**REPÚBLICA BOLIVARIANA DE VENEZUELA**

**MINISTERIO DEL PODER POPULAR**

**PARA LA EDUCACIÓN UNIVERSITARIA, CIENCIA Y TECNOLOGÍA**

**UNIVERSIDAD PLITÉCNICA TERRITORIAL DE ESTADO TRUJILLO**

**“MARIO BRICEÑO IRAGORRY”**

**PROGRAMA NACIONAL DE FORMACIÓN EN INFORMÁTICA**

**Trujillo, Edo. Trujillo**

**Manejos de Archivos, Pilas, Colas en Python y Ejemplos Práctico**

**Realizado por:**

Luis Salas C.I. 31735034

Samuell Ruíz C.I. 31735099

Jorge Briceño C.I. 30140768

**Profesor**: Carlos Bravo

**Unidad Curricular:** Programación

Trujillo, Nov de 2024

**Manejo de Archivos en Python**

El manejo de archivos en Python es un proceso fascinante que permite interactuar con datos de forma dinámica, abriendo la puerta a la creación de aplicaciones más completas y funcionales. Este concepto va mucho más allá de leer y escribir en archivos; implica la gestión eficiente de recursos, la manipulación de información almacenada y la automatización de procesos clave en el desarrollo de software.

Python facilita este manejo a través de una sintaxis intuitiva y de sus potentes funciones integradas. Usando construcciones como “ with open() para gestionar archivos, no solo se simplifica la escritura del código, sino que también se mejora su seguridad y eficiencia, al garantizar el cierre automático del archivo y evitar fugas de recursos.

En resumen, se refiere a la forma en que puedes abrir, leer, escribir y cerrar archivos en tu sistema. Los conceptos básicos son los siguientes:

1. \*\*Abrir un archivo\*\*: Usas la función `open()`, que acepta el nombre del archivo y el modo de apertura (por ejemplo, `'r'` para leer, `'w'` para escribir, `'a'` para agregar).

2. Leer un archivo: Puedes usar métodos como `.read()`, `.readline()` o `.readlines()` para obtener el contenido del archivo.

3. Escribir en un archivo: Al abrir un archivo en modo de escritura (`'w'`), puedes usar el método `.write()` para agregar contenido.

4. Cerrar un archivo: Es importante cerrar el archivo después de usarlo con el método `.close()`, así liberas recursos del sistema.

5. Manejo de excepciones: Es recomendable usar bloques `try-except` para manejar posibles errores, como que el archivo no exista.

6. Context Manager: Usar `with open(...) as ...:` es una buena práctica, ya que asegura que el archivo se cierre automáticamente al salir del bloque.

**Ejemplo Práctico**

# Leer un archivo y almacenar su contenido

with open('entrada.txt', 'r') as archivo:

contenido = archivo.readlines()

# Agregar más texto al archivo

with open('archivo\_ejemplo.txt', 'a') as archivo:

archivo.write('Esta es una nueva línea agregada al archivo.\n')

# Escribir en un archivo

with open('mi\_archivo.txt', 'w') as archivo:

archivo.write('Hola, mundo!\n')

# Leer el archivo

with open('mi\_archivo.txt', 'r') as archivo:

contenido = archivo.read()

print(contenido)

**Pilas**

Las **pilas en Python** pueden verse como una representación simplificada de decisiones que requieren reversibilidad. Imagina un asistente que va almacenando cada paso de tus acciones en una lista; si decides retroceder, solo necesitas deshacer la última acción en orden inverso. Este mecanismo natural de deshacer y rehacer es ideal para tareas como deshacer operaciones en editores de texto o navegar en un historial de páginas web. La facilidad de implementación con listas convierte a las pilas en una herramienta poderosa y flexible para gestionar estados temporales en programación. Es decir., una **pila** es una estructura de datos que sigue el principio **LIFO** (Last In, First Out), es decir, el último elemento en entrar es el primero en salir. Puedes implementarlas fácilmente usando listas con métodos como append() para agregar elementos y pop() para remover el último elemento añadido.

**Ejemplo Práctico**

pila = []

pila.append(1) # Agrega 1 a la pila

pila.append(2) # Agrega 2 a la pila

pila.append(3) # Agrega 3 a la pila

print(pila) # [1, 2, 3]

pila.pop() # Remueve y devuelve 3

print(pila) # [1, 2]

# Ejemplo: Leer línea por línea

with open('archivo\_ejemplo.txt', 'r') as archivo:

print('\nLeyendo línea por línea:')

for linea in archivo:

print(linea.strip())

class Pila:

def \_\_init\_\_(self):

self.items = []

def is\_empty(self):

return len(self.items) == 0

def push(self, item):

self.items.append(item)

def pop(self):

if not self.is\_empty():

return self.items.pop()

return None

def peek(self):

if not self.is\_empty():

return self.items[-1]

return None

**Colas en Python**

Una **cola** en Python es una estructura de datos que opera bajo el principio **FIFO** (First In, First Out), donde el primer elemento en entrar es el primero en salir. Imagina una fila en un banco: los primeros en llegar son los primeros en ser atendidos. pueden actuar como un mecanismo de **buffer de tareas**, donde cada tarea pendiente se agrega al final y se procesa desde el frente. Esto es especialmente útil en escenarios como la programación de trabajos en segundo plano, simulaciones de tráfico o en sistemas de impresión, donde las tareas deben seguir un orden natural

**Ejemplo Práctico**

# Atender al primer cliente (quitar del frente de la cola)

atendido = cola.popleft()

print(f'\nAtendiendo a: {atendido}')

print('Estado de la cola después de atender al cliente:')

print(cola)

# Agregar otro cliente

cola.append('Cliente 4')

print('\nEstado de la cola después de agregar otro cliente:')

print(cola)

# Atender a los siguientes clientes

while cola:

atendido = cola.popleft()

print(f'Atendiendo a: {atendido}')

print('\nTodos los clientes han sido atendidos.')

# Crear una cola

cola = deque()

# Añadir elementos

cola.append('a')

cola.append('b')

cola.append('c')

# Retirar elementos

primer\_elemento = cola.popleft() # 'a'

segundo\_elemento = cola.popleft() # 'b'

print(primer\_elemento) # Salida: a

print(segundo\_elemento) # Salida: b

print(cola) # Salida: deque(['c'])