Unidad 11

Manejo de archivos

Veremos en esta unidad cómo manipular archivos desde nuestros programas. Los archivos permiten almacenar información que persistirá luego de que el programa finalice su ejecución. Los archivos también se pueden compartir o ser transmitidos entre diferentes computadoras, mediante dispositivos de almacenamiento o redes como Internet.

11.1 ¿Qué es un archivo?

Un archivo no es otra cosa más que una secuencia de bytes. Por ejemplo, un archivo puede contener la secuencia de 4 bytes: `48 6f 6c 61` (notación hexadecimal).

Sabías que...

Un byte está formado de 8 bits, y según el valor de cada uno de los bits, un byte puede representar $2^8 = 256$ combinaciones diferentes. Si asignamos un valor numérico a cada una de esas combinaciones, comenzando de 0, con un byte podemos representar cualquier número entre 0 y 255.

Cuando se representan datos binarios, en lugar de utilizar la notación binaria (base 2) o la decimal (base 10), se suele utilizar la notación *hexadecimal* (base 16).

En Python se puede escribir un número en notación binaria con el prefijo `0b`, y en notación hexadecimal con el prefijo `0x`. Además, las funciones `bin` y `hex` permiten obtener la representación de un número en binario y hexadecimal, respectivamente.

```
>>> 0b11111101
253
>>> 0xfd
253
>>> bin(0xfd)
'0b11111101'
>>> hex(253)
'0xfd'
```

Binario	Decimal	Hexadecimal
00000000	0	00
00000001	1	01
00000010	2	02
00000011	3	03
00000100	4	04
00000101	5	05
00000110	6	06
00000111	7	07
00001000	8	08
00001001	9	09
00001010	10	0a
00001011	11	0b
00001100	12	0c
00001101	13	0d
00001110	14	0e
00001111	15	0f
00010000	16	10
...	...	...
11111101	253	fd
11111110	254	fe
11111111	255	ff

Un archivo se identifica con un *nombre*, por ejemplo `hola.txt`. Para facilitar la gestión y búsqueda eficiente, los archivos se organizan en *carpetas* y *subcarpetas*, formando una estructura jerárquica: cada carpeta puede contener archivos y otras carpetas, permitiendo una organización lógica y estructurada de la información.

La ubicación de un archivo se identifica mediante una *ruta*, que es una cadena formada por la secuencia de carpetas y subcarpetas que lleva a dicho archivo desde la *carpeta raíz* (`/`). Por ejemplo, si nuestro archivo se encuentra dentro de la carpeta `home` y subcarpeta `alan`, su ruta sería `"/home/alan/hola.txt"`[1]. Una ruta se puede escribir en forma *absoluta* (comenzando con `/` y conteniendo la secuencia completa de carpetas y subcarpetas desde la carpeta raíz), o *relativa* a alguna carpeta (sin comenzar con `/`). En nuestro ejemplo, la ruta del archivo relativa a la carpeta `"/home"` sería `"alan/hola.txt"`.

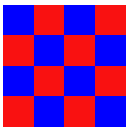
11.2 Formatos de archivos

Para cualquier información que se desee almacenar en un archivo, se debe elegir una codificación que permita representar esa información mediante una secuencia de bytes.

Por ejemplo, si deseamos almacenar el texto `"Hola mundo!"` en un archivo, lo más simple sería elegir la codificación ASCII, en la que cada caracter se almacena como 1 byte. De esta manera el archivo contendría en total 11 bytes: `48 6f 6c 61 20 6d 75 6e 64 6f 21`. Según la codificación ASCII, el caracter `H` se codifica con el valor hexadecimal `48`, el caracter `o` con el valor `6f`, y así sucesivamente.

ASCII es un ejemplo de un *formato de codificación de texto*, pero no es el único. La limitación principal del formato ASCII es que solo permite representar 128 caracteres. El formato UTF-8 es más complejo que ASCII, ya que representa cada caracter con una cantidad variable de bytes, pero a cambio permite representar más de un millón de caracteres. UTF-8 es la codificación que se usa por defecto en la mayoría de los sistemas.

Si en lugar de texto deseamos almacenar una imagen en un archivo, tendríamos que elegir otra codificación. En la Figura 11.1 se muestra una imagen codificada en formato BMP. Notar que en este caso el algoritmo de codificación no es tan directo o intuitivo.



```
4d 42 00 ba 00 00 00 00 00 00 8a 00 00 00 7c 00 00 00 04 00 00 00 04 00 00
00 01 00 18 00 00 00 00 00 30 00 00 2e 23 00 00 2e 23 00 00 00 00 00 00 00
00 00 00 00 00 ff ff 00 00 00 00 ff 00 00 00 00 00 00 47 42 73 52 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00 00 00
00 00 00 00 00 00 00 00 11 11 ff f9 00 00 11 11 ff f9 00 00 00 ff 11 00 f9
00 ff 11 00 f9 11 11 11 ff f9 00 00 11 11 ff f9 00 00 00 ff 11 00 f9 11 00
11 00 f9 11
```

Figura 11.1: Una imagen de 4x4 pixels codificada en formato BMP.

BMP, JPG y PNG son algunos ejemplos de formatos de codificación de imágenes. También hay formatos de codificación para sonido, video y cualquier otro tipo de información. Se dice que todos estos son *formatos binarios*, en contraste con los formatos de texto.[2]

[1] En sistemas Windows se utiliza el caracter \ como separador en lugar del caracter `/`. Sin embargo, al especificar rutas en nuestros programas Python se acepta que utilicemos el separador `/`; de esta manera nuestros programas podrán funcionar en cualquier sistema operativo.

[2] La distinción entre formatos «de texto» y «binarios» es histórica, pero cabe destacar que técnicamente todos los formatos son binarios, incluso los de texto.

Notar que el archivo está definido únicamente por su contenido (la secuencia de bytes), independientemente de lo que representen esos bytes. Si solo tenemos el contenido de un archivo dado, no hay manera de saber qué codificación se utilizó para crear esos datos; es decir, de qué formato es el archivo. Es por eso que se utiliza la convención de agregar una *extensión* al nombre del archivo para indicar su formato. Por ejemplo, a los archivos de texto se les agrega la extensión `.txt`, a los archivos en formatp BMP se les agrega la extensión `.bmp`, etc. Si bien es una práctica recomendable, no es obligatorio que el nombre del archivo incluya una extensión, o que concuerde con el formato real del archivo.

A continuación veremos cómo manipular archivos en nuestros programas Python.

11.3 Abrir un archivo

En Python hay dos *modos* para acceder a un archivo:

- **Modo texto** en el que podemos leer o escribir cadenas de texto (`str`), que serán codificadas automáticamente según la codificación elegida (ASCII, UTF-8, etc.). En la mayoría de los sistemas, el modo por defecto es texto con codificación UTF-8.
- **Modo binario** en el que no se aplica ninguna codificación, y leemos o escribimos datos de tipo `bytes`.

Para poder leer o escribir un archivo, primero debemos pedirle permiso al sistema operativo. Esta operación se llama *abrir* el archivo. En Python, para abrir un archivo usaremos la función `open`, que recibe la ruta del archivo a abrir.

```
archivo = open("archivo.txt")
```

Esta función intentará abrir el archivo con el nombre indicado, por defecto en modo texto. Si tiene éxito devolverá un valor de un tipo especial, que nos permitirá manipular el archivo de diversas maneras.

11.4 Leer un archivo de texto

La operación más sencilla a realizar sobre un archivo es leer su contenido. Podemos utilizar la función `read` para leer uno o más caracteres. Por ejemplo, para almacenar en `s` una cadena con los 3 primeros caracteres leidos del archivo:

```
s = archivo.read(3)
```

Si volvemos a invocar a `read(3)` se leerán los próximos 3 caracteres del archivo. Esto es así ya que cada archivo que se encuentre abierto tiene una posición asociada, que indica el último punto que fue leído. Cada vez que se lee un byte, avanza esa posición.

Lo más usual al trabajar con archivos de texto es procesarlos línea por línea; es decir, leer hasta que encontramos el caracter *nueva línea* o `\n`. La función `readline` hace esto de forma automática:

```
linea = archivo.readline()
while linea != '':
    # procesar linea
    linea = archivo.readline()
```

El lenguaje Python nos permite hacer lo mismo de una manera más abreviada:

Sabías que...

Los archivos de texto son sencillos de manejar, pero existen por lo menos 3 formas distintas de marcar un fin de línea. En Unix tradicionalmente se usa el caracter '\n' (código ASCII 10, definido como «nueva línea») para el fin de línea, mientras que en los sitemas de Apple el fin de línea se solía representar como un '\r' (valor ASCII 13, definido como retorno de carro) y en Windows se usan ambos caracteres '\r\n'.

Al leer archivos en modo texto, Python acepta cualquier tipo de fin de línea como si fuese un \n. Al escribir archivos, Python elegirá automáticamente el modo más apropiado. Si queremos modificar este comportamiento podemos especificar el modo utilizando la opción `newline` de la función `open`.

```
for linea in archivo:
    # procesar linea
```

De esta manera, la variable `linea` irá almacenando distintas cadenas correspondientes a cada una de las líneas del archivo.

Es posible, además, obtener todas las líneas del archivo utilizando una sola llamada a función:

```
lineas = archivo.readlines()
```

En este caso, la variable `lineas` tendrá una lista de cadenas con todas las líneas del archivo.

Atención

Es importante tener en cuenta que cuando se utilizan funciones como `archivo.readlines()`, se está cargando en la memoria de la computadora el contenido completo del archivo. Siempre que una instrucción cargue un archivo completo en memoria debe tenerse cuidado de utilizarla sólo con archivos pequeños, ya que de otro modo podría agotarse la memoria de la computadora.

11.5 Cerrar un archivo

Al terminar de trabajar con un archivo, es importante cerrarlo, por diversos motivos: en algunos sistemas los archivos sólo pueden ser abiertos de a un programa por la vez; en otros, lo que se haya escrito no se guardará realmente hasta no cerrar el archivo; o el límite de cantidad de archivos que puede manejar un programa puede ser bajo, etc.

Para cerrar un archivo simplemente se debe llamar a:

```
archivo.close()
```

Además, Python nos provee con una estructura que permite cerrar el archivo automáticamente, sin necesidad de llamar a `close`:

```
with open("archivo.txt") as archivo:
    #
    # procesar el archivo
    #

# Cuando la ejecución sale del bloque 'with',
# el archivo se cierra automáticamente.
```

11.6 Ejemplo: procesamiento de archivos de texto

Por ejemplo, para mostrar todas las líneas de un archivo, precedidas por el número de línea, podemos hacerlo de la siguiente manera:

```
archivo = open("archivo.txt")
i = 1
for linea in archivo:
    linea = linea.rstrip("\n")
    print(f"{i}: {linea}")
    i += 1
archivo.close()
```

La llamada a `rstrip` es necesaria ya que cada línea que se lee del archivo contiene un caracter especial llamado *fin de línea* y con la llamada a `rstrip("\n")` se remueve.

Notar que sería equivalente usar el bloque `with` para ahorrarnos la llamada a `close`:

```
with open("archivo.txt") as archivo:
    i = 1
    for linea in archivo:
        linea = linea.rstrip("\n")
        print(f"{i}: {linea}")
        i += 1
```

También podemos utilizar la función `enumerate` (explicada en la sección 7.2.3) para no tener que mantener el contador `i` a mano:

```
with open("archivo.txt") as archivo:
    for i, linea in enumerate(archivo):
        linea = linea.rstrip("\n")
        print(f"{i}: {linea}")
```

11.7 Modo de apertura de los archivos

La función `open` recibe un parámetro opcional para indicar el modo en que se abrirá el archivo. Como ya mencionamos, un archivo se puede abrir en **modo texto** (`'t'`) o **modo binario** (`'b'`).

También se pueden especificar tres modos en cuanto a los permisos de lectura y escritura:

- Modo de **sólo lectura** (`'r'`). En este caso no es posible realizar modificaciones sobre el archivo, solamente leer su contenido.
- Modo de **sólo escritura** (`'w'`). En este caso el archivo es truncado (vaciado) si existe, y se lo crea si no existe.
- Modo **sólo escritura posicionándose al final del archivo** (`a`). En este caso se crea el archivo, si no existe, pero en caso de que exista se posiciona al final, manteniendo el contenido original.

Por ejemplo, para abrir un archivo en modo binario y escritura:

```
archivo = open("imagen.jpg", "wb")
```

En cualquiera de los modos `r`, `w` o `a` se puede agregar un `+` para pasar a un modo lectura-escritura. El comportamiento de `r+` y de `w+` no es el mismo, ya que en el primer caso se tiene el archivo completo, y en el segundo caso se trunca el archivo, perdiendo así los datos.

Si un archivo no existe y se lo intenta abrir en modo lectura, se generará un error; en cambio si se lo abre para escritura, Python se encargará de crear el archivo al momento de abrirlo, ya sea con `'w'`, `'a'`, `'w+'` o con `'a+'`.

En caso de que no se especifique el modo, los archivos serán abiertos en modo sólo lectura (`r`).

Atención

Si un archivo existente se abre en modo escritura (`'w'` o `'w+'`), todos los datos anteriores son borrados y reemplazados por lo que se escriba en él.

11.8 Escribir un archivo de texto

De la misma forma que para la lectura, existen dos formas distintas de escribir a un archivo. Mediante cadenas:

```
archivo.write(cadena)
```

O mediante listas de cadenas:

```
archivo.writelines(lista_de_cadenas)
```

Así como la función `readline` devuelve las líneas con los caracteres de fin de línea (`\n`), será necesario agregar los caracteres de fin de línea a las cadenas que se vayan a escribir en el archivo.

Por ejemplo, el siguiente programa genera a su vez el código de otro programa Python y lo guarda en el archivo `saludo.py`:

```
with open("saludo.py", "w") as saludo:
    saludo.write("# Este programa fue generado por otro programa!\n")
    saludo.write("print('Hola Mundo')\n")
```

Atención

Si un archivo existente se abre en modo lectura-escritura, al escribir en él se sobreescribirán los datos anteriores, a menos que se haya llegado al final del archivo.

Este proceso de sobreescritura se realiza caracter por caracter, sin consideraciones adicionales para los caracteres de fin de línea ni otros caracteres especiales.

11.9 Agregar información a un archivo

Abrir un archivo en modo *agregar al final* puede parece raro, pero es bastante útil.

Uno de sus usos es para escribir un archivo de bitácora (o archivo de *log*), que nos permita ver los distintos eventos que se fueron sucediendo, y así encontrar la secuencia de pasos (no siempre evidente) que hace nuestro programa.

Esta es una forma muy habitual de buscar problemas o hacer un seguimiento de los sucesos. Para los administradores de sistemas es una herramienta esencial de trabajo.

En el Código 11.1 se muestra un módulo para manejo de logs, que se encarga de la apertura del archivo, del guardado de las líneas una por una y del cerrado final del archivo.

Código 11.1 `log.py`**:** Módulo para manipulación de archivos de log

```
# El módulo datetime se utiliza para obtener la fecha y hora actual.
import datetime

def abrir(nombre_log):
    """Abre el archivo de log indicado. Devuelve el archivo abierto.
    Pre: el nombre corresponde a un nombre de archivo válido.
    Post: el archivo ha sido abierto posicionándose al final."""
    archivo_log = open(nombre_log, "a")
    guardar(archivo_log, "Iniciando registro de errores")
    return archivo_log

def guardar(archivo_log, mensaje):
    """Guarda el mensaje en el archivo de log, con la hora actual.
    Pre: el archivo de log ha sido abierto correctamente.
    Post: el mensaje ha sido escrito al final del archivo."""
    # Obtiene la hora actual en formato de texto
    hora_actual = str(datetime.datetime.now())
    # Guarda la hora actual y el mensaje de error en el archivo
    archivo_log.write(f"[{hora_actual}] {mensaje}\n")

def cerrar(archivo_log):
    """ Cierra el archivo de log.
    Pre: el archivo de log ha sido abierto correctamente.
    Post: el archivo de log se ha cerrado. """
    guardar(archivo_log, "Fin del registro de errores")
    archivo_log.close()
```

En este módulo se utiliza el módulo de Python `datetime` para obtener la fecha y hora actual que se guardará en los archivos. Es importante notar que en el módulo mostrado no se abre o cierra un archivo en particular, sino que las funciones están programadas de modo tal que puedan ser utilizadas desde otro programa.

Se trata de un módulo genérico que podrá ser utilizado por diversos programas, que requieran la funcionalidad de registrar los posibles errores o eventos que se produzcan durante la ejecución.

Para utilizar este módulo, será necesario primero llamar a `log.abrir()` para abrir el archivo de log, luego llamar a `log.guardar()` por cada mensaje que se quiera registrar, y finalmente llamar a `log.cerrar()` cuando se quiera concluir la registración de mensajes:

```
import log

archivo_log = log.abrir("log.txt")
# ...
# Hacer cosas que pueden dar error
if error:
    log.guardar(archivo_log, "ERROR: " + error)
# ...
# Finalmente cerrar el archivo
log.cerrar(archivo_log)
```

Este código, que incluye el módulo `log` mostrado anteriormente, muestra una forma básica de utilizar un archivo de log. Al iniciarse el programa se abre el archivo de log, de forma que

queda registrada la fecha y hora de inicio. Posteriormente se realizan tareas varias que podrían provocar errores, y de haber algún error se lo guarda en el archivo de log. Finalmente, al terminar el programa, se cierra el archivo de log, quedando registrada la fecha y hora de finalización.

El archivo de log generado tendrá la forma:

```
2016-04-10 15:20:32.229556 Iniciando registro de errores
2016-04-10 15:20:50.721415 ERROR: no se pudo acceder al recurso
2016-04-10 15:21:58.625432 ERROR: formato de entrada inválido
2016-04-10 15:22:10.109376 Fin del registro de errores
```

11.10 Archivos binarios

Para abrir un archivo y manejarlo de forma binaria es necesario agregarle una `'b'` al parametro de modo:

```
archivo_binario = open('imagen.jpg', 'rb')
```

Para procesar el archivo de a bytes en lugar de líneas, se utiliza la función `contenido = archivo.read(n)` para leer `n` bytes y `archivo.write(contenido)`, para escribir `contenido` en la posición actual del archivo.

Al manejar un archivo binario, frecuentemente es necesario conocer la posición actual en el archivo y poder modificarla. Para obtener la posición actual se utiliza `archivo.tell()`, que indica la cantidad de bytes desde el comienzo del archivo.

Para modificar la posición actual se utiliza `archivo.seek(posicion)`, que permite desplazarse hacia el byte ubicado en la posición indicada.

```
>>> f = open('imagen.jpg', 'rb')
>>> f.tell()
0
>>> datos = f.read(3)
>>> datos
b'\xff\xd8\xff'
>>> type(datos)
<class 'bytes'>
>>> f.tell()
3
>>> f.seek(0)
0
>>> datos = f.read() # leer el contenido completo del archivo
>>> len(datos)
3150
```

> **Atención**
>
> Al trabajar con archivos binarios, la función `read` no devuelve cadenas de caracteres (`str`), sino que devuelve una *secuencia de bytes* (`bytes`). Análogamente, la función `write` recibe una secuencia de bytes.

11.11 Persistencia de datos

Se llama **persistencia** a la capacidad de guardar la información de un programa para poder volver a utilizarla en otro momento. Es lo que los usuarios conocen como *Guardar el archivo* y

después *Abrir el archivo*. Pero para un programador puede significar más cosas y suele involucrar un proceso de *serialización* de los datos a un archivo o a una base de datos o a algún otro medio similar, y el proceso inverso de recuperar los datos a partir de la información *serializada*.

Por ejemplo, supongamos que en el desarrollo de un juego se quiere guardar en un archivo la información referente a los ganadores, el puntaje máximo obtenido y el tiempo de juego en el que obtuvieron ese puntaje.

En el juego, esa información podría estar almacenada en una lista de tuplas:

```
[(nombre1, puntaje1, tiempo1), (nombre2, puntaje2, tiempo2), ...]
```

Esta información se puede guardar en un archivo de muchas formas distintas. En este caso, para facilitar la lectura del archivo de puntajes para los humanos, se decide guardarlos en un archivo de texto, donde cada tupla ocupará una línea y los valores de las tuplas estarán separados por comas.

En el Código 11.2 se muestra un módulo capaz de guardar y recuperar los puntajes en el formato especificado.

Código 11.2 `puntajes.py`**:** Módulo para guardar y recuperar puntajes en un archivo

```
1  def guardar_puntajes(nombre_archivo, puntajes):
2      """Guarda la lista de puntajes en el archivo.
3      Pre: nombre_archivo corresponde a un archivo válido,
4           puntajes corresponde a una lista de tuplas de 3 elementos.
5      Post: se guardaron los valores en el archivo,
6            separados por comas.
7      """
8
9      with open(nombre_archivo, "w") as f:
10         for nombre, puntaje, tiempo in puntajes:
11             f.write(f"{nombre},{puntaje},{tiempo}\n")
12
13 def recuperar_puntajes(nombre_archivo):
14     """Recupera los puntajes a partir del archivo provisto.
15     Devuelve una lista con los valores de los puntajes.
16     Pre: el archivo contiene los puntajes en el formato esperado,
17          separados por comas
18     Post: la lista devuelta contiene los puntajes en el formato:
19           [(nombre1,puntaje1,tiempo1),(nombre2,puntaje2,tiempo2)].
20     """
21
22     puntajes = []
23     with open(nombre_archivo, "r") as f:
24         for linea in f:
25             nombre, puntaje, tiempo = linea.rstrip("\n").split(",")
26             puntajes.append((nombre, int(puntaje), tiempo))
27     return puntajes
```

Dadas las especificaciones del problema al guardar los valores en el archivo, es necesario convertir el puntaje (que es un valor numérico) en una cadena, y al abrir el archivo es necesario convertirlo nuevamente en un valor numérico.

Es importante notar que tanto la función que almacena los datos como la que los recupera requieren que la información se encuentre de una forma determinada y de no ser así, fallarán. Es por eso que estas condiciones se indican en la documentación de las funciones como sus precondiciones. En próximas unidades veremos cómo evitar que falle una función si alguna de sus condiciones no se cumple.

Es bastate sencillo probar el módulo programado y ver que lo que se guarda es igual que lo que se recupera:

```
>>> import puntajes
>>> valores = [("Pepe", 108, "4:16"), ("Juana", 2315, "8:42")]
>>> puntajes.guardar_puntajes("puntajes.txt", valores)
>>> recuperado = puntajes.recuperar_puntajes("puntajes.txt")
>>> print(recuperado)
[('Pepe', 108, '4:16'), ('Juana', 2315, '8:42')]
```

Guardar el estado de un programa se puede hacer tanto en un archivo de texto, como en un archivo binario. En muchas situaciones es preferible guardar la información en un archivo de texto, ya que de esta manera es posible modificarlo fácilmente desde cualquier editor de textos.

En general, los archivos de texto consumen más espacio, pero son más faciles de entender y fáciles de usar desde cualquier programa.

Por otro lado, en un archivo binario bien definido se puede evitar el desperdicio de espacio, o también hacer que sea más eficiente acceder a los datos.

En definitiva, la decisión de qué formato usar queda a discreción del programador. Es importante recordar que el sentido común es el valor más preciado en un programador.

11.11.1 Persistencia en archivos CSV

Un formato que suele usarse para transferir datos entre programas es **CSV** (del inglés *comma separated values*: valores separados por comas). Es un formato bastante sencillo, tanto para leerlo como para procesarlo desde el código, parecido al formato visto en el ejemplo anteriormente.

```
Nombre,Apellido,Telefono,Cumpleaños
"John","Smith","555-0101","1973-11-24"
"Jane","Smith","555-0101","1975-06-12"
```

En el ejemplo se puede ver una pequeña base de datos. En la primera línea del archivo tenemos los nombres de los campos, un dato opcional desde el punto de vista del procesamiento de la información, pero que facilita entender el archivo.

En las siguientes lineas se ingresan los datos de la base de datos, cada campo separado por comas. Los campos que son cadenas se suelen escribir entre comillas dobles. Si alguna cadena contiene alguna comilla doble se la reemplaza por `\"` y una contrabarra se escribe como `\\`.

En Python es bastante sencillo procesar de este tipo de archivos, tanto para la lectura como para la escritura, mediante el módulo `csv`.

La funciones del ejemplo anterior podrían programarse mediante el módulo `csv`. En el Código 11.3 se muestra una posible implementación que utiliza este módulo. Si se prueba este código, se obtiene un resultado idéntico al obtenido anteriormente.

El código en este caso es muy similar, ya que en el ejemplo original se hacían muy pocas consideraciones al respecto de los valores: se asumía que el primero y el tercero eran cadenas mientras que el segundo necesitaba ser convertido a cadena.

Código 11.3 `puntajes_csv.py`**:** Módulo para guardar y recuperar puntajes en un archivo CSV

```
import csv

def guardar_puntajes(nombre_archivo, puntajes):
    """Guarda la lista de puntajes en el archivo.
    Pre: nombre_archivo corresponde a un archivo válido,
         puntajes corresponde a una lista de secuencias de elementos.
    Post: se guardaron los valores en el archivo,
          separados por comas.
    """

    with open(nombre_archivo, "w") as f:
        archivo_csv = csv.writer(f)
        archivo_csv.writerows(puntajes)

def recuperar_puntajes(nombre_archivo):
    """Recupera los puntajes a partir del archivo provisto.
    Devuelve una lista con los valores de los puntajes.
    Pre: el archivo contiene los puntajes en el formato esperado,
         separados por comas
    Post: la lista devuelta contiene los puntajes en el formato:
          [(nombre1,puntaje1,tiempo1),(nombre2,puntaje2,tiempo2)].
    """

    puntajes = []
    with open(nombre_archivo, "r") as f:
        archivo_csv = csv.reader(f)
        for nombre, puntaje, tiempo in archivo_csv:
            puntajes.append((nombre, int(puntaje), tiempo))
    return puntajes
```

Es importante notar, entonces, que al utilizar el módulo `csv` en lugar de hacer el procesamiento en forma manual, se obtiene un comportamiento más robusto, ya que el módulo `csv` tiene en cuenta muchos más casos que nuestro código original no. Por ejemplo, el código anterior no tenía en cuenta que el nombre pudiera contener una coma.

En el apéndice de esta unidad puede verse una aplicación completa de una agenda, que almacena los datos del programa en archivos CSV.

11.11.2 Persistencia en archivos binarios

En el caso de que decidiéramos grabar los datos en un archivo binario, Python incluye varios módulos que pueden ser de ayuda. Entre ellos se destacan los módulos `pickle` y `struct`.

El módulo `pickle` permite codificar automáticamente un valor de cualquier tipo a una secuencia de bytes, y luego decodificarlo. Hay que tener en cuenta, sin embargo, que no es nada simple acceder a un archivo en el formato `pickle` desde un programa que no esté escrito en Python.

El módulo `struct` permite codificar y decodificar a mano cada fragmento de información. Es bastante más complicado de usar que `pickle`, pero es esencial si deseamos codificar datos binarios de forma tal que podamos decodificarlos en otros sistemas o lenguajes.

En el Código 11.4 se muestra el mismo ejemplo de almacenamiento de puntajes, utilizando el módulo `pickle`.

Código 11.4 `puntajes_pickle.py`**:** Módulo para guardar y recuperar puntajes en un archivo que usa *pickle*

```
import pickle

def guardar_puntajes(nombre_archivo, puntajes):
    """Guarda la lista de puntajes en el archivo.
    Pre: nombre_archivo corresponde a un archivo válido,
         puntajes corresponde a los valores a guardar
    Post: se guardaron los valores en el archivo en formato pickle.
    """

    with open(nombre_archivo, "wb") as f:
        pickle.dump(puntajes, f)

def recuperar_puntajes(nombre_archivo):
    """Recupera los puntajes a partir del archivo provisto.
    Devuelve una lista con los valores de los puntajes.
    Pre: el archivo contiene los puntajes en formato pickle
    Post: la lista devuelta contiene los puntajes en el
          mismo formato que se los almacenó.
    """

    with open(nombre_archivo, "r") as f:
        return pickle.load(f)
```

En el apéndice de esta unidad puede verse una aplicación completa de una agenda, que utiliza el módulo `struct` para almacenar datos en archivos.

11.12 Resumen

- Para utilizar un archivo desde un programa, es necesario abrirlo, y cuando ya no se lo necesite, se lo debe cerrar.
- Las instrucciones más básicas para manejar un archivo son leer y escribir.
- Cada archivo abierto tiene relacionada una posición que se puede consultar o cambiar.
- Los archivos de texto se procesan generalmente línea por línea y sirven para intercambiar información entre diversos programas o entre programas y humanos.
- Los archivos binarios se procesan generalmente byte por byte. Suelen ser más eficientes al ser interpretados por una computadora, pero son ilegibles para humanos.
- Es posible acceder de forma secuencial a los datos, o se puede ir accediendo a posiciones en distintas partes del archivo, dependiendo de cómo esté almacenada la información y qué se quiera hacer con ella.
- Es posible leer todo el contenido de un archivo y almacentarlo en una única variable, pero si el archivo es muy grande puede consumir memoria innecesariamente.

Referencia Python

`archivo = open(nombre[, modo])`

Abre un archivo. `nombre` es el nombre completo del archivo, `modo` especifica si se va usar para lectura (`'r'`), escritura truncando el archivo (`'w'`), o escritura agregando al final del archivo (`'a'`), agregándole un `'+'` al modo el archivo se abre en lectura-escritura, agregándole una `'b'` el archivo se maneja como archivo binario, agregándole `'U'` los fin de línea se manejan a mano.

`archivo.close()`

Cierra el archivo.

`with open(nombre) as archivo:`

Abre el archivo para procesar dentro del bloque `with`. El archivo se cerrará automáticamente al salir del bloque.

`linea = archivo.readline()`

Lee una línea de texto del archivo
Si la cadena devuelta es vacía, es que se ha llegado al final del archivo.

`for linea in archivo:`

Itera sobre las lineas del archivo.

`lineas = archivo.readlines()`

Devuelve una lista con todas las líneas del archivo.

`datos = archivo.read([n])`

Si se trata de un archivo de texto, devuelve la cadena de `n` caracteres situada en la posición actual del archivo.
Si se trata de un archivo binario, devuelve una secuencia de `n` bytes.
Si la secuencia devuelta es vacía, es que se ha llegado al final del archivo.
De omitirse el parámetro `n`, devuelve todo el contenido del archivo.

`archivo.write(contenido)`

Escribe `contenido` en la posición actual de `archivo`.

`posicion = archivo.tell()`

Devuelve un número que indica la posición actual en `archivo`, equivalente a la cantidad de bytes desde el comienzo del archivo.

`archivo.seek(posicion)`

Modifica la posición actual en `archivo`, trasladándose hasta el byte `posicion`.

`os.path.exists(ruta)`

Indica si la ruta existe o no. No nos dice si es una carpeta, un archivo u otro tipo de archivo especial del sistema.

`os.path.isfile(ruta)`

Indica si la ruta existe y es un archivo.

`os.path.isdir(ruta)`

Indica si la ruta existe y es una carpeta (directorio).

`os.path.join(ruta, ruta1[, ... rutaN])`

Une las rutas con el caracter de separación de carpetas que le corresponda al sistema en uso.

11.13 Ejercicios

Ejercicio 11.13.1. Escribir una función, llamada `head` que reciba un archivo y un número `N` e imprima las primeras `N` líneas del archivo.

Ejercicio 11.13.2. Escribir una función, llamada `cp`, que copie todo el contenido de un archivo (sea de texto o binario) a otro, de modo que quede exactamente igual.
Nota: Tener en cuenta que el contenido completo del archivo puede ser más grande que la memoria de la computadora. Utilizar `archivo.read(bytes)` para leer como máximo una cantidad determinada de bytes.

Ejercicio 11.13.3. Escribir una función, llamada `wc`, que dado un archivo de texto, lo procese e imprima por pantalla cuántas líneas, cuantas palabras y cuántos caracteres contiene el archivo.

Ejercicio 11.13.4. Escribir una función, llamada `grep`, que reciba una cadena y un archivo de texto, e imprima las líneas del archivo que contienen la cadena recibida.

Ejercicio 11.13.5. Escribir una función, llamada `rot13`, que reciba un archivo de texto de origen y uno de destino, de modo que para cada línea del archivo origen, se guarde una línea *cifrada* en el archivo destino. El algoritmo de cifrado a utilizar será muy sencillo: a cada caracter comprendido entre la a y la z, se le suma 13 y luego se aplica el módulo 26, para obtener un nuevo caracter.

Ejercicio 11.13.6. Sea una lista de números enteros entre -2.147.483.648 y 2.147.483.647.

a) Escribir una función `guardar_numeros` que reciba la lista y una ruta, y guarde el contenido de la lista en el archivo, en modo texto, escribiendo un número por línea.

b) Escribir una función `cargar_numeros` que reciba una ruta a un archivo con el formato anterior y devuelva la lista de números cargada.

c) Modificar las funciones anteriores para que almacenen el archivo en formato binario, almacenando cada número en 4 bytes. Analizar las ventajas y desventajas entre el formato de texto y binario.

Ejercicio 11.13.7. Sea un diccionario cuyas claves y valores son cadenas.

a) Escribir una función `guardar_diccionario` que reciba un diccionario y una ruta, y guarde el contenido del diccionario en el archivo, en modo texto, escribiendo un par clave-valor por línea con el formato `clave,valor`.

b) Escribir una función `cargar_diccionario` que reciba una ruta a un archivo con el formato anterior y devuelva el diccionario cargado.

Ejercicio 11.13.8. Sea una imagen de 8x8 pixels, en el que cada pixel puede ser blanco o negro. La imagen se representa en Python como una matriz de 8x8 valores `bool`, donde un valor `True` o `False` representa un pixel blanco o negro respectivamente.

a) Escribir funciones para leer y escribir una imagen en un formato de texto. En este formato, el archivo contiene 8 líneas de 8 caracteres, y cada caracter representa un pixel. Un pixel blanco o negro se representa con un caracter ASCII `B` o `N` respectivamente.

b) ⋆ Escribir funciones para leer y escribir una imagen en un formato binario. En este formato, la imagen se almacena en un archivo que contiene exactamente 64 bits (8 bytes), donde cada bit representa un pixel, y un pixel blanco o negro se representa con un bit 1 o 0, respectivamente.

Apéndice 11.A Agenda con archivos CSV

A continuación se muestra un programa de agenda que utiliza archivos CSV. Luego se muestran los cambios necesarios para que la agenda que utilice archivos en formato binario utilizando el módulo `struct`, en lugar de CSV.

`agenda-csv.py` Agenda con los datos en formato CSV

```
import csv, os, datetime

RUTA = "agenda.csv"
FORMATO_FECHA = "%d/%m/%Y"

def cargar_agenda(ruta):
    """Carga todos los datos del archivo en una lista y la devuelve."""
    agenda = []
    if not os.path.exists(ruta):
        return agenda
    with open(ruta) as f:
        datos_csv = csv.reader(f)
        encabezado = next(datos_csv)
        for item in datos_csv:
            nombre, apellido, telefono, nacimiento = item
            agenda.append((nombre, apellido, telefono, cadena_a_fecha(
    ↪ nacimiento)))
    return agenda

def guardar_agenda(agenda, ruta):
    """Guarda la agenda en el archivo."""
    with open(ruta, "w") as f:
        datos_csv = csv.writer(f)
        # cabecera:
        datos_csv.writerow(("Nombre", "Apellido", "Telefono", "FechaNacimiento"
    ↪ ))
        for item in agenda:
            nombre, apellido, telefono, nacimiento = item
            datos_csv.writerow((nombre, apellido, telefono, fecha_a_cadena(
    ↪ nacimiento)))

def leer_busqueda():
    """Solicita al usuario nombre y apellido y los devuelve."""
    nombre = input("Nombre: ")
    apellido = input("Apellido: ")
    return nombre, apellido

def buscar(nombre, apellido, agenda):
    """Busca el primer item que coincida con nombre y con apellido."""
    for item in agenda:
        if nombre in item[0] and apellido in item[1]:
            return item
    return None

def menu_alta(nombre, apellido, agenda):
    """Pregunta si se desea ingresar un nombre y apellido y
```

```
44          de ser así, pide los datos al usuario."""
45      print(f"No se encuentra {nombre} {apellido} en la agenda.")
46      confirmacion = input("¿Desea ingresarlo? (s/n): ")
47      if confirmacion.lower() != "s":
48          return
49      telefono = input("Telefono: ")
50      nacimiento = input("Fecha de nacimiento (dd/mm/aaaa): ")
51      agenda.append([nombre, apellido, telefono, cadena_a_fecha(nacimiento)])
52
53  def mostrar_item(item):
54      """Muestra por pantalla un item en particular."""
55      nombre, apellido, telefono, nacimiento = item
56      print()
57      print(f"{nombre} {apellido}")
58      print(f"Telefono: {telefono}")
59      print(f"Fecha de nacimiento (dd/mm/aaaa): {fecha_a_cadena(nacimiento)}")
60      print()
61
62  def menu_item():
63      """Muestra por pantalla las opciones disponibles para un item
64         existente."""
65      o = input("b: borrar, m: modificar, ENTER para continuar (b/m): ")
66      return o.lower()
67
68  def modificar(viejo, nuevo, agenda):
69      """Reemplaza el item viejo con el nuevo, en la lista datos."""
70      indice = agenda.index(viejo)
71      agenda[indice] = nuevo
72
73  def menu_modificacion(item, agenda):
74      """Solicita al usuario los datos para modificar una entrada."""
75      nombre = input("Nuevo nombre: ")
76      apellido = input("Nuevo apellido: ")
77      telefono = input("Nuevo teléfono: ")
78      nacimiento = input("Nueva fecha de nacimiento (dd/mm/aaaa): ")
79      modificar(item, [nombre, apellido, telefono, cadena_a_fecha(nacimiento)],
        ↪ agenda)
80
81  def baja(item, agenda):
82      """Elimina un item de la lista."""
83      agenda.remove(item)
84
85  def confirmar_salida():
86      """Solicita confirmación para salir"""
87      confirmacion = input("¿Desea salir? (s/n): ")
88      return confirmacion.lower() == "s"
89
90  def agenda():
91      """Función principal de la agenda.
92      Carga los datos del archivo, permite hacer búsquedas, modificar
93      borrar, y al salir guarda. """
94      agenda = cargar_agenda(RUTA)
95      while True:
96          nombre, apellido = leer_busqueda()
```

```
        if nombre + apellido == "":
            if confirmar_salida():
                break
        item = buscar(nombre, apellido, agenda)
        if not item:
            menu_alta(nombre, apellido, agenda)
            continue
        mostrar_item(item)
        opcion = menu_item()
        if opcion == "m":
            menu_modificacion(item, agenda)
        elif opcion == "b":
            baja(item, agenda)
    guardar_agenda(agenda, RUTA)

def fecha_a_cadena(fecha):
    """Convierte una fecha de tipo `date` a una cadena"""
    return fecha.strftime(FORMATO_FECHA)

def cadena_a_fecha(s):
    """Convierte una cadena a una fecha de tipo `date`"""
    return datetime.datetime.strptime(s, FORMATO_FECHA).date()

agenda()
```

Apéndice 11.B Agenda con archivos binarios

`agenda-struct.py` Modificaciones a la agenda para guardar los datos en formato binario, utilizando el módulo `struct`

```
import struct, os, datetime

FORMATO_FECHA = "%d/%m/%Y"

RUTA = "agenda.dat"
STRUCT_CANTIDAD_ITEMS = struct.Struct("I") # 4 bytes, entero sin signo
STRUCT_LONGITUD_CADENA = struct.Struct("H") # 2 bytes, entero sin signo
STRUCT_FECHA = struct.Struct("BBH") # 1 byte, 1 byte, 2 bytes, enteros sin
    ↪ signo
CODIFICACION_CADENAS = 'utf-8'

def cargar_agenda(ruta):
    """Carga todos los datos del archivo en una lista y la devuelve."""
    agenda = []
    if not os.path.exists(ruta):
        return agenda
    with open(ruta, "rb") as f:
        (n,) = STRUCT_CANTIDAD_ITEMS.unpack(f.read(STRUCT_CANTIDAD_ITEMS.size))
        for _ in range(n):
            nombre = leer_cadena(f)
            apellido = leer_cadena(f)
            telefono = leer_cadena(f)
```

```
            d, m, y = STRUCT_FECHA.unpack(f.read(STRUCT_FECHA.size))
            nacimiento = datetime.date(y, m, d)
            agenda.append((nombre, apellido, telefono, nacimiento))
    return agenda

def guardar_agenda(agenda, ruta):
    """Guarda la agenda en el archivo."""
    with open(ruta, "wb") as f:
        f.write(STRUCT_CANTIDAD_ITEMS.pack(len(agenda)))
        for item in agenda:
            nombre, apellido, telefono, nacimiento = item
            escribir_cadena(f, nombre)
            escribir_cadena(f, apellido)
            escribir_cadena(f, telefono)
            d, m, y = nacimiento.day, nacimiento.month, nacimiento.year
            f.write(STRUCT_FECHA.pack(d, m, y))

def escribir_cadena(f, nombre):
    """Escribe una cadena de longitud variable en el archivo"""
    b = bytes(nombre, CODIFICACION_CADENAS)
    f.write(STRUCT_LONGITUD_CADENA.pack(len(b)))
    f.write(b)

def leer_cadena(f):
    """Lee una cadena de longitud variable del archivo"""
    (n,) = STRUCT_LONGITUD_CADENA.unpack(f.read(STRUCT_LONGITUD_CADENA.size))
    b = f.read(n)
    return b.decode(CODIFICACION_CADENAS)
```