         posicion_actual = i
48         while posicion_actual > 0 and lista[posicion_actual - 1] > valor_actual:
49             lista[posicion_actual] = lista[posicion_actual - 1]
50             posicion_actual -= 1
51         lista[posicion_actual] = valor_actual
52     return lista
53 # --- Pruebas inserción ---
54 lista1 = [6, 3, 8, 2, 5]
55 print("Antes:", lista1)
56 ordenamiento_insercion(lista1)
57 print("Después Ordenamiento Inserción:", lista1)
58 assert lista1 == [2, 3, 5, 6, 8]
59 lista2 = [1, 2, 3, 4, 5]
60 print("Antes:", lista2)
61 ordenamiento_insercion(lista2)
62 print("Después Ordenamiento Inserción:", lista2)
63 assert lista2 == [1, 2, 3, 4, 5]
64 lista3 = [5, 4, 3, 2, 1]
65 ordenamiento_insercion(lista3)
66 assert lista3 == [1, 2, 3, 4, 5]
67 lista4 = [5, 1, 4, 2, 8, 5, 2]
68 ordenamiento_insercion(lista4)
69 assert lista4 == [1, 2, 2, 4, 5, 5, 8]
70 assert ordenamiento_insercion([]) == []
71 assert ordenamiento_insercion([42]) == [42]
72 print("¡Todas las pruebas del ordenamiento por inserción pasaron! ✅")
73 print("JOSE ALEJANDRO ZABALA ROMERO - FIN PROGRAMA ORDENAMIENTO INSERCIÓN")

```

Prueba

```
~/.../Practico_I/busquedas$ python order_busbj.py
Antes: [6, 3, 8, 2, 5]
Después Ordenamiento Burbuja: [2, 3, 5, 6, 8]
¡Todas las pruebas ordenamiento burbuja pasaron! ✓
JOSE ALEJANDRO ZABALA ROMERO - FIN PROGRAMA ORDENAMIENTO BURBUJA
Antes: [6, 3, 8, 2, 5]
Después Ordenamiento Inserción: [2, 3, 5, 6, 8]
Antes: [1, 2, 3, 4, 5]
Después Ordenamiento Inserción: [1, 2, 3, 4, 5]
¡Todas las pruebas del ordenamiento por inserción pasaron! ✓
JOSE ALEJANDRO ZABALA ROMERO - FIN PROGRAMA ORDENAMIENTO INSERCIÓN
```

Este código implementa dos algoritmos de ordenamiento: burbuja e inserción, utilizando listas, bucles for, condicionales y control de índices. Ambos métodos ordenan los elementos in-place (sin crear nuevas listas) y están bien estructurados. La parte compleja fue entender cómo los elementos se intercambian o desplazan dentro del ciclo interno, especialmente al manejar condiciones como si ya está ordenado o hay elementos repetidos.

12.4 Ordenamiento con Heap Sort

```
1  def heapificar(lista, tamano_heap, indice_raiz):
2      """
3          Asegura que el subárbol con raíz en el índice dado cumpla la propiedad de Max Heap.
4      """
5      mayor = indice_raiz
6      hijo_izquierdo = 2 * indice_raiz + 1
7      hijo_derecho = 2 * indice_raiz + 2
8      if hijo_izquierdo < tamano_heap and lista[hijo_izquierdo] > lista[mayor]:
9          mayor = hijo_izquierdo
10     if hijo_derecho < tamano_heap and lista[hijo_derecho] > lista[mayor]:
11         mayor = hijo_derecho
12     if mayor != indice_raiz:
13         lista[indice_raiz], lista[mayor] = lista[mayor], lista[indice_raiz]
14         heapificar(lista, tamano_heap, mayor)
15 def ordenar_por_heap(lista):
16     """
17     Ordena la lista utilizando el algoritmo Heap Sort.
18     """
19     n = len(lista)
20     # Construir el Max Heap
21     for i in range(n // 2 - 1, -1, -1):
22         heapificar(lista, n, i)
23     # Extraer elementos del heap uno por uno
24     for i in range(n - 1, 0, -1):
25         lista[0], lista[i] = lista[i], lista[0]
26         heapificar(lista, i, 0)
27     # ✅ PRUEBAS CON ASSERTS (TESTING)
28     # Prueba 1: Lista desordenada
29     numeros1 = [12, 11, 13, 5, 6, 7]
30     ordenar_por_heap(numeros1)
31     assert numeros1 == sorted(numeros1), f"✗ Error en números1: {numeros1}"
32     # Prueba 2: Lista ya ordenada
33     numeros2 = [1, 2, 3, 4, 5]
34     ordenar_por_heap(numeros2)
35     assert numeros2 == [1, 2, 3, 4, 5], f"✗ Error en números2: {numeros2}"
36     # Prueba 3: Lista en orden inverso
37     numeros3 = [9, 8, 7, 6, 5]
38     ordenar_por_heap(numeros3)
39     assert numeros3 == [5, 6, 7, 8, 9], f"✗ Error en números3: {numeros3}"
40     # Prueba 4: Lista con elementos repetidos
41     numeros4 = [3, 1, 2, 3, 1]
42     ordenar_por_heap(numeros4)
43     assert numeros4 == sorted(numeros4), f"✗ Error en números4: {numeros4}"
44     # Prueba 5: Lista con un solo elemento
45     numeros5 = [42]
46     ordenar_por_heap(numeros5)
47     assert numeros5 == [42], f"✗ Error en números5: {numeros5}"
48     # Prueba 6: Lista vacía
49     numeros6 = []
50     ordenar_por_heap(numeros6)
51     assert numeros6 == [], f"✗ Error en números6: {numeros6}"
52     print("✅ Todas las pruebas pasaron correctamente.")
53     print("Fin del programa----Leonardo Montenegro")
```

Prueba

```
~/. . . /Practico_I/busquedas$ python heapsort.py
✓ Todas las pruebas pasaron correctamente.
Fin del programa----Leonardo Montenegro
~/. . . /Practico_I/busquedas$ █
```

Este programa implementa el algoritmo de ordenamiento Heap Sort usando un enfoque basado en montículos (Max Heap). Incluye pruebas automáticas con listas variadas. La parte compleja fue entender la función 'heapificar', que ajusta el subárbol para mantener la propiedad del heap.

13. Vector Misterioso (Lista Secreta Interactiva)

13.1 Lista Secreta

```
1 # Lista secreta predefinida por el profesor
2 lista_secreta = [7, 14, 3, 21, 9]
3 # Instrucciones para los estudiantes
4 print(" ¡Bienvenidos al juego del Ahorcado Lógico con Listas!")
5 print("Debes descubrir qué números hay en la lista secreta de 5 elementos.")
6 print("Pero no puedes verla directamente. En cada turno puedes hacer una 'pregunta' en forma de código Python.")
7 print("Ejemplos: len(lista_secreta), lista_secreta[2], lista_secreta[0] > 10")
8 print("Cuando creas tener la lista completa, escribe: adivinar")
9 print("-----")
10 # Bucle de interacción
11 while True:
12     instrucion = input("">>>> Escribe tu instrucción (o 'adivinar'): ")
13     if instrucion.strip().lower() == "adivinar":
14         intento = input(" Escribe tu intento de lista (separa por comas): ")
15         try:
16             intento_lista = [int(x.strip()) for x in intento.split(",")]
17             if intento_lista == lista_secreta:
18                 print(" ¡Correcto! Has descubierto la lista secreta.")
19                 break
20             else:
21                 print(" Esa no es la lista correcta. Sigue preguntando.")
22         except ValueError:
23             print(" Error: asegúrate de escribir solo números separados por comas.")
24     else:
25         try:
26             # Evaluar la instrucción del alumno en un entorno controlado
27             resultado = eval(instrucion, {"lista_secreta": lista_secreta})
28             print("Resultado:", resultado)
29         except Exception as e:
30             print(" Error en tu instrucción:", e)
31
32
33 print("fin del programa ---- Junior Pacajes Banegas")
```

Prueba

```
~/. . . /Practico_I/vector_misterioso$ python lista_secret.py
¡Bienvenidos al juego del Ahorcado Lógico con Listas!
Debes descubrir qué números hay en la lista secreta de 5 elementos.
Pero no puedes verla directamente. En cada turno puedes hacer una 'pregunta' en forma de código Python.
Ejemplos: len(lista_secreta), lista_secreta[2], lista_secreta[0] > 10
Cuando creas tener la lista completa, escribe: adivinar
-----
>>> Escribe tu instrucción (o 'adivinar'): 5
Resultado: 5
>>> Escribe tu instrucción (o 'adivinar'): 7
Resultado: 7
>>> Escribe tu instrucción (o 'adivinar'): 21
Resultado: 21
>>> Escribe tu instrucción (o 'adivinar'): adivinar
Escribe tu intento de lista (separa por comas): 7,14,3,21,9
¡Correcto! Has descubierto la lista secreta.
Fin del programa ---- Junior Pacajes Banegas
~/. . . /Practico_I/vector_misterioso$ █
```

Este programa implementa un juego interactivo sobre el uso de listas y el método eval() para evaluar instrucciones de los usuarios. Utiliza estructuras como bucles while, condicionales if, manejo de excepciones (try-except), y entrada por teclado con input(). Lo complejo de entender fue controlar el uso de eval, ya que evalúa código dinámico y requiere cuidado. También, interpretar las instrucciones tipo lista_secreta[2] o lista_secreta[0] > 10.

13.2 Lista Secreta con Validación de Longitud

```

1  # ====== ETAPA 1: INGRESO DE LISTA SECRETA ======
2  print("💡 Ingreso de la lista secreta (NO visible para los estudiantes)")
3  while True:
4      try:
5          cantidad = int(input("¿Cuántos elementos tendrá la lista secreta?: "))
6          if cantidad >= 1:
7              break
8          print("⚠ Debes ingresar al menos un elemento.")
9      except ValueError:
10         print("⚠ Ingresa un número válido.")
11 while True:
12     entrada = input(f"👉 Ingresa {cantidad} números separados por coma: ")
13     try:
14         lista_secreta = [int(x.strip()) for x in entrada.split(",")]
15         if len(lista_secreta) == cantidad:
16             break
17         print(f"⚠ Debes ingresar {cantidad} elementos. Ingresaste {len(lista_secreta)}." )
18     except ValueError:
19         print("⚠ Ingresa solo números válidos separados por comas.")
20 print(f"\n✅ Lista secreta de {cantidad} elementos cargada correctamente.\n")
21 # ====== ETAPA 2: INSTRUCCIONES ======
22 print("🌟 ¡Bienvenidos al juego del Ahorcado Lógico con Listas!")
23 print(f"La lista secreta tiene {cantidad} elementos.")
24 print("Puedes hacer preguntas como: len(lista_secreta), lista_secreta[2], lista_secreta[0] > 10")
25 print("Cuando creas saber la lista completa, escribe: adivinar")
26 print("-")
27 # ====== ETAPA 3: BUCLE DE JUEGO ======
28 while True:
29     instrucion = input("">>>> Instrucción (o 'adivinar'): ").strip().lower()
30
31     if instrucion == "adivinar":
32         intento = input("👉 Tu intento de lista (separa por comas): ")
33         try:
34             intento_lista = [int(x.strip()) for x in intento.split(",")]
35             if intento_lista == lista_secreta:
36                 print("🎉 ¡Correcto! Has descubierto la lista secreta.")
37                 break
38             print("✖ Esa no es la lista correcta.")
39             aciertos = sum(
40                 1 for i, val in enumerate(intento_lista)
41                 if i < len(lista_secreta) and val == lista_secreta[i]
42             )
43             for i in range(len(lista_secreta)):
44                 if i >= len(intento_lista):
45                     print(f"⚠ Faltó ingresar un número en la posición {i}")
46                 elif intento_lista[i] == lista_secreta[i]:
47                     print(f"✅ Posición {i}: correcto ({intento_lista[i]})")
48                 else:
49                     print(f"✖ Posición {i}: incorrecto (tuviste {intento_lista[i]})")
50             print(f"👉 Aciertos totales: {aciertos} de {len(lista_secreta)}")
51         except ValueError:
52             print("⚠ Ingresa solo números separados por comas.")
53     else:
54         try:
55             resultado = eval(instrucion, {"lista_secreta": lista_secreta})
56             print("Resultado:", resultado)
57         except Exception as e:
58             print("⚠ Error en tu instrucción:", e)
59 print("fin del programa ---- Richard Hurtado")
60

```

Prueba

```
~/.../Practico_I/vector_misterioso$ python 2list_secret.py
💡 Ingreso de la lista secreta (NO visible para los estudiantes)
¿Cuántos elementos tendrá la lista secreta?: 5
👉 Ingresa 5 números separados por coma: 8,3,10,7,5

✅ Lista secreta de 5 elementos cargada correctamente.

💡 ¡Bienvenidos al juego del Ahorcado Lógico con Listas!
La lista secreta tiene 5 elementos.
Puedes hacer preguntas como: len(lista_secreta), lista_secreta[2], lista_secreta[0] > 10
Cuando creas saber la lista completa, escribe: adivinar
-----
>>> Instrucción (o 'adivinar'): 5
Resultado: 5
>>> Instrucción (o 'adivinar'): 8
Resultado: 8
>>> Instrucción (o 'adivinar'): adivinar
💡 Tu intento de lista (separa por comas): 8,3,10,7,5
👉 ¡Correcto! Has descubierto la lista secreta.
fin del programa ---- Richard Hurtado
```

Este código implementa un juego interactivo "**Ahorcado Lógico con Listas**", donde el usuario debe **descubrir una lista secreta de números** haciendo preguntas en lenguaje Python. La estructura del programa está organizada en tres etapas claramente delimitadas.

En la **Etapa 1**, se solicita al usuario (en este caso, el profesor o moderador del juego) que ingrese la cantidad de elementos que tendrá la lista secreta. Se valida que dicho número sea mayor o igual a 1. Luego, se pide que introduzca los elementos de la lista en una sola línea, separados por comas. El código incluye validaciones robustas para que se ingresen exactamente la cantidad correcta de números y que todos sean válidos. Este control previene errores y asegura una base sólida para el resto del juego.

En la **Etapa 2**, se presentan instrucciones al jugador (estudiante), explicando cómo puede interactuar con la lista secreta. El jugador tiene la libertad de ejecutar expresiones como `len(lista_secreta)` o `lista_secreta[2]`, evaluadas mediante la función `eval()` dentro de un entorno controlado, lo cual convierte el juego en una herramienta pedagógica muy valiosa.

Finalmente, en la **Etapa 3**, se desarrolla el bucle principal del juego. Si el jugador cree conocer la lista completa, puede escribir la palabra clave "adivinar" para intentar acertarla. Si falla, el programa proporciona un análisis detallado de los aciertos y errores posición por posición, en caso de que el jugador acierte completamente, el juego se termina con un mensaje de felicitación. Todo el flujo incluye retroalimentación constante, uso de emojis para mejorar la experiencia visual, y una estructura clara y funcional.

Durante la **ejecución del programa**, el jugador puede ir probando diferentes consultas para deducir los valores de la lista secreta. Si cree haber descubierto la lista completa, puede escribir el comando "adivinar" para ingresar su intento. Si el intento no es correcto, el sistema responde con una **retroalimentación detallada por posición**, indicando cuáles elementos están correctos y cuáles no, así como cuántos aciertos obtuvo en total.

13.3 Lista Secreta con getpass

```

1 import getpass
2 # ====== ETAPA 1: INGRESO DE LISTA SECRETA OCULTA ======
3 print("💡 Ingreso de la lista secreta (NO visible para los estudiantes)")
4 # Paso 1: Definir la longitud
5 while True:
6     try:
7         cantidad = int(input("¿Cuántos elementos tendrá la lista secreta?: "))
8         if cantidad < 1:
9             print("⚠ Debes ingresar al menos un número.")
10        else:
11            break
12    except ValueError:
13        print("⚠ Por favor, ingresa un número entero válido.")
14 # Paso 2: ingreso oculto uno por uno
15 lista_secreta = []
16 for i in range(cantidad):
17     while True:
18         try:
19             try:
20                 entrada = getpass.getpass(f"Ingrese el número secreto #{i + 1} (oculto): ")
21             except Exception:
22                 print("⚠ Entrada oculta no compatible. Usando entrada visible.")
23                 entrada = input(f"Ingrese el número secreto #{i + 1} (visible): ")
24             numero = int(entrada)
25             lista_secreta.append(numero)
26             break
27         except ValueError:
28             print("⚠ Ingresa un número entero válido.")
29     print(f"\n💡 Lista secreta de {cantidad} elementos cargada correctamente.\n")
30 # ====== ETAPA 2: INSTRUCCIONES PARA EL JUEGO ======
31 print("🌟 ¡Bienvenidos al juego del Ahorcado Lógico con Listas!")
32 print(f"La lista secreta tiene {len(lista_secreta)} elementos.")
33 print("En cada turno puedes hacer una 'pregunta' en forma de código Python.")
34 print("Ejemplos: len(lista_secreta), lista_secreta[2], lista_secreta[0] > 10")
35 print("Cuando creas tener la lista completa, escribe: adivinar")
36 print("-----")
37 # ====== ETAPA 3: BUCLE DE JUEGO ======
38 while True:
39     instrucion = input("">>> Escribe tu instrucción (o 'adivinar'): ").strip().lower()
40     if instrucion == "adivinar":
41         intento = input("👉 Escribe tu intento de lista (separa por comas): ")
42         try:
43             intento_lista = [int(x.strip()) for x in intento.split(",")]
44
45             if intento_lista == lista_secreta:
46                 print("🎉 ¡Correcto! Has descubierto la lista secreta.")
47                 break
48             else:
49                 print("❌ Esa no es la lista correcta.")
50                 aciertos = 0
51                 for i in range(len(lista_secreta)):
52                     if i < len(intento_lista):
53                         if intento_lista[i] == lista_secreta[i]:
54                             print(f"💡 Posición {i}: correcto ({intento_lista[i]})")
55                             aciertos += 1
56                         else:
57                             print(f"❌ Posición {i}: incorrecto (tuviste {intento_lista[i]}, se esperaba otro número)")
58                     else:
59                         print(f"⚠ Faltó ingresar un número en la posición {i}")
60                 print(f"👉 Aciertos totales: {aciertos} de {len(lista_secreta)}")
61             except ValueError:
62                 print("⚠ Error: asegúrate de ingresar solo números separados por comas.")
63         else:
64             try:
65                 # Evaluar la instrucción del alumno en un entorno controlado
66                 resultado = eval(instrucion, {"lista_secreta": lista_secreta})
67                 print("Resultado:", resultado)
68             except Exception as e:
69                 print("⚠ Error en tu instrucción:", e)
70     print("fin del programa ---- Jose Alejandro Zabala Romero")

```

Este código es una versión mejorada del juego con listas que ya se habían visto antes. La diferencia principal es que aquí la lista secreta se ingresa uno por uno y de forma oculta usando getpass, mientras que en la versión anterior se ingresaba toda la lista visible de una vez. Por lo demás, mantiene la misma estructura. El programa valida las entradas y da feedback detallado sobre aciertos y errores en cada intento. La parte compleja sigue siendo manejar la entrada oculta y el uso de eval.

Prueba

```
~/. . . /Práctico_1/vector_misterioso$ python 3list_secret.py
🌟 Ingreso de la lista secreta (NO visible para los estudiantes)
¿Cuántos elementos tendrá la lista secreta?: 3
Ingrrese el número secreto #1 (oculto):
Ingrrese el número secreto #2 (oculto):
Ingrrese el número secreto #3 (oculto):

✓ Lista secreta de 3 elementos cargada correctamente.

💡 ¡Bienvenidos al juego del Ahorcado Lógico con Listas!
La lista secreta tiene 3 elementos.
En cada turno puedes hacer una 'pregunta' en forma de código Python.
Ejemplos: len(lista_secreta), lista_secreta[2], lista_secreta[0] > 10
Cuando creas tener la lista completa, escribe: adivinar
-----
>>> Escribe tu instrucción (o 'adivinar'): 9,4,5
Resultado: (9, 4, 5)
>>> Escribe tu instrucción (o 'adivinar'): adivinar
🔴 Escribe tu intento de lista (separa por comas): 9,4,5
✗ Esa no es la lista correcta.
✓ Posición 0: correcto (9)
✗ Posición 1: incorrecto (tuviste 4, se esperaba otro número)
✓ Posición 2: correcto (5)
💡 Aciertos totales: 2 de 3
>>> Escribe tu instrucción (o 'adivinar'): adivinar
🔴 Escribe tu intento de lista (separa por comas): 9,2,5
💡 ¡Correcto! Has descubierto la lista secreta.
fin del programa ---- Jose Alejandro Zabala Romero
~/. . . /Práctico_1/vector_misterioso$
```

El programa inicia solicitando al usuario la cantidad de elementos que tendrá la lista secreta, validando que el dato sea un número entero positivo. Luego, procede a pedir uno por uno los números que componen dicha lista. Durante este ingreso, intenta ocultar la entrada para evitar que se muestre en pantalla, pero si el entorno no lo permite, cambia automáticamente a entrada visible sin interrumpir la ejecución.

Una vez que la lista está completa, el programa muestra instrucciones y entra en un ciclo donde recibe entradas del usuario. Si la entrada es una expresión Python, se evalúa usando solo la lista secreta como variable disponible, y se muestra el resultado o un mensaje de error si la expresión es inválida.

Si el usuario escribe el comando especial para adivinar, el programa solicita una lista de números separados por comas, valida que sean números enteros y compara la lista ingresada con la lista secreta. Luego informa si la adivinanza fue correcta o, en caso contrario, señala detalladamente qué elementos coinciden en posición y cuáles no, así como si faltaron números.

El ciclo se repite hasta que el usuario adivina correctamente la lista secreta, momento en que el programa imprime un mensaje de finalización y termina la ejecución.

En toda la ejecución, el programa maneja errores de entrada (como ingresar texto en lugar de números) sin detenerse, guiando al usuario para que ingrese datos válidos. Además, evita que errores en las expresiones Python ingresadas por el usuario causen una terminación inesperada.

13.4 Lista Secreta con opción "Lo siento"

```

1 import getpass
2 # ====== ETAPA 1: INGRESO DE LISTA SECRETA OCULTA ======
3 print("⚠️ Ingreso de la lista secreta (NO visible para los estudiantes)")
4 # Paso único: Ingreso de todos los valores en una línea separada por comas
5 while True:
6     try:
7         entrada_oculta = getpass.getpass("🔒 Ingrese los números secretos separados por comas (oculto): ")
8     except Exception:
9         print("⚠️ Entrada oculta no compatible en este entorno. Usando entrada visible.")
10        entrada_oculta = input("⚠️ Ingrese los números secretos separados por comas (visible): ")
11
12    lista_secreta = [int(x.strip()) for x in entrada_oculta.split(",") if x.strip() != '']
13
14    if not lista_secreta:
15        print("⚠️ La lista no puede estar vacía.")
16    else:
17        break
18    except ValueError:
19        print("⚠️ Error: asegúrate de ingresar solo números enteros separados por comas.")
20 print(f"\n✅ Lista secreta de {len(lista_secreta)} elementos cargada correctamente.\n")
21 # ====== ETAPA 2: INSTRUCCIONES PARA EL JUEGO ======
22 print("👋 ¡Bienvenidos al juego del Ahorcado Lógico con Listas!")
23 print(f"La lista secreta tiene {len(lista_secreta)} elementos.")
24 print("En cada turno puedes hacer una 'pregunta' en forma de código Python.")
25 print("Ejemplos: len(lista_secreta), lista_secreta[2], lista_secreta[0] > 10")
26 print("Cuando creas tener la lista completa, escribe: adivinar")
27 print("Si te rindes, puedes escribir: lo siento")
28 print("-----")
29 # ====== ETAPA 3: BUCLE DE JUEGO ======
30 while True:
31     instrucion = input("">>> Escribe tu instrucción (o 'adivinar' / 'lo siento'): ").strip().lower()
32     if instrucion == "adivinar":
33         intento = input("👉 Escribe tu intento de lista (separa por comas): ")
34         try:
35             intento_lista = [int(x.strip()) for x in intento.split(",")]
36
37             if intento_lista == lista_secreta:
38                 print("🎉 ¡Correcto! Has descubierto la lista secreta.")
39                 break
40             else:
41                 print("❌ Esa no es la lista correcta.")
42                 aciertos = 0
43                 for i in range(len(lista_secreta)):
44                     if i < len(intento_lista):
45                         if intento_lista[i] == lista_secreta[i]:
46                             print(f"✅ Posición {i}: correcto ({intento_lista[i]})")
47                             aciertos += 1
48                         else:
49                             print(f"❌ Posición {i}: incorrecto (tuviste {intento_lista[i]}, se esperaba otro número)")
50                     else:
51                         print(f"⚠️ Faltó ingresar un número en la posición {i}")
52                 print(f"👉 Aciertos totales: {aciertos} de {len(lista_secreta)}")
53             except ValueError:
54                 print("⚠️ Error: asegúrate de ingresar solo números separados por comas.")
55         elif instrucion == "lo siento":
56             print("👉 Gracias por jugar. Aquí estaba la lista secreta:")
57             print("✅ Lista secreta:", lista_secreta)
58             break
59         else:
60             try:
61                 resultado = eval(instrucion, {"lista_secreta": lista_secreta})
62                 print("Resultado:", resultado)
63             except Exception as e:
64                 print("⚠️ Error en tu instrucción:", e)
65     print("fin del programa ---- Jose Alejandro Zabala Romero")

```

Mantiene la misma lógica básica: ingresar una lista secreta, permitir preguntas en Python evaluadas con eval(), y que los estudiantes adivinen. Las diferencias son que aquí la lista secreta se ingresa oculta en una sola línea (con getpass y respaldo visible) y se añade la opción para que el jugador se rinda con "lo siento", mostrando la lista secreta

Prueba

```
~/.../Practico_I/vector_misterioso$ python 4list_secret.py
👤🧑 Ingreso de la lista secreta (NO visible para los estudiantes)
🔒 Ingrese los números secretos separados por comas (oculto):
✓ Lista secreta de 1 elementos cargada correctamente.

🎉 ¡Bienvenidos al juego del Ahorcado Lógico con Listas!
La lista secreta tiene 1 elementos.
En cada turno puedes hacer una 'pregunta' en forma de código Python.
Ejemplos: len(lista_secreta), lista_secreta[2], lista_secreta[0] > 10
Cuando creas tener la lista completa, escribe: adivinar
Si te rindes, puedes escribir: lo siento
-----
>>> Escribe tu instrucción (o 'adivinar' / 'lo siento'): lo siento
👉 Gracias por jugar. Aquí estaba la lista secreta:
📋 Lista secreta: [2]
```

El programa comienza solicitando al usuario ingresar todos los números secretos en una sola línea, separados por comas. Intenta ocultar la entrada para proteger la lista secreta, pero si el entorno no permite la entrada oculta, cambia a entrada visible sin interrumpir la ejecución. Durante este proceso, valida que todos los elementos ingresados sean números enteros y que la lista no esté vacía, mostrando mensajes claros en caso de error y pidiendo que se intente de nuevo.

Una vez cargada correctamente la lista secreta, se muestran las instrucciones del juego, incluyendo ejemplos de cómo interactuar con la lista a través de expresiones Python, y las opciones para adivinar o rendirse.

En el ciclo principal, el programa espera instrucciones del usuario. Si se escribe adivinar, solicita que el usuario ingrese su intento de lista, valida que contenga solo números enteros, y luego compara elemento por elemento con la lista secreta. Si el intento es correcto, felicita al jugador y termina la ejecución; si no, muestra cuáles posiciones fueron acertadas y cuáles no, ayudando al jugador a ajustar su próxima jugada.

Si el usuario escribe lo siento, el programa muestra la lista secreta para revelar la respuesta y finaliza la ejecución.

Para cualquier otra instrucción, el programa evalúa la expresión Python dada en un entorno restringido donde solo está disponible la lista secreta, y muestra el resultado o un mensaje de error si la instrucción es inválida o causa una excepción.

Durante toda la ejecución, el programa maneja entradas incorrectas y errores, evitando interrupciones inesperadas y guiando al usuario con mensajes claros para mejorar la experiencia. Al finalizar, imprime un mensaje de cierre personalizado.