Tema 14: Acceso a archivos

Lectura fácil

Acceso a Archivos de Texto en Python

Trabajar con archivos de texto es una habilidad esencial en programación, ya que permite a los programas leer y escribir información de una manera persistente, lo cual es útil para manejar configuraciones, almacenar datos de usuarios, guardar resultados y mucho más. En este documento, exploraremos cómo acceder y manipular archivos de texto en Python de manera didáctica y con muchos ejemplos para comprender todos los conceptos. Python nos ofrece varias formas sencillas y prácticas de trabajar con archivos, lo cual lo convierte en un lenguaje adecuado para tareas de manipulación de datos y automatización.

1. Abrir Archivos en Python

Para trabajar con archivos de texto en Python, utilizamos la función `open()`. Esta función nos permite abrir un archivo y realizar operaciones de lectura o escritura. El modo de apertura se especifica con un argumento adicional que indica cómo queremos manipular el archivo.

Los modos de apertura más comunes son:

- `'r'`: Modo de lectura. Abre el archivo solo para leer. El archivo debe existir. Si no se encuentra el archivo, Python genera un error.
- `'w'`: Modo de escritura. Crea un archivo nuevo o sobrescribe uno existente. Si el archivo ya tiene datos, estos se eliminarán.
- `'a'`: Modo de anexar. Abre el archivo para agregar información al final sin sobrescribirlo. Si el archivo no existe, se crea automáticamente.
- `'r+'`: Modo de lectura y escritura. Permite tanto leer como escribir en el archivo. El archivo debe existir previamente o Python genera error. La escritura se produce en modo anexar, no elimina el contenido previo del archivo.

Un ejemplo sencillo para abrir un archivo en modo de lectura sería:

```
archivo = open('mi_archivo.txt', 'r')
contenido = archivo.read()
print(contenido)
```

archivo.close()

En este ejemplo, `open()` abre el archivo `mi_archivo.txt` en modo de lectura (`'r'`). Luego usamos `read()` para leer todo el contenido del archivo y, finalmente, lo cerramos con `close()` para liberar recursos. Es importante cerrar el archivo después de usarlo para evitar consumir memoria innecesariamente y garantizar que el sistema operativo gestione correctamente los recursos.

2. Leer Archivos

Cuando se trata de leer archivos, Python nos ofrece varias maneras de hacerlo. Dependiendo de la necesidad, podemos leer el contenido completo de una sola vez o ir leyendo línea por línea.

2.1 Leer Todo el Contenido

Podemos leer todo el contenido de un archivo con el método `read()`. Esto es útil cuando queremos cargar todo el texto en memoria y trabajar con él como un bloque:

```
with open('mi_archivo.txt', 'r') as archivo:
    contenido = archivo.read()
    print(contenido)
```

Usar `with` es una buena práctica, ya que cierra automáticamente el archivo al finalizar el bloque, incluso si ocurre un error. Esto hace que nuestro código sea más seguro y fácil de mantener.

2.2 Leer Líneas Individuales

Si el archivo tiene varias líneas, podemos leerlas una a una con `readline()`. Este método lee la siguiente línea cada vez que se llama y nos permite procesar el archivo de manera incremental:

```
with open('mi_archivo.txt', 'r') as archivo:
    linea = archivo.readline()
    while linea:
        print(linea.strip())
        linea = archivo.readline()
```

En este caso, `readline()` lee una línea a la vez. Utilizamos `strip()` para eliminar los saltos de línea al final de cada línea. Esto resulta muy útil cuando queremos trabajar con cada línea de manera independiente, por ejemplo, para buscar una palabra o hacer algún procesamiento específico.

2.3 Leer Todas las Líneas como una Lista

Otra opción es usar `readlines()`, que devuelve una lista con todas las líneas del archivo. Este método es adecuado cuando queremos manipular o recorrer las líneas de forma ordenada:

```
with open('mi_archivo.txt', 'r') as archivo:
    lineas = archivo.readlines()
    for linea in lineas:
        print(linea.strip())
```

El uso de `readlines()` nos permite trabajar cómodamente con cada línea de texto, ya que las convierte en elementos de una lista que podemos manipular fácilmente con bucles o comprensiones de listas.

3. Escribir en Archivos

Para escribir en un archivo, utilizamos el modo `'w'` (escribir) o `'a'` (anexar). Estos modos determinan si queremos reemplazar todo el contenido del archivo o simplemente agregar nueva información al final.

3.1 Escribir un Archivo Nuevo

Si usamos `'w'`, creamos un archivo nuevo o sobrescribimos uno existente. Es importante tener en cuenta que, si el archivo ya existe, el contenido anterior se perderá:

```
with open('mi_archivo.txt', 'w') as archivo:
    archivo.write('Hola, este es un archivo de ejemplo.')
    archivo.write('Podemos escribir varias líneas así.')
```

Cada vez que ejecutemos este código, el contenido anterior del archivo será reemplazado. Este modo es ideal para crear nuevos archivos o escribir datos cuando no necesitamos conservar lo que había antes.

3.2 Anexar Información a un Archivo

Si queremos **agregar** información sin borrar el contenido existente, utilizamos el modo ' a ' . De esta manera, podemos agregar nuevas líneas al archivo sin afectar lo que ya contenía:

```
with open('mi_archivo.txt', 'a') as archivo:
    archivo.write('Esta es una línea adicional.')
```

Esto añade la línea al final del archivo sin borrar lo que ya contenía. Es útil para registrar información de manera acumulativa, como registros de actividad o resultados de operaciones.

4. Ejemplos Prácticos

4.1 Contar Líneas, Palabras y Caracteres

Podemos crear un script que lea un archivo y cuente el número de líneas, palabras y caracteres:

```
with open('mi_archivo.txt', 'r') as archivo:
    lineas = archivo.readlines()
    num_lineas = len(lineas)
    num_palabras = sum(len(linea.split()) for linea in lineas)
    num_caracteres = sum(len(linea) for linea in lineas)

print(f'Líneas: {num_lineas}, Palabras: {num_palabras}, Caracteres:
{num_caracteres}')
```

En este ejemplo, usamos `split()` para dividir las líneas en palabras y contar cuántas hay en cada línea. Esta técnica es muy útil para realizar análisis básicos de archivos de texto y obtener estadísticas rápidas.

4.2 Copiar el Contenido de un Archivo a Otro

Podemos crear una copia de un archivo leyendo su contenido y escribiéndolo en otro archivo. Esto puede ser útil para realizar respaldos o duplicar información:

```
with open('mi_archivo.txt', 'r') as archivo_lectura:
    contenido = archivo_lectura.read()

with open('copia_archivo.txt', 'w') as archivo_escritura:
    archivo_escritura.write(contenido)
```

Este ejemplo muestra cómo podemos trabajar con múltiples archivos y realizar operaciones con ellos, lo cual es fundamental para muchos tipos de tareas automatizadas.

5. Uso del Modo Binario

Además de trabajar con archivos de texto, Python permite trabajar con **archivos binarios** (como imágenes o archivos ejecutables) usando los modos `'rb'` (lectura binaria) y `'wb'` (escritura binaria). Estos modos son útiles cuando necesitamos manipular datos que no están en formato de texto:

```
with open('imagen.jpg', 'rb') as archivo_binario:
    contenido = archivo_binario.read()

with open('copia_imagen.jpg', 'wb') as archivo_copia:
    archivo_copia.write(contenido)
```

Esto es útil cuando queremos copiar archivos que no son de texto, como fotos, videos o cualquier otro archivo binario. Trabajar con archivos binarios nos permite manejar cualquier tipo de dato, sin importar el formato.

6. Gestión de Errores al Trabajar con Archivos

Cuando trabajamos con archivos, es común encontrarse con errores (como intentar abrir un archivo que no existe). Podemos manejar estos errores usando **try-except**, lo cual ayuda a que nuestros programas sean más robustos y no se detengan inesperadamente:

```
try:
    with open('archivo_inexistente.txt', 'r') as archivo:
        contenido = archivo.read()
except FileNotFoundError:
    print('El archivo no fue encontrado.')
```

Esto ayuda a que el programa no se detenga si ocurre un error, y en su lugar muestra un mensaje amigable para el usuario. Esto es especialmente importante en aplicaciones que necesitan manejar diferentes tipos de archivos que podrían no estar siempre disponibles.

7. Ejemplos Adicionales

7.1 Leer y Modificar un Archivo

Podemos leer el contenido de un archivo, modificarlo y guardarlo nuevamente. Esto es útil cuando necesitamos actualizar o corregir datos:

```
with open('mi_archivo.txt', 'r') as archivo:
    lineas = archivo.readlines()

# Agregar un prefijo a cada línea
nuevas_lineas = [f'Prefijo: {linea}' for linea in lineas]

with open('mi_archivo.txt', 'w') as archivo:
    archivo.writelines(nuevas_lineas)
```

Este ejemplo lee todas las líneas, les añade un prefijo y luego guarda el resultado en el mismo archivo. Este tipo de procesamiento es útil para tareas como añadir marcas de tiempo, identificar autores, o simplemente agregar contexto adicional a cada línea del archivo.

7.2 Contar Ocurrencias de una Palabra

Podemos buscar cuántas veces aparece una palabra específica en un archivo. Esto es útil para analizar el contenido de textos o realizar búsquedas:

```
palabra_a_buscar = 'Python'
contador = 0

with open('mi_archivo.txt', 'r') as archivo:
    for linea in archivo:
        contador += linea.lower().count(palabra_a_buscar.lower())

print(f'La palabra "{palabra_a_buscar}" aparece {contador} veces en el archivo.')
```

Este script convierte todas las líneas a minúsculas antes de buscar la palabra, lo que permite realizar una búsqueda sin diferenciar