Tema 12: Procesamiento de archivos

Héctor Xavier Limón Riaño

May 31, 2024

Contents

1 Introducción

- Los archivos son una forma de almacenar información

- El sistema operativo se encarga de gestionarlos mediante el sistema de archivos
- Hay dos tipos principales de archivos (aunque existen más tipos dependiendo del SO):
  - Directorios: Son una forma de agrupar archivos
  - Archivos normales: pueden ser de dos tipos:
    * Texto: requieren de alguna codificación especial como utf-8, sirven para almacenar texto plano
    * Binarios: cualquier archivo que no sea de texto es binario, sirven para cualquier propósito (música, vídeo, juegos, etc.)

1.1 Rutas de archivo

- En el sistema todos los archivos se almacenan en alguna ruta lógica
- Esa ruta depende de una estructura de árbol
- El sistema de archivos se estructura mediante un árbol, donde cada directorio representa un nodo
- En un sistema tipo Unix (como Linux, MacOS y BSD) el directorio raíz es simplemente `/`
- En sistemas windows cada dispositivo de almacenamiento se identifica con una letra (como C) y tiene su propio árbol de directorios
- En sistemas tipo Unix cada subdirectorio de la ruta se separa mediante una `/`, por ejemplo:

  ```
  /home/xl666/oneDrive/Dropbox/clases/estructurasCiber/notas/notasEstructuras24/tema
  ```

- En sistemas windows se puede usar `/` o `\` para separar cada subdirectorio
- En la parte final de la ruta puede venir el nombre de un archivo normal

1.1.1 Rutas absolutas

- Si la ruta se da desde la raíz

1.1.2 Rutas relativas

- No empieza con diagonal
- Es relativa al directorio de trabajo actual
- Rutas especiales relativas:
  - . referencia a la ruta actual
  - .. referencia al directorio padre

2 Procesamiento de archivos de texto

- `Python` cuenta con un API muy simple y útil para el procesamiento de archivos
- Se verán las siguientes operaciones comunes con archivos:
  - Abrir un archivo existente para lectura
  - Volcar contenido completo de un archivo de lectura
  - Recorrer línea por línea un archivo de lectura
  - Cerrar archivos
  - Abrir un archivo existente o nuevo para escritura
  - Abrir un archivo existente para escribir al final
  - Volcar buffer de escritura

2.1 Abrir un archivo existente para lectura

- Para abrir archivos se utiliza la función `open`
- A esta función se le pasan principalmente dos parámetros:
  - Ruta del archivo
  - Modo de uso del archivo: si es de texto, binario, lectura, escritura o append (se verán todos más adelante)
- La función regresa un objeto especial que representa al archivo y permite manipularlo

```
archivo = open('/tmp/nuevo.txt', 'tr')
# t quiere decir modo texto
# r quiere decir modo lectura (read)

# por defecto, open usa modo tr, así que no es necesario ponerlo
# la siguiente línea es equivalente
archivo = open('/tmp/nuevo.txt')
```

2.2 Volcar contenido completo de un archivo de lectura

- Una vez se abrió el archivo en modo lectura se puede leer todo el contenido y pasarse a cadena mediante el método `read` del objeto archivo
- Una vez se lee el contenido completo, no se puede volver a leer a menos que regreses explícitamente al principio del archivo (operación seek)

```
archivo = open('/tmp/nuevo.txt')
contenido = archivo.read()
archivo.close() # cerrar, explicado más adelante
```

- CUIDADO: sólo se debe usar esta operación si el archivo no es demasiado grande (cabe en RAM) de lo contrario es muy ineficiente
- Se verá a continuación una forma alterna de leer el contenido del archivo sin cargarlo por completo a RAM

2.3 Recorrer línea por línea un archivo de lectura

- Mediante `for` es posible recorrer un archivo de texto línea por línea
- Es una forma muy común y conveniente en `Python`
- Evita que se cargue todo el contenido en RAM, en cambio se procesa línea por línea
- A este tipo de procesamiento se le conoce en programación como `perezoso` o `lazy`, se verá más sobre esto es un curso posterior de programación

```
for linea in open('nuevo.txt'):
    print(linea)
```

- Una ventaja extra de usar `for` de esta forma, es que no es necesario cerrar el archivo, se cierra automáticamente al terminar el `for`

2.4 Cerrar archivos

- Cuando se abre un archivo se asocia un recurso al proceso que abre el archivo
- A este recurso se le conoce como `descriptor de archivo`
- Cada proceso tiene un número finito de `descriptores de archivos`
- Si un proceso se excede en este número, al querer abrir un nuevo archivo habrá una excepción
- Es buena práctica, para evitar este problema, cerrar el archivo siempre que se termine de usarlo
- Esto se logra directamente con el método `close` como se vio antes

```
ar = open('nuevo.txt')
contenido = ar.read()
ar.close()
```

- También se puede lograr de forma indirecta usando un `for` como se explicó antes, o mediante la sentencia especial `with`
- `with` permite crear un contexto especial de ejecución, en el caso de archivos, permite que se cierren de forma robusta (aunque haya errores) al terminar el bloque `with`
- En general se recomienda usar `with` siempre que se pueda

```
# equivalente
archivo = open('nuevo.txt')
contenido = archivo.read()
archivo.close()

# mejor

with open('nuevo.txt') as archivo:
    contenido = archivo.read()

# al terminar el bloque se cierra el archivo automáticamente
```

2.5 Abrir un archivo existente o nuevo para escritura

- Hay que tener cuidado con esta operación
- Si la ruta de archivo que se proporciona a `open` no existe, entonces se crea un archivo nuevo (siempre y cuando los subdirectorios de la ruta si existan)
- Si la ruta es de un archivo existente, éste será sobre escrito, esto es, se perderá su contenido original

```
archivo = open('/tmp/nuevo.txt', 'tw')
# equivalente, la t está por defecto:
archivo = open('/tmp/nuevo.txt', 'w')

# para escribir, tener cuidado si se quieren saltos de línea:
archivo.write('cadena a agregar\n')
archivo.write('otra línea\n')
arhivo.close()
```

2.6 Abrir un archivo existente para escribir al final

- Como ya se mencionó, al utilizar la configuración 'w' de `open` se sobreescribe en el archivo si es que existe
- Para evitar esto, se puede usar la configuración 'a' para activar el modo `append`, en este modo no se sobreescribe el archivo, sino que se escribe al final de éste
- Si el archivo no existe se crea al igual que con 'w'

```
with open('/tmp/nuevo.txt', 'a') as archivo:
    archivo.write('nueva linea') # se agrega al final
```

2.7 Volcar buffer de escritura

- Cuando se escribe en un archivo, el contenido no se va directamente a disco, se escribe temporalmente en un buffer de memoria
- Este es un mecanismo tradicional para limitar el acceso a disco, ya que es ineficiente (sobre todo en comparación al acceso a memoria)

- Hasta que el buffer no se llena no se pasa el contenido a disco
- Esto quiere decir que puedes estar escribiendo en tu archivo y no ver cambios en él al abrirlo
- Si quieres volcar el contenido a disco directamente se puede de dos formas:
  - Cierra el archivo (con `close` por ejemplo)
  - Utiliza el método `flush`

```
with open('/tmp/nuevo.txt', 'w') as archivo:
    archivo.write('hola')
    archivo.flush() # se va a disco
    archivo.write('otra cosa')

# al terminar el bloque with se cierra el archivo y se vuelca el buffer
```

2.8 Ejercicios sugeridos

- Hacer un script que recibe un archivo de texto todo en minúsculas y genera un nuevo archivo de texto con todas las cadenas en mayúsculas
- Hacer un script que recibe un archivo `passwd` con información de usuarios de un sistema tipo `Linux` y determina qué usuarios son los que pueden iniciar sesión en el sistema (explicado en clase)

3 Procesamiento de archivos binarios

- El procesamiento de archivos binarios es más complejo que el de texto
- Es necesario primero familiarizarse con cadenas binarias
- Las cadenas binarias son similares a las cadenas de texto, incluso comparten un API similar
- `Python` facilita mucho el manejo de binario gracias a la noción de cadenas binarias (en muchos otros lenguajes suele ser más complejo)
- Un cadena binaria esencialmente contiene una serie de bytes
- El byte es la unidad mínima que procesa la computadora (no se procesan bits individuales)

- En `Python` una cadena binaria literal empieza con la letra `b`

```
binario = b'hola mundo'
print(type(binario))
# mostrar bytes internos
print(list(binario))
```

```
<class 'bytes'>
[104, 111, 108, 97, 32, 109, 117, 110, 100, 111]
```

- `Python` hace su mejor esfuerzo para tratar de imprimir como texto una cadena binaria, pero si no puede muestra símbolos especiales llamados `code points`

```
import os

cadena = b'hola'
# cadena aleatoria de bytes
aleatoria = os.urandom(12)
print(type(aleatoria))
print(aleatoria)
# bytes internos
print(list(aleatoria))
```

- Cualquier archivo se puede tratar como binario, incluso los archivos de texto (internamente todo es binario)
- Se verá cómo realizar las siguientes operaciones con archivos binarios:
  - Abrir archivo binario
  - Leer un archivo binario
  - Escribir en archivo binario

3.1 Abrir archivo binario

- Se utiliza la configuración 'b' de `open`
- También es obligatorio establecer el modo (r, w, a)

```
ar = open('nuevo.bin', 'rb')
ar.close()
```

3.2 Leer archivo binario

- Es un poco diferente a como se leen archivos de texto

3.2.1 Leer todo el contenido de golpe

- Igual que con archivos de texto, se utiliza el método `read`
- Hay que tener la misma precaución de que el archivo no sea demasiado grande

```
with open('archivo.bin', 'rb') ar archivo:
    contenido = archivo.read()
```

1. Determinar tamaño de archivo
    - Se puede hacer mediante `os.path.getsize`

    ```
    print(os.path.getsize('tema12.org'))
    ```

3.2.2 Leer por pedazos

- A los pedazos se les llama `chunks`
- Se refiere a leer por bloques de bytes
- `read` puede recibir como primer parámetro el número de bytes que se quiere leer

```
with open('archivo.bin', 'rb') as archivo:
    pedazo = archivo.read(100) # leer 100 bytes
    otro = archivo.read(100) # leer los siguientes 100 bytes
```

- Se puede mandar a llamar a `read` varias veces hasta finalizar la lectura del archivo
- Cada vez que se llama a `read` se lee el siguiente pedazo del archivo
- Si quedan menos bytes en el archivo de lo que se pasa en `read` se lee hasta donde es posible, sin generar errores
- Si se trata de leer un archivo que ya ha sido terminado de leer, simplemente se regresa cadena binaria vacía `b''` y no se generan errores

- La cadena binaria vacía es una forma de saber que ya se terminó de leer el archivo
- Por ejemplo, se puede leer la totalidad del archivo de la siguiente manera con un `while`

```
with open('archivo.bin', 'rb') as archivo:
    chunk = archivo.read(1024):
    contenido = chunk
    while chunk:
        chunk = archivo.read(1024)
        contenido += chunk
print(contenido) # todo el contenido del archivo
```

3.2.3 Leer con un for

- Se comporta similar a un archivo de texto
- La diferencia es que no se lee línea por línea, sino chunk por chunk
- Osea que se hace básicamente lo mismo que pasando el parámetro de tamaño de chunk a `read`
- El tamaño del chunk está determinado por aspectos del SO
- Se puede saber el tamaño del chunk de la siguiente manera

```
import io
print(io.DEFAULT_BUFFER_SIZE)
```

```
8192
```

```
contenido = b''
for chunk in open('archivo.bin', 'rb'):
    contenido += chunk

print(contenido)
```

- Notar que al usar `for` se cierra automáticamente el archivo

3.3 Escribir en archivo binario

- Básicamente igual que la escritura en archivos de texto, sólo que se escriben cadenas binarias
- Se puede usar el modo `'w'` o el modo `'a'` como se explicó antes
- Por ejemplo, para crear un archivo binario aleatorio:

```
import os

with open('/tmp/archivo.bin', 'wb') as archivo:
    for _ in range(100): # 100 cadenas aleatorias
        chunk = os.urandom(10)
        archivo.write(chunk)
```

3.4 Ejercicios sugeridos

- Oculta una imagen en un archivo pdf (estenografía)
- Recupera la imagen a partir del archivo alterado