Ficha 22

Archivos

1.] Introducción.

En todos los programas que se desarrollaron hasta aquí existía un problema común: cada vez que el programa se ejecutaba, se debía volver a cargar todos sus datos y el programa volvía a calcular todos los resultados. Pero al terminar la ejecución de un programa, ni los datos cargados ni los resultados calculados quedaban *resguardados* de forma que pudieran volver a usarse si fuera necesario en otra corrida posterior. Todas las estructuras de datos que hasta aquí se estudiaron (arreglos de cualquier dimensión, registros, arreglos de registros, pilas, colas, etc.) tenían ese inconveniente, y debido a ello se dice que son *estructuras de datos de naturaleza volátil*.

Es obvio que esa situación es poco útil: ya se sabe lo tedioso e impráctico que resulta cargar *n* registros por teclado en un arreglo cada vez que un programa arranca, o cargar todos los valores de nuevo en un arreglo cada vez que se ejecuta un programa para verificar un algoritmo de ordenamiento o de búsqueda. Además, muchas veces un programa genera como resultado un arreglo de varios componentes, pero cuando el programa termina ese arreglo se pierde, y no queda disponible para que pueda ser usado en otros programas... Y estamos nombrando situaciones de programación que sólo se dan en las aulas: a nivel profesional es absolutamente inútil desarrollar un sistema complejo basado sólo en estructuras volátiles. Es fácil comprender que el sistema de información de un banco *debe* tener almacenados en *forma permanente* (y no volátil) los datos de todos sus clientes y movimientos de cuentas: sería inaceptable que los empleados del centro de cómputos del banco deban volver a cargar todos los datos para que el banco pueda operar cada vez que abre sus puertas por la mañana, (y aún cuando así lo hicieran, no les bastaría un solo día para cargar todos los datos: la cantidad de clientes y movimientos de cuentas que tiene un banco es enorme)[1].

Para dar una solución a este inconveniente existen los *archivos*, que son conjuntos de datos *persistentes* (es decir, no volátiles): un *archivo* es una estructura de datos que en lugar de almacenarse en memoria principal, se almacena en dispositivos de almacenamiento

[1] Olvidar absolutamente todo suele tener consecuencias desastrosas, y el cine ha explorado en películas muy conocidas esa situación. En 2013, la película *Oblivion* (conocida en español como "*Oblivion: El Tiempo del Olvido*"), dirigida por *Joseph Kosinski* y protagonizada por *Tom Cruise,* gira alrededor de un planeta Tierra devastado por la guerra contra una civilización extraterrestre. Los invasores fueron derrotados pero la Tierra es inhabitable por la radiación, por lo que los sobrevivientes están abandonando en forma sistemática el planeta. Distintos grupos de trabajo de alta tecnología se encargan de vigilar los trabajos de transporte de toda el agua del mar hacia una nave gigante en el espacio, para llevarla consigo cuando se produzca la migración final. Pero uno de los vigilantes sospecha que algo anda mal… y finalmente descubre que su memoria ha sido borrada y reconfigurada, entre otros detalles…

externos (como discos o memorias flash) y por lo tanto el contenido de un *archivo* no se pierde al finalizar el programa que lo utiliza [1].

El uso de *archivos* permite el manejo de grandes volúmenes de datos sin tener que volverlos a cargar desde teclado cada vez que se desee procesarlos. Por otra parte, una vez que un *archivo* fue creado y grabado en un dispositivo externo, ese *archivo* puede ser utilizado por cualquier otro programa, ya sea para obtener datos del archivo y/o para modificar los datos del mismo (siempre y cuando la estructura interna de ese archivo sea conocida por quien desarrolla esos otros programas).

Al igual que ocurre con la representación de cualquier tipo de valor en la memoria de un computador, los datos que se graban en un archivo se representan en sistema binario y por cada dato se utilizan tantos bytes como sea necesario para representar ese dato. De allí que, entonces, el *tamaño de un archivo* es la *cantidad total de bytes* que contiene el archivo.

Si bien, como acabamos de indicar, en ***todos*** los archivos se representa su contenido en base al sistema binario, en la práctica es común hacer una distinción entre los llamados *archivos de texto* y los llamados *archivos binarios* [2]:

- ***Archivos de texto*:** Todos los bytes del archivo son *interpretados* como *caracteres* (en base, por ejemplo, a la tabla ASCII). Si un archivo es de texto también contiene bytes, *pero se asume que todos esos bytes representan caracteres que pueden ser visualizados en pantalla*, con la única excepción del caracter de salto de línea (que solemos representar en Python como "\n"). En la tabla ASCII, por caso, los primeros 32 caracteres (numerados entre el 0 y el 31) no tienen representación visual y se designan como *caracteres de control* (de hecho, el caracter número 13 es el que corresponde al *retorno de carro* (que en Windows implica también un *salto de línea*) y lo denotamos como "\n"). Pero desde el caracter 32 (que es el espacio en blanco) en adelante, todos tienen representación visual (por ejemplo, el caracter 65 es la letra "A"). Pues bien: si un archivo es "de texto", debería contener solamente bytes cuyos valores numéricos sean mayores o iguales a 32, con la sola excepción del byte que representa un salto de línea[2]. Son archivos de texto, por ejemplo, los archivos fuente en Python de cada programa que hemos desarrollado a lo largo del curso, así como los archivos fuente de cualquier otro lenguaje. Estos archivos no sólo pueden ser abiertos y editados desde un IDE para el respectivo lenguaje, sino que pueden ser abiertos y editados con *cualquier editor de textos* que conozca.

- ***Archivos binarios*:** Las secuencias o bloques de bytes que el archivo contiene *representan información de cualquier tipo* (números en formato binario, caracteres, valores booleanos, etc.) y ***no se asume*** que cada byte representa un caracter. En sí mismo, el archivo no distingue si los bytes grabados pertenecen a un número entero o a un número flotante o a una cadena de caracteres: es el *programador* el que determina cómo interpretar el contenido. Un archivo binario puede entonces contener bytes que en caso de ser interpretados como caracteres llevarían a elementos de la tabla de conversión (ASCII u otra) que no tengan representación visual, haciendo que en su lugar el programa muestre caracteres de reemplazo, o bien haría que todos los bytes sean mostrados como caracteres

[2] En algunos archivos de texto se admite la inclusión adicional de caracteres de control e información a modo de marcas para darle formato al texto (por ejemplo, incorporando distintos tipos de letras (como *Arial* o *Courier*) y estilos (como *italic* o *bold*)). En ese sentido, los archivos de texto que sólo contienen bytes con valores desde el 32 en adelante y ocasionalmente algún salto de línea, *sin inclusión de información de formato*, se suelen designar también como *archivos de texto plano* para indicar que se está hablando de los archivos de texto más simples posibles.

provocando una secuencia ininteligible de caracteres en la pantalla. Archivos binarios típicos son, por ejemplo, los archivos ejecutables *.exe* de la plataforma Windows, o cualquier archivo que contenga una imagen, o (en general) *cualquier archivo que* ***no sea*** *específicamente de texto*.

Quede claro: ***todos*** los archivos contienen información representada en binario, y por lo tanto y en ese sentido, *todos los archivos son binarios*. Pero en la práctica, se habla de *archivos de texto* y *archivos binarios* para distinguir a aquellos en los que sólo se espera encontrar bytes que representen caracteres visualizables (los *archivos de texto*) de aquellos en los que no se asume nada respecto del significado previo de cada byte y la forma de interpretarlos (los *archivos binarios*) [1].

En esta Ficha de estudio nos concentraremos en el tratamiento de archivos binarios, dejando la gestión y aplicación de archivos de texto para una ficha posterior. Además, si bien es perfectamente posible almacenar datos de cualquier tipo en un archivo binario, nuestro enfoque estará orientado al trabajo con *archivos binarios en los que se guardarán registros*, debido a que el registro es una estructura de datos que permite describir muy bien a cualquier entidad de la cual se quiera almacenar datos; y analizaremos además distintas estrategias de aplicación para combinar el uso de arreglos, registros y archivos en situaciones de programación típicas.

2.] Archivos binarios en Python: conceptos básicos.

En Python el acceso a datos almacenados en dispositivos externos en forma de archivos, o a datos gestionados mediante algún otro tipo de recurso (como un buffer interno, o como el sistema estándar de entrada/salida) se realiza mediante *objetos*[3] conocidos como *file objects* o como *file-like objects*. Estos objetos se crean y se dejan disponibles para el programador a través de la función interna *open()* de Python y a partir de allí cada *file object* brinda acceso a una colección de métodos (funciones contenidas en el objeto) que facilitan el manejo de los datos representados por ese objeto [2] [3].

La siguiente instrucción en Python crea una variable *m* asociada a un *file object* y deja abierto el archivo *datos.dat* en modo de *grabación*:

```
m = open("datos.dat", "w")
```

El primer parámetro de la función es una cadena de caracteres con el *nombre físico* (o *file descriptor*) del archivo a abrir: es el nombre con el cual el archivo figura grabado en el sistema de carpetas del sistema operativo. Este nombre físico puede incluir también la ruta de acceso a ese archivo:

```
m = open("c:\\documents\\datos.dat", "w")
```

Si no se indica la ruta de acceso, el intérprete Python lo buscará en la carpeta actual del proyecto o programa que se está ejecutando.

[3] Esencialmente, un *objeto* es una variable similar a un registro, pero tal que además de contener *campos* de datos (llamados *atributos* en un objeto) contienen también *funciones* (designadas como *métodos* cuando forman parte de un objeto).

El segundo parámetro es otra cadena de caracteres indicando el *modo de apertura para el archivo*. Estos modos en general son los mismos que los del lenguaje C, y con los mismos significados (ver *Tabla 1*). Al igual que en C, si se agrega una 'b' como último caracter en la cadena que especifica el modo de apertura entonces el archivo será tratado como *archivo binario*. Si en lugar de una 'b' se agrega una 't' (o no se agrega ninguna de ambas) entonces el archivo será considerado como un archivo de texto:

Tabla 1: Distintos modos de apertura de un archivo en Python.

Modo	Significado
r (o **rt**)	El archivo se abre como *archivo de texto* en *modo de solo lectura* (no está permitido grabar). No será creado en caso de no existir previamente. Este es el modo por defecto si se invoca a *open()* sin especificar modo de apertura alguno.
w (o **wt**)	El archivo se abre como archivo *de texto* en *modo de solo grabación*. Si ya existía, su contenido se perderá y se abrirá vacío. Si el archivo no existía, será creado.
a (o **at**)	El archivo se abre como archivo *de texto* en *modo de solo append* (todas las grabaciones se hacen al final del archivo, *preservando* su contenido previo si el archivo ya existía). Si no existía, será creado.
r+ (o **r+t**)	El archivo se abre como *archivo de texto* en *modo de lectura y grabación*. El archivo debe existir previamente: no será creado en caso de no existir.
w+ (o **w+t**)	El archivo se abre como archivo *de texto* en *modo de grabación y lectura*. Si ya existía su contenido será eliminado y abierto vacío. Si no existía, será creado.
a+ (o **a+t**)	El archivo se abre como archivo *de texto* en modo de *lectura y de append* (todas las *grabaciones* se hacen al final del archivo, preservando su contenido previo). Si no existía, será creado.
rb	El archivo se abre como *archivo binario* en *modo de sólo lectura*. No será creado en caso de no existir previamente.
wb	El archivo se abre como archivo *binario* en *modo de sólo grabación*. Si ya existía su contenido será eliminado. Si no existía, será creado.
ab	El archivo se abre como archivo *binario* en *modo de sólo append* (todas las grabaciones se hacen al final del archivo, preservando su contenido previo si el archivo ya existía). Si no existía, será creado.
r+b	El archivo se abre como *archivo binario* en *modo de lectura y grabación*. El archivo debe existir previamente: no será creado en caso de no existir.
w+b	El archivo se abre como archivo *binario* en *modo de grabación y lectura*. Si ya existía su contenido será eliminado. Si no existía, será creado.
a+b	El archivo se abre como archivo *binario* en *modo de lectura y de append* (todas las *grabaciones* se hacen al final del archivo, preservando su contenido previo si ya existía). Si no existía, será creado.

La función *open()* es la encargada de abrir el canal de comunicación entre el dispositivo que contiene al archivo y la memoria principal. El objeto creado y retornado por la función (que en nuestros ejemplo se designó con la variable *m*) se aloja en memoria y contiene diversos atributos con datos que permiten manejar el archivo. Por ese motivo, ese objeto se suele designar como el *manejador del archivo* [1].

Así como es necesario abrir un archivo para poder acceder a su contenido, también es necesario *cerrarlo* (o sea, cerrar el canal de comunicación) cuando se termine de usarlo. El método *close()* se usa para cerrar esa conexión y liberar además cualquier recurso que estuviese asociado al archivo. Luego de invocar a *close()*, la variable u objeto que representaba al archivo (el manejador del archivo) queda indefinida. Note, además, que en Python los archivos son cerrados automáticamente cuando la variable para accederlos sale

del ámbito en que fue definida, por lo que en muchos contextos no es estrictamente necesario que el programador invoque a *close()*. Pero en general, el programador debe estar seguro que el archivo que estuvo manejando se cierre oportunamente, ya sea en forma manual o en forma automática. La siguiente secuencia elemental de instrucciones abre un archivo en modo 'wb'. Si el archivo no existía, será entonces creado. E inmediatamente luego de crearlo/abrirlo, se procede a cerrarlo, sin grabar nada en su interior:

```
m = open('datos.dat', 'wb')
m.close()
```

Lo único que hará el script anterior será crear el archivo *datos.dat* en la carpeta del proyecto, dejándolo vacío (tamaño = 0).

El modo de apertura es muy importante al invocar a *open()*: el script anterior crea el archivo si el mismo no existía o lo abre y elimina su contenido si ya existía. Si en lugar de 'wb' se usase el modo 'ab', el efecto será el mismo, pero con una pequeña diferencia: si el archivo ya existiese, su contenido no será eliminado y los nuevos datos se agregarán al final (por lo cual el modo 'ab' suele ser el que se usa para permitir agregar nuevos datos a un archivo). Ahora bien: si el modo de apertura fuese 'rb' (sólo lectura) y el archivo que se quiere abrir no existe en el momento de invocar a *open()*, entonces el programa lanzará un *error de runtime* (lo que se conoce como una *excepción*) y se interrumpirá:

```
m = open('noexiste.num', 'rb')
```

```
Traceback (most recent call last):
576
  File "C:/[F22] Archivos/test00.py", line 20, in <module>
    test()
  File "C:/[F22] Archivos/test00.py", line 11, in test
    m = open('noexiste.num', 'rb')
FileNotFoundError: [Errno 2] No such file or directory: 'noexiste.num'
```

Los objetos de tipo file (*file object*) que se crean con *open()* contienen numerosos métodos adicionales para el manejo del archivo. Entre ellos, existen los métodos *read()* y *write()* que permiten respectivamente *leer y grabar datos en el archivo* [2]. Ambos métodos son directos y simples de usar cuando se trata de *archivos de texto*, pero no son tan directos ni tan simples cuando se trata de leer o grabar en un *archivo binario*, sobre todo si la intención es trabajar con registros.

Por este motivo, para la lectura y grabación de datos en un *archivo binario* emplearemos una técnica diferente, mucho más simple, que en general se designa como *serialización*. La *serialización* es un proceso por el cual el contenido de una variable normalmente de estructura compleja (como puede ser un registro, un objeto, una lista, etc.) se convierte automáticamente en una secuencia de bytes listos para ser almacenados en un archivo, pero de tal forma que luego esa secuencia de bytes puede recuperarse desde el archivo y volver a crear con ella la estructura de datos original [2] [3]. El mecanismo de *serialización* no sólo está disponible en *Python* sino también en muchos otros lenguajes (sobre todo orientados a objetos, *Java* entre ellos).

El mecanismo de serialización en Python puede ser aplicado empleando distintos módulos y funciones según la necesidad del programador. En nuestro caso, emplearemos el módulo *pickle*, que entre otras funciones, provee las dos que necesitaremos: *dump()* y *load()*.

Si bien lo normal es que se use serialización para almacenar (o *preservar*, y de allí el curioso nombre de *pickle*) en un archivo variables de estructura compleja, no hay problema en usarla para variables simples (como variables de tipo entero, flotante o boolean, por ejemplo) [2]. El siguiente programa está incluido como *test01.py* en el proyecto *[F22] Archivos* que acompaña a esta Ficha, y muestra un ejemplo simple de grabación y lectura de números enteros y flotantes en un archivo:

```
import os.path
import pickle

__author__ = 'Cátedra de AED'


def test():
    print('Procediendo a grabar números en el archivo')
    m = open('prueba.num', 'wb')
    x, y = 2.345, 19
    pickle.dump(x, m)
    pickle.dump(y, m)
    m.close()

    m = open('prueba.num', 'rb')
    a = pickle.load(m)
    b = pickle.load(m)
    m.close()
    print('Datos recuperados desde el archivo:', a, ' - ', b)

    print('Hecho...')


if __name__ == '__main__':
    test()
```

El programa anterior es simple pero muy ilustrativo: el primer bloque abre el archivo *prueba.num* (en modo 'wb', de forma que si no existía lo crea, y si ya existía elimina su contenido) y lo deja abierto para permitir grabaciones. La variable para manejar el archivo es *m*, en la que se ha asignado el objeto retornado por *open()*. Luego, se usa dos veces la función *pickle.dump()* para grabar en el archivo los valores de las variables *x* e *y* (valiendo 2.345 y 19 respectivamente). Ambos números son grabados en el archivo representado por *m*, uno a continuación del otro, haciendo que el archivo aumente su tamaño en bytes (que inicialmente era cero). Al terminar estas dos grabaciones, el archivo se cierra con *m.close()*.

El hecho de cerrar el archivo al terminar de grabar, permite que ahora el mismo pueda ser reabierto pero *cambiando el modo de apertura*: como en este momento se pretende leer su contenido, se vuelve a abrir el mismo archivo pero ahora en modo 'rb', y a continuación se invoca dos veces a la función *pickle.load()* para leer el contenido del archivo: la primera invocación a *pickle.load()* traerá desde el archivo una copia del valor 2.345, y la segunda traerá una copia del 19. Ambos valores son almacenados en las variables *a* y *b* respectivamente. El archivo no pierde sus datos después de ser leído.

La función *pickle.dump()* toma dos parámetros: el primero es la variable cuyos datos se quieren grabar, y la segunda es la variable manejadora del archivo (creada previamente con *open()*) en el cual se desea grabar el valor de la variable que entra como primer parámetro. Y la función *pickle.load()* sólo toma un parámetro: la variable manejadora del archivo que se

quiere leer. Al terminar la lectura, *pickle.load()* reconstruye la estructura leída (sea simple o compleja) y la retorna [2].

Un ejemplo algo más complejo muestra la forma en que podemos hacer lo mismo, pero con un variables de tipo registro en lugar de variables simples (ver modelo *test02.py* en el proyecto *[F22] Archivos*):

```
import pickle

__author__ = 'Catedra de AED'


class Libro:
    def __init__(self, cod, tit, aut):
        self.isbn = cod
        self.titulo = tit
        self.autor = aut


def display(libro):
    print('ISBN:', libro.isbn, end='')
    print(' - Título:', libro.titulo, end='')
    print(' - Autor:', libro.autor)


def test():
    print('Prueba de grabación de varios registros...')
    lib1 = Libro(2134, 'Fundación', 'Isaac Asimov')
    lib2 = Libro(5587, 'Fundación e Imperio', 'Isaac Asimov')
    lib3 = Libro(3471, 'Segunda Fundación', 'Isaac Asimov')

    fd = 'libros.dat'
    m = open(fd, 'wb')
    pickle.dump(lib1, m)
    pickle.dump(lib2, m)
    pickle.dump(lib3, m)
    m.close()
    print('Se grabaron varios registros en el archivo', fd)

    m = open(fd, 'rb')
    lib1 = pickle.load(m)
    lib2 = pickle.load(m)
    lib3 = pickle.load(m)
    m.close()

    print('Se recuperaron estos registros desde el archivo', fd, ':')
    display(lib1)
    display(lib2)
    display(lib3)


if __name__ == '__main__':
    test()
```

El modelo que acabamos de mostrar define un tipo registro llamado *Libro*, y las típicas funciones *init()* para inicializar un *Libro*, y *display()* para mostrar un *Libro*. La función *test()* crea tres registros y procede luego a grabar esos registros en un archivo libros.dat, manejado a través de la variable *m*. Cada una de las invocaciones a *pickle.dump()* graba el registro que toma como parámetro en el archivo representado por *m*, y lo hace de forma que el

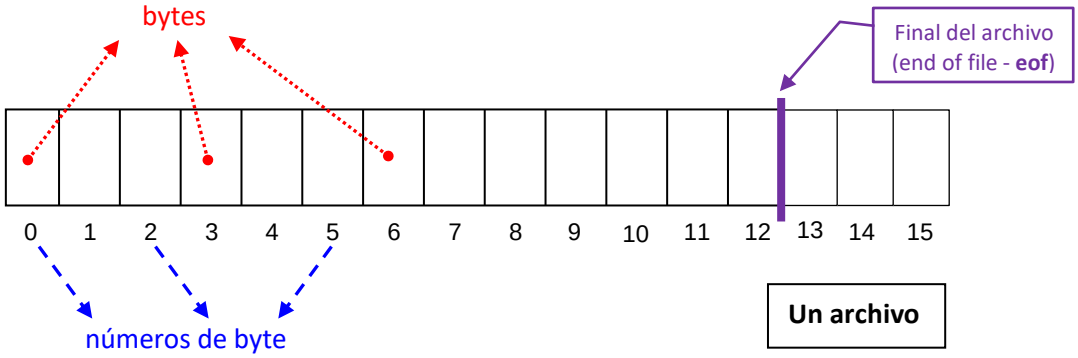
programador no deba preocuparse por los detalles de cómo se grabará cada campo. Los registros son grabados uno a continuación de otro, y luego el archivo se cierra.

La segunda parte de la función *test()* vuelve a abrir el archivo *libros.dat*, pero ahora en modo de sólo lectura, y procede a leer el contenido del mismo usando tres invocaciones a la función *pickle.load()*. En cada una de ellas, se recupera el contenido completo de uno de los registros grabados previamente, y cada registro recuperado se asigna nuevamente en las mismas variables iniciales. El archivo luego se cierra, y se muestran en pantalla los tres registros. Se puede observar que el mecanismo es exactamente el mismo al que se aplicó cuando se grabaron variables simples en el modelo *test01.py* anterior.

3.] Archivos binarios en Python: recorrido secuencial y acceso directo.

Esencialmente, y para completar nuestra aproximación conceptual de la sección anterior, un *archivo* (binario o de texto) puede entenderse como un gran *arreglo o vector de bytes* ubicado en memoria externa en lugar de estar alojado en memoria principal. Cada *componente* de ese gran *arreglo en memoria externa* es uno de los bytes grabados en el archivo, y cada byte tiene (a modo de índice) un número que lo identifica, correlativo desde el cero en adelante. El *byte cero* es el primer byte del archivo. Notar que entonces el número de cada byte es un *indicador de su posición relativa al inicio del archivo* [1]:

Figura 1: Esquema conceptual de un archivo como un arreglo de bytes en memoria externa.



El sistema operativo es el responsable de recordar en qué lugar termina un archivo. Ese punto se conoce como el *final del archivo* o *end of file* (abreviado comúnmente como *eof* por su significado en inglés). Es importante notar que los bytes que siguen al final del archivo (los bytes desde el 13 en adelante en la figura anterior) no pertenecen al archivo pero aún así están numerados en forma correlativa, continuando con la numeración que traía el archivo.

Y un detalle interesante que se deduce de lo anterior, es que debido al hecho de que la numeración de los bytes comienza en cero, entonces el *número total de bytes* (o *tamaño*) del archivo *coincide con el número del primer byte que se encuentra fuera del mismo* (en el gráfico anterior, el archivo tiene 13 bytes, y el byte 13 es el primero que está fuera del archivo: el tamaño del archivo es de 13 bytes).

En muchas situaciones el programador necesita obtener el *tamaño en bytes de un archivo*. Hay varias formas de hacerlo en Python, pero el siguiente ejemplo muestra la que quizás sea la más simple, usando la función *getsize()* incluida en el módulo *os.path* [2]:

```
import os.path
```

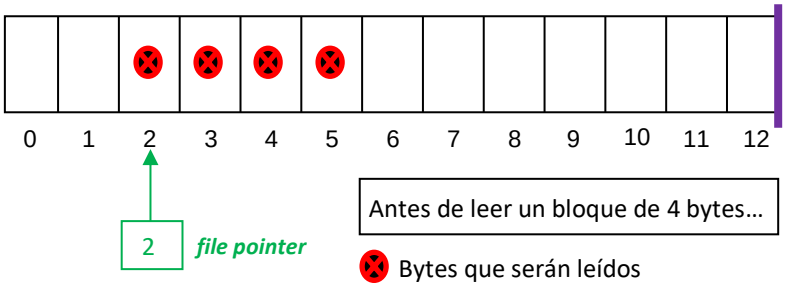
```
t = os.path.getsize('libros.dat')
print(t)
```

La función *os.path.getsize()* toma como parámetro el nombre físico del archivo (que aquí suponemos que se llama *'libros.dat'*, y simplemente retorna su tamaño en bytes. Note que no es necesario que el archivo esté abierto con *open()* para poder usar esta función.

En general, todo archivo cuenta con una especie de *cursor* o *indicador* o *marcador interno* llamado ***file pointer***, que no es otra cosa que una variable de tipo entero tal que, mientras el archivo está abierto, contiene el número del byte sobre el cual se realizará la próxima operación de lectura o de grabación. Tanto las operaciones de lectura como las de grabación, comienzan la operación en el byte indicado por el ***file pointer*** y al terminar la misma, dejan el ***file pointer*** apuntando al byte siguiente a aquel en el cual terminó la operación. Si una operación de salida graba bytes detrás del byte de finalización del archivo, entonces el tamaño del archivo crece y se ajusta para abarcar los nuevos bytes [1].

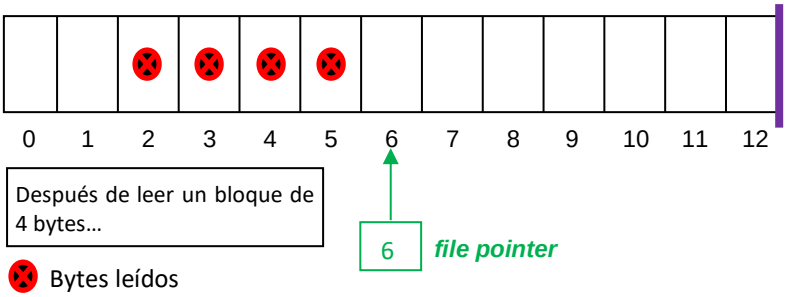
Por ejemplo, supongamos el mismo archivo de la *Figura 1*, con 13 bytes ya grabados. Supongamos también en un momento dado el ***file pointer*** está apuntando al byte 2 de este archivo:

Figura 2: Esquema del estado del *file pointer* en un archivo antes de proceder a una lectura.



Si en estas condiciones se lanza una operación de lectura para recuperar desde el archivo un bloque de cuatro bytes (por ejemplo, con la función *load()*), entonces dicha operación leerá desde el archivo 4 bytes comenzando desde el byte 2 (pues allí apunta ahora el ***file pointer***). Al terminar de leer, el ***file pointer*** quedará apuntando al byte 6 (pues la lectura llegó hasta el 5 inclusive):

Figura 3: Esquema del estado del *file pointer* en un archivo después de proceder a una lectura.



En general el valor del ***file pointer*** es gestionado automáticamente por las funciones y métodos básicos para gestión de archivos: *open()* abre el archivo y asigna el valor inicial al

file pointer (típicamente el valor cero), mientras que *dump()* y *load()* (si se aplica serialización a través de *pickle*) ajustan el valor del ***file pointer*** para dejarlo apuntando al byte en donde comenzará la próxima operación. Si el programador se limita a usar estos mecanismos grabando y leyendo datos estrictamente en el orden que surge de este manejo automático del file pointer, entonces *el archivo siempre será accedido y recorrido en forma estrictamente secuencial*, comenzando desde el primer byte y avanzado sobre el resto sin saltearse ninguno.

El acceso y recorrido en forma secuencial es común y simple de implementar. Muchos procesos requieren justamente tomar todos los datos de un archivo, uno por uno en el orden en que aparecen, y procesarlos en ese orden. Por ejemplo, esto será necesario si se desea hacer un listado de todo el contenido del archivo, o buscar un dato particular en ese archivo.

Pero en muchas otras ocasiones, el programador necesitará poder acceder en forma directa a ciertos datos o posiciones en el archivo, y en casos así sería deseable no tener que pasar en forma secuencial por los datos que estén grabados antes que los que se quiere acceder, para evitar la pérdida de tiempo. Y para lograr esto, el programador necesita poder controlar en *forma manual* el valor del ***file pointer***.

En ese sentido, los objetos tipo *file object* creados con *open()* contienen un método *tell()* que retorna el valor del *file pointer* en forma de un número entero, y también un método *seek()* que permite cambiar el valor del mismo (y por lo tanto, *seek()* permite *elegir cuál será el próximo byte que será leído o grabado*) [2] [3].

La función *tell()* es muy útil cuando por algún motivo se requiere saber en qué byte está posicionado un archivo en un momento dado. Esto es particularmente necesario cuando se quiere leer el contenido completo de un archivo en forma secuencial, usando un ciclo. Suponga que se tiene un archivo que contiene numerosos registros grabados mediante *pickle.dump()* y se necesita leer ese archivo y mostrar su contenido en pantalla. Está claro que el programador requerirá un ciclo, de forma de hacer una lectura (con *pickle.load()*) y una visualización en cada iteración. Pero si se desconoce el número de registros que contiene el archivo, ¿cómo hará el programador para controlar que no se haya llegado al final del archivo en una lectura? En general, si se intenta leer un bloque de bytes ubicado más allá del final del archivo, los métodos y funciones de lectura provocarán un error de runtime y el programa se interrumpirá.

Una forma de resolver el problema, consiste en chequear en cada vuelta del ciclo y antes de invocar a *pickle.load()* si el valor del ***file pointer*** es menor que el tamaño del archivo. Como vimos, el tamaño del archivo es igual al número del primer byte que se encuentra fuera del mismo y puede obtenerse con la función *os.path.getsize()*. Si el ***file pointer*** en un momento dado está apuntando a un byte cuyo número es menor que el tamaño, entonces el ***file pointer*** está posicionado en un byte que pertenece al archivo y la lectura sería en ese caso válida. Consideremos el siguiente programa a modo de ejemplo (incluido en el modelo *test03.py* del proyecto *[F22] Archivos*):

```
import pickle
import os.path

__author__ = 'Catedra de AED'
```

```
class Libro:
    def __init__(self, cod, tit, aut):
        self.isbn = cod
        self.titulo = tit
        self.autor = aut


def display(libro):
    print('ISBN:', libro.isbn, end='')
    print(' - Título:', libro.titulo, end='')
    print(' - Autor:', libro.autor)


def test():
    print('Prueba de grabación de varios registros...')
    lib1 = Libro(2134, 'Fundación', 'Isaac Asimov')
    lib2 = Libro(5587, 'Fundación e Imperio', 'Isaac Asimov')
    lib3 = Libro(3471, 'Segunda Fundación', 'Isaac Asimov')
    lib4 = Libro(1122, 'Los Límites de la Fundación', 'Isaac Asimov')
    lib5 = Libro(2286, 'Fundación y Tierra', 'Isaac Asimov')

    fd = 'libros.dat'
    m = open(fd, 'ab')
    pickle.dump(lib1, m)
    pickle.dump(lib2, m)
    pickle.dump(lib3, m)
    pickle.dump(lib4, m)
    pickle.dump(lib5, m)
    m.close()
    print('Se grabaron varios registros en el archivo', fd)

    m = open(fd, 'rb')
    t = os.path.getsize(fd)

    print('Se recuperaron estos registros desde el archivo', fd, ':')
    while m.tell() < t:
        lib = pickle.load(m)
        display(lib)
    m.close()

    t = os.path.getsize(fd)
    print('Tamaño del archivo al terminar:', t, 'bytes')


if __name__ == '__main__':
    test()
```

La función *test()* almacena varios registros de tipo *Libro* en el archivo *libros.dat* y luego lo cierra. Inmediatamente después se usa la función *os.path.getsize()* para obtener el tamaño en bytes del archivo (que se almacena en la variable *t*):

```
# fd = 'libros.dat'

m = open(fd, 'rb')
t = os.path.getsize(fd)
```

Luego, la función *test()* lanza un ciclo para leer en forma secuencial el archivo y mostrar su contenido:

```
print('Se recuperaron estos registros desde el archivo', fd, ':')
while m.tell() < t:
    lib = pickle.load(m)
    display(lib)
m.close()
```

En cada vuelta de ese ciclo, se usa el método *tell()* para controlar si el valor del file pointer del archivo gestionado con m es menor que el tamaño en bytes del archivo (almacenado en la variable *t*). Si eso es cierto, el ciclo ejecuta su bloque de acciones, invocando a *pickle.load()* para leer el siguiente registro, mostrándolo por pantalla inmediatamente después. Como cada vez que se hace una lectura (o una grabación) el valor del *file pointer* pasa automáticamente al inicio del siguiente bloque a leer, el resultado será que en algún momento el ciclo se detendrá por llegar al final del archivo (en cuyo caso el *file pointer* estará apuntando al primer byte exterior al archivo).

La técnica que hemos mostrado permite controlar con mucha sencillez el recorrido y lectura secuencial de un archivo, y será aplicada en numerosas oportunidades prácticas.

Por su parte, el método *seek(offset, from_what)* permite cambiar el valor del *file pointer*. En general recibe dos parámetros: el primero (que aquí llamamos *offset*) indica cuántos bytes debe moverse el *file pointer*, y el segundo (llamado aquí *from_what*, si está presente) indica desde donde se hace ese salto (un valor 0 indica saltar desde el principio del archivo, un 1 desde donde está el file pointer en este momento y un 2 indica saltar desde el final). El valor por default del segundo parámetro es 0, por lo cual por omisión se asume que los saltos son desde el inicio del archivo. El valor del segundo parámetro puede ser indicado como un número o por medio de una de las siguientes tres constantes predefinidas (las tres dentro del módulo io que debe ser importado en el programa para poder usarlas) [2]:

Tabla 2: Constantes predefinidas para el método seek().

Constante	Valor	Significado
io.SEEK_SET	0	Reposicionar comenzando desde el principio del archivo. El valor del primer parámetro (offset) puede ser 0 o positivo (pero no negativo).
io.SEEK_CUR	1	Reposicionar comenzando desde la posición actual del puntero de registro activo. El valor de offset puede ser entonces negativo, 0 o positivo.
io.SEEK_END	2	Reposicionar comenzando desde el final del archivo. El valor de offset típicamente es negativo, aunque puede ser 0 o positivo.

Si no se indica el segundo parámetro al invocar a *seek()*, se asume por defecto que su valor es 0, y en ese caso el reposicionamiento se hará desde el inicio del archivo.

En general, la forma de entender el funcionamiento de este método consiste en suponer que el *file pointer* está apuntando al lugar indicado por el segundo parámetro, y sumar a esa posición el valor del primer parámetro. Veamos algunos ejemplos. Siempre suponemos la instrucción *import io* está incluida en el programa, que el archivo está abierto, y que la variable manejadora del archivo se llama *m*:

a.) Sin importar donde esté ubicado el *file pointer* en este momento, suponga que queremos cambiar su valor para que pase a valer 10. Podemos hacerlo así:

```
m.seek(10, io.SEEK_SET)
```

Si el segundo parámetro es SEEK_SET (o sea, 0), se debe asumir que el file pointer está ubicado al inicio del archivo (o sea, su valor sería el 0), a ese valor sumarle el 10 que se pasó como primer parámetro, y asignar el resultado en el *file pointer* que pasará entonces a valer el número 10.

b.) Sin importar donde esté ubicado el *file pointer* en este momento, suponga que queremos cambiar su valor para que salte al final del archivo. Podemos hacerlo así:

```
m.seek(0, io.SEEK_END)
```

Si el segundo parámetro es SEEK_END (o sea, 2), se debe asumir que el file pointer está ubicado al final del archivo (o sea, su valor sería el número del primer byte que está fuera del archivo), a ese valor sumarle el 0 que se pasó como primer parámetro, y asignar el resultado al *file pointer* que pasará entonces a valer el número del primer byte que está fuera del archivo (es decir, el file *pointer* quedará apuntando al final).

c.) Suponga que el *file pointer* en este momento está apuntando al byte 7 del archivo (el valor del file pointer es 7), y suponga que queremos cambiar su valor para que pase a apuntar al byte 4. Podemos hacerlo así:

```
m.seek(-3, io.SEEK_CUR)
```

Si el segundo parámetro es SEEK_CUR (o sea, 1), se debe asumir que el file pointer está ubicado en su posición actual (o sea, su valor sería el byte 7 que ya tenía), a ese valor sumarle el -3 que se pasó como primer parámetro, y asignar el resultado en el file pointer. Como 7 + (-3) = 4, el *file pointer* pasará a valer 4.

d.) Está claro que el ejemplo anterior también podría haber sido resuelto así:

```
m.seek(4, io.SEEK_SET)
```

Si el segundo parámetro es SEEK_SET (o sea, 0), se debe asumir que el file pointer está ubicado al inicio del archivo (o sea, su valor sería el 0), a ese valor sumarle el 4 que se pasó como primer parámetro, y asignar el resultado en el file pointer, que pasará a valer 4.

Aunque el método *seek()* admite todas estas variantes, en la práctica lo más común es que se utilice siempre con el segundo parámetro valiendo io.SEEK_SET, que es claramente lo más cómodo: si se usa io.SEEK_SET, entonces el valor pasado como primer parámetro es directamente el valor que tomará el file pointer, y el programador no tiene que realizar en este caso ningún cálculo auxiliar.

No obstante lo anterior, ocurre con frecuencia que el programador necesite llevar el *file pointer* al final del archivo, y en ese caso el ejemplo *b.)* muestra la forma de hacerlo. Una aplicación de esto consiste en la siguiente forma alternativa de calcular el tamaño del archivo (sin usar la función *os.path.getsize()*):

```
def size(fd):
    file = open(fd, 'rb')
    file.seek(0, io.SEEK_END)
    t = file.tell()
    file.close()
    return t
```

La función anterior toma como parámetro una cadena de caracteres que representa el nombre físico del archivo cuyo tamaño se quiere medir. Luego abre el archivo con *open()* (en modo de sólo lectura es suficiente) y posiciona el file pointer el final del archivo con *seek()*. Al hacer esto, el file pointer queda valiendo el número del primer byte que está fuera del

archivo, que como ya sabemos, es también igual al tamaño del archivo. Por lo tanto, se invoca a *tell()* para obtener ese valor, y se lo retorna sin más.

El uso del método *seek()* permite cambiar el valor del *file pointer* a voluntad del programador, de forma que luego puede leer o grabar en cualquier posición del archivo sin tener que hacer un recorrido secuencial previo. Si el programador sabe o puede calcular en qué byte específico del archivo necesita leer o grabar, sólo debe invocar a *seek()*, cambiar el valor del *file pointer* al lugar requerido, y luego leer o grabar.

Si *m* es la variable para manejar un archivo, y ese archivo está abierto en un modo que permita tanto leer como grabar, la siguiente secuencia grabará el contenido de la variable *x* (sea *x* del tipo que sea…) a partir del byte cuyo número es *p* dentro del archivo:

```
m.seek(p, io.SEEK_SET)
pickle.dump(x, m)
```

Lo anterior es equivalente a:

```
m.seek(p)
pickle.dump(x, m)
```

ya que si el segundo parámetro en *seek()* se omite, se asume que vale justamente io.SEEK_SET.

No olvide que luego de grabar o leer, las funciones *dump()* y *load()* cambian automáticamente el valor del *file pointer* para que quede apuntando al byte siguiente a aquel en el cual terminó la operación. Por lo tanto, en el ejemplo anterior, el valor del file pointer al terminar la grabación no será el mismo que se le asignó con *seek()*.

Y si lo que buscaba era leer en lugar de grabar, la siguiente secuencia leerá los bytes que correspondan desde el archivo gestionado por *m*, comenzando desde la posición *p*, y retornará la estructura reconstruida asignándola en *x*:

```
# m.seek(p)
m.seek(p, io.SEEK_SET)
x = pickle.load(m)
```

Cuando se trabaja de esta forma se dice que el programador está realizando *acceso directo* (o también *acceso aleatorio*) al contenido del archivo, en lugar del *acceso secuencial* que antes hemos analizado. Ambas técnicas pueden tranquilamente coexistir en un mismo programa, del mismo modo que el procesamiento de un arreglo puede requerir recorridos secuenciales y accesos directos a sus componentes en la misma aplicación.

4.] Procesamiento combinado de registros, arreglos y archivos.

Se propone ahora un caso de aplicación en el que se pedirá combinar el uso de distintas estructuras de datos, tales como arreglos, registros y archivos. Cada una de las situaciones será analizada y resuelta, y en el análisis que corresponda a cada una de ellas se explicarán algunas técnicas elementales de procesamiento combinado.

Problema 53.) *Desarrollar un programa controlado por menú de opciones, que permita gestionar un arreglo de registros (puede suponer el mismo tipo Libro que hemos usado como ejemplo en esta misma Ficha), y a partir de las opciones del menú proceda a generar un*

archivo con los datos del arreglo (o viceversa: volver a crear el arreglo a partir de los datos del archivo). Las opciones que debería incluir son las siguientes:

a.) *Crear y cargar un arreglo v de n registros de tipo Libro (puede hacer esta carga en forma automática).*

b.) *Mostrar el arreglo.*

c.) *Crear un archivo libros.dat que contenga todos los registros del arreglo original. Si el archivo ya existía, eliminar su contenido y comenzar desde cero.*

d.) *Mostrar el contenido del archivo libros.dat.*

e.) *Crear nuevamente el archivo libros.dat, de forma que ahora contenga sólo los datos de los libros cuyo código sea menor que x, cargando el código x por teclado. Si el archivo ya existía, eliminar su contenido y comenzar desde cero.*

f.) *A partir del archivo libros.dat, volver a crear en memoria el arreglo v, de forma que contenga todos los registros del archivo. Reemplace el arreglo creado en el punto a.) por el que se le pide crear ahora.*

g.) *A partir del archivo libros.dat, volver a crear en memoria el arreglo v, que contenga sólo los registros de los libros cuyo código sea mayor a x (cargue x por teclado). Reemplace el arreglo creado en el punto a.) por el que se le pide crear ahora.*

Discusión y solución: El proyecto *[F22] Archivos* que acompaña a esta Ficha contiene un modelo *test04.py* con el programa completo que resuelve este caso de análisis.

La declaración del registro *Libro* es directa y similar a los hecho en casos anteriores. Se importan además algunos módulos que serán necesarios en distintos puntos del desarrollo del programa:

```
import random
import pickle
import os
import os.path
import io

__author__ = 'Catedra de AED'


class Libro:
    def __init__(self, cod, tit, aut):
        self.isbn = cod
        self.titulo = tit
        self.autor = aut


def display(libro):
    print('ISBN:', libro.isbn, end='')
    print(' - Título:', libro.titulo, end='')
    print(' - Autor:', libro.autor)
```

Como es costumbre, la función *test()* gestiona el menú principal, y en ella se crea el vector *v* inicialmente vacío, además de una variable *fd* que contiene el nombre del archivo con el que se trabajará:

```
def test():
    v = []
```

```
    fd = 'libros.dat'
    op = -1
    while op != 8:
        print('Procesamiento combinado de arreglos, registros y archivos...')
        print('1. Crear el arreglo de libros (en forma automática)')
        print('2. Mostrar el arreglo de libros')
        print('3. Crear el archivo con TODOS los libros del arreglo')
        print('4. Mostrar el archivo de libros')
        print('5. Crear el archivo con ALGUNOS de los libros del arreglo')
        print('6. Volver a crear el arreglo con TODOS los libros del archivo')
        print('7. Volver a crear el arreglo con ALGUNOS de los libros del archivo')
        print('8. Salir')
        op = int(input('\t\tIngrese número de opción: '))
        print()

        if op == 1:
            cargar_arreglo(v)

        elif op == 2:
            mostrar_arreglo(v)

        elif op == 3:
            crear_archivo_todos(v, fd)

        elif op == 4:
            mostrar_archivo(fd)

        elif op == 5:
            crear_archivo_algunos(v, fd)

        elif op == 6:
            v = crear_arreglo_todos(fd)

        elif op == 7:
            v = crear_arreglo_algunos(fd)

        elif op == 8:
            pass


if __name__ == '__main__':
    test()
```

De aquí en más, siguen las funciones invocadas desde el menú. Las dos primeras son las encargadas de crear y cargar el arreglo en forma automática (evitando la carga por teclado, para ganar tiempo en este programa de prueba) y de mostrar el contenido del arreglo:

```
def cargar_arreglo(v):
    n = int(input('Cuantos libros desea cargar?: '))
    for i in range(n):
        cod = random.randint(1, 10000)
        tit = 'Título ' + str(i)
        aut = 'Autor ' + str(i)
        lib = Libro(cod, tit, aut)
        v.append(lib)


def mostrar_arreglo(v):
    if len(v) == 0:
        print('No hay datos en el arreglo...')
        print()
        return

    print('Los libros registrados son:')
    for libro in v:
```

```
                display(libro)

            print()
```

La función *crear_archivo_todos()* crea el archivo pedido, almacenando en él todos los registros contenidos en el arreglo *v*:

```
def crear_archivo_todos(v, fd):
    if len(v) == 0:
        print('No hay datos en el arreglo...')
        print()
        return

    m = open(fd, 'wb')

    # forma 1: si se desea que el archivo contenga un vector...
    # pickle.dump(v, m)

    # forma 2: si se desea que al archivo contenga los registros
    # almacenados uno a uno en forma secuencial...
    for lib in v:
        pickle.dump(lib, m)

    m.close()
    print('Se creó el archivo', fd, 'con todos los registros del vector')
    print()
```

Esta función comienza chequeando si el arreglo *v* tomado como parámetro está vacío, en cuyo caso retorna al menú sin hacer nada. Luego abre el archivo en modo 'wb', y procede a grabar el contenido del arreglo *v*. Aquí tenemos dos posibles técnicas (que en el código fuente están incluidas y marcadas con comentarios): si el programador necesita que el archivo contenga directamente grabado el arreglo de tal forma que pueda ser directamente recuperado luego, entonces puede aplicar la *forma 1*, que consiste en usar *pickle.dump()* y pasarle todo el arreglo como parámetro:

```
# forma 1
m = open(fd, 'wb')

# forma 1: si se desea que el archivo contenga un vector...
pickle.dump(v, m)

m.close()
```

El mecanismo de serialización está definido de manera que es posible no sólo grabar y recuperar variables simples y registros: en rigor, la serialización es una técnica general para preservar el contenido de cualquier tipo de objeto, sin importar su complejidad interna. Las variables de tipo list, así como las tuplas, las cadenas, y los rangos, son objetos… y pueden ser serializados en forma directa, con una sola invocación a *pickle.dump()*, o recuperados (des-serializados) con una sola invocación a *pickle.load()*.

La forma anterior de guardar todo el arreglo en un archivo es muy útil cuando el programador está seguro de que luego volverá a recuperar el arreglo entero, y que no necesitará procesar uno por uno los registros que se grabaron en el archivo.

Si el programador necesitase almacenar uno por uno los registros en el archivo (por ejemplo, si no está seguro de volver a necesitarlos a todos en un arreglo más adelante, o si sospecha que necesitará procesar esos registros directamente desde el archivo, sin subirlos a memoria

en un arreglo) entonces puede aplicarse la *forma 2*, que consiste en usar un ciclo para recorrer el arreglo y grabar uno por uno todos los registros:

```
# forma 2
for lib in v:
    pickle.dump(lib, m)
```

que sería exactamente lo mismo que:

```
# forma 2
for i in range len(v):
    pickle.dump(v[i], m)
```

Ambas técnicas son simples y directas. La aplicación de una u otra dependerá del contexto del problema y de las necesidades del programador. En cada una de las funciones que quedan en el programa, hemos incluido ambas formas de trabajo: una de ellas, la forma 1, está en el código fuente pero en forma de comentarios. La otra, la *forma 2*, es la que efectivamente está activa en todo el programa. Si el estudiante quisiera activar la *forma 1* y desactivar la *foma 2*, sólo tiene que debe "apagar" con comentarios la *forma 2* y eliminar los comentarios de la *forma 1*, pero recuerde: debe hacerlo en todo el programa: sólo una de ambas formas debería estar activa, si quiere evitar luego problemas cuando quiera leer el archivo y recuperar con *pickle.load()* los objetos grabados.

La función *crear_archivo_algunos()* crea también el archivo pedido, pero almacenando en él sólo los registros contenidos en el arreglo *v* que tengan isbn o código menor que el valor *x* que se carga por teclado:

```
def crear_archivo_algunos(v, fd):
    if len(v) == 0:
        print('No hay datos en el arreglo...')
        print()
        return

    x = int(input('Ingrese el código a comparar: '))
    m = open(fd, 'wb')

    # forma 1: si se desea que el archivo contenga un vector...
    # v2 = []
    # for lib in v:
    #     if lib.isbn < x:
    #         v2.append(lib)
    # pickle.dump(v2, m)

    # forma 2: si se desea que al archivo contenga los registros
    # almacenados uno a uno en forma secuencial..
    for lib in v:
        if lib.isbn < x:
            pickle.dump(lib, m)

    m.close()
    print('Se creó el archivo', fd, 'con los registros con código <', x)
    print()
```

La *forma 1* consiste en crear un vector *v2* que contenga referencias a los libros cuyo código sea menor a *x*, y luego grabar ese nuevo arreglo en el archivo (que de esta forma, también contendrá un arreglo en lugar de registros grabados uno por uno) Recuerde que esta

alternativa está incluida entre comentarios en el programa, y por lo tanto no es la que efectivamente se está aplicando:

```
# forma 1
v2 = []
for lib in v:
   if lib.isbn < x:
       v2.append(lib)
pickle.dump(v2, m)
```

La *forma 2*, consiste en grabar uno por uno los registros en el archivo, directamente, sin almacenarlos primero en un arreglo auxiliar (esta es la forma que está activa en el programa):

```
# forma 2
for lib in v:
    if lib.isbn < x:
        pickle.dump(lib, m)
```

que también equivale a:

```
# forma 2
for i in range(len(v)):
    if v[i].isbn < x:
        pickle.dump(v[i], m)
```

Cuando el archivo ha sido creado (con todos los registros del arreglo o con algunos), la función *mostrar_archivo()* es la encargada de mostrar el contenido del archivo en pantalla:

```
def mostrar_archivo(fd):
    if not os.path.exists(fd):
        print('El archivo', fd, 'no existe...')
        print()
        return

    print('Contenido actual del achivo', fd, ':')
    m = open(fd, 'rb')

    # forma 1: si el archivo contenía un vector...
    # v = pickle.load(m)
    # for lib in v:
    #     display(lib)

    # forma 2: si el archivo contenía registros almacenados
    # uno a uno en forma secuencial...
    t = os.path.getsize(fd)
    while m.tell() < t:
        lib = pickle.load(m)
        display(lib)

    m.close()
    print()
```

Lo primero que hace la función es verificar si el archivo que se quiere mostrar realmente existe. Si no fuese el caso, la función informa con un mensaje y retorna al menú sin hacer nada más. La forma de comprobar si un archivo existe o no, consiste en usar la función *os.path.exists()*, la cual toma como parámetro el nombre físico del archivo a verificar, y retorna *True* si ese archivo existe, o *False* en caso contrario:

```
    if not os.path.exists(fd):
        print('El archivo', fd, 'no existe...')
        print()
        return
```

Si todo está bien, la función *mostrar_archivo()* abre el archivo en modo de sólo lectura y lo que sigue depende de cómo haya sido grabado el archivo: si se aplicó la *forma 1* y el archivo contiene directamente el vector de registros serializado, entonces sólo hay que recuperar ese arreglo con una sola invocación a *pickle.load()* y mostrar ese arreglo (esta técnica está incluida con comentarios en el código fuente):

```
    # forma 1: si el archivo contenía un vector...
    v = pickle.load(m)
    for lib in v:
        display(lib)
```

O bien:

```
    # forma 1: si el archivo contenía un vector...
    v = pickle.load(m)
    for i in range(len(v)):
        display(v[i])
```

Pero si al crear el archivo se aplicó la forma 2, entonces contiene los registros grabados uno por uno, y deben ser recuperados uno por uno (en este caso, se recuperan y se muestran, sin almacenarlos en un arreglo):

```
    # forma 2: si el archivo contenía registros almacenados
    # uno a uno en forma secuencial...
    t = os.path.getsize(fd)
    while m.tell() < t:
        lib = pickle.load(m)
        display(lib)
```

Sólo nos quedan las dos funciones que vuelven a crear el vector, a partir de los datos del archivo. La primera es la función *crear_arreglo_todos()*, que debe recuperar todo el contenido del archivo y volver a crear un arreglo con ellos:

```
def crear_arreglo_todos(fd):
    if not os.path.exists(fd):
        print('El archivo', fd, 'no existe...')
        print()
        return

    m = open(fd, 'rb')

    # forma 1: si el archivo contenía un vector...
    # v = pickle.load(m)

    # forma 2: si el archivo contenía los registros almacenados
    # uno a uno en forma secuencial...
    v = []
    t = os.path.getsize(fd)
    while m.tell() < t:
        lib = pickle.load(m)
        v.append(lib)

    m.close()
    print('Se creó el vector con todo el contenido del archivo', fd)
```

```
    print()

    return v
```

Si el archivo se creó con la *forma 1* y contiene entonces el arreglo grabado en forma directa, entonces la tarea es simple: sólo hay que invocar a *pickle.load()*, recuperarlo y retornarlo sin más antes de terminar la ejecución de la función (esta forma está desactivada con comentarios en el programa):

```
# forma 1: si el archivo contenía un vector...
v = pickle.load(m)
```

Pero si el archivo fue generado con la *forma 2*, grabando uno por uno los registros, entonces se debe leer el archivo registro por registro, con un ciclo, y almacenar en el arreglo los registros a medida que se leen: (este es el proceso que está activo en el programa):

```
# forma 2: si el archivo contenía los registros almacenados
# uno a uno en forma secuencial...
v = []
t = os.path.getsize(fd)
while m.tell() < t:
    lib = pickle.load(m)
    v.append(lib)
```

Finalmente, la función *crear_arreglo_algunos()* debe volver a crear el arreglo *v*, pero recuperando sólo los registros de los libros cuyo código sea mayor a *x* (cargando *x* por teclado):

```
def crear_arreglo_algunos(fd):
    if not os.path.exists(fd):
        print('El archivo', fd, 'no existe...')
        print()
        return

    x = int(input('Ingrese el código a comparar: '))
    m = open(fd, 'rb')

    # forma 1: si el archivo contenía un vector...
    # v = []
    # v2 = pickle.load(m)
    # for lib in v2:
    #     if lib.isbn > x:
    #         v.append(lib)

    # forma 2: si el archivo contenía los registros almacenados
    # uno a uno en forma secuencial...
    v = []
    t = os.path.getsize(fd)
    while m.tell() < t:
        lib = pickle.load(m)
        if lib.isbn > x:
            v.append(lib)

    m.close()
    print('Se creó el vector con parte del archivo', fd)
    print()

    return v
```

Y otra vez, si el archivo fue creado con la *forma 1* y contiene un arreglo completo grabado en forma directa, entonces debe recuperarse ese arreglo y a partir de él crear un segundo arreglo que contenga los registros pedidos, retornando el segundo arreglo al finalizar (esta forma está desactivada con comentarios en el código fuente):

```
# forma 1: si el archivo contenía un vector...
v = []
v2 = pickle.load(m)
for lib in v2:
   if lib.isbn > x:
       v.append(lib)
```

Y si el archivo fue creado con la *forma 2*, registro por registro, entonces esos registros deben ser leídos uno por uno de forma que se copien en el arreglo *v* sólo los que tengan código mayor a *x* (en este caso no es necesario un segundo arreglo):

```
# forma 2: si el archivo contenía los registros almacenados
# uno a uno en forma secuencial...
v = []
t = os.path.getsize(fd)
while m.tell() < t:
    lib = pickle.load(m)
    if lib.isbn > x:
        v.append(lib)
```

El programa completo se muestra a continuación:

```
import random
import pickle
import os
import os.path
import io

__author__ = 'Catedra de AED'


class Libro:
    def __init__(self, cod, tit, aut):
        self.isbn = cod
        self.titulo = tit
        self.autor = aut


def display(libro):
    print('ISBN:', libro.isbn, end='')
    print(' - Título:', libro.titulo, end='')
    print(' - Autor:', libro.autor)


def cargar_arreglo(v):
    n = int(input('Cuantos libros desea cargar?: '))
    for i in range(n):
        cod = random.randint(1, 10000)
        tit = 'Título ' + str(i)
        aut = 'Autor ' + str(i)
        lib = Libro(cod, tit, aut)
        v.append(lib)


def mostrar_arreglo(v):
    if len(v) == 0:
        print('No hay datos en el arreglo...')
        print()
```

```
        return

    print('Los libros registrados son:')
    for libro in v:
        display(libro)

    print()


def crear_archivo_todos(v, fd):
    if len(v) == 0:
        print('No hay datos en el arreglo...')
        print()
        return

    m = open(fd, 'wb')

    # forma 1: si se desea que el archivo contenga un vector...
    # pickle.dump(v, m)

    # forma 2: si se desea que al archivo contenga los registros
    # almacenados uno a uno en forma secuencial..
    for lib in v:
        pickle.dump(lib, m)

    m.close()
    print('Se creó el archivo', fd, 'con todos los registros del vector')
    print()


def mostrar_archivo(fd):
    if not os.path.exists(fd):
        print('El archivo', fd, 'no existe...')
        print()
        return

    print('Contenido actual del achivo', fd, ':')
    m = open(fd, 'rb')

    # forma 1: si el archivo contenía un vector...
    # v = pickle.load(m)
    # for lib in v:
    #    display(lib)

    # forma 2: si el archivo contenía registros almacenados
    # uno a uno en forma secuencial...
    t = os.path.getsize(fd)
    while m.tell() < t:
        lib = pickle.load(m)
        display(lib)

    m.close()
    print()


def crear_archivo_algunos(v, fd):
    if len(v) == 0:
        print('No hay datos en el arreglo...')
        print()
        return

    x = int(input('Ingrese el código a comparar: '))
    m = open(fd, 'wb')

    # forma 1: si se desea que el archivo contenga un vector...
    # v2 = []
    # for lib in v:
    #    if lib.isbn < x:
```

```
    #         v2.append(lib)
    # pickle.dump(v2, m)

    # forma 2: si se desea que al archivo contenga los registros
    # almacenados uno a uno en forma secuencial..
    for lib in v:
        if lib.isbn < x:
            pickle.dump(lib, m)

    m.close()
    print('Se creó el archivo', fd, 'con los registros con código <', x)
    print()


def crear_arreglo_todos(fd):
    if not os.path.exists(fd):
        print('El archivo', fd, 'no existe...')
        print()
        return

    m = open(fd, 'rb')

    # forma 1: si el archivo contenía un vector...
    # v = pickle.load(m)

    # forma 2: si el archivo contenía los registros almacenados
    # uno a uno en forma secuencial...
    v = []
    t = os.path.getsize(fd)
    while m.tell() < t:
        lib = pickle.load(m)
        v.append(lib)

    m.close()
    print('Se creó el vector con todo el contenido del archivo', fd)
    print()

    return v


def crear_arreglo_algunos(fd):
    if not os.path.exists(fd):
        print('El archivo', fd, 'no existe...')
        print()
        return

    x = int(input('Ingrese el código a comparar: '))
    m = open(fd, 'rb')

    # forma 1: si el archivo contenía un vector...
    # v = []
    # v2 = pickle.load(m)
    # for lib in v2:
    #    if lib.isbn > x:
    #        v.append(lib)

    # forma 2: si el archivo contenía los registros almacenados
    # uno a uno en forma secuencial...
    v = []
    t = os.path.getsize(fd)
    while m.tell() < t:
        lib = pickle.load(m)
        if lib.isbn > x:
            v.append(lib)

    m.close()
    print('Se creó el vector con parte del archivo', fd)
    print()
```

```
    return v


def test():
    v = []
    fd = 'libros.dat'
    op = -1
    while op != 8:
        print('Procesamiento combinado de arreglos, registros y archivos...')
        print('1. Crear el arreglo de libros (en forma automática)')
        print('2. Mostrar el arreglo de libros')
        print('3. Crear el archivo con TODOS los libros del arreglo')
        print('4. Mostrar el archivo de libros')
        print('5. Crear el archivo con ALGUNOS de los libros del arreglo')
        print('6. Volver a crear el arreglo con TODOS los libros del archivo')
        print('7. Volver a crear el arreglo con ALGUNOS de los libros del archivo')
        print('8. Salir')
        op = int(input('\t\tIngrese número de opción: '))
        print()

        if op == 1:
            cargar_arreglo(v)

        elif op == 2:
            mostrar_arreglo(v)

        elif op == 3:
            crear_archivo_todos(v, fd)

        elif op == 4:
            mostrar_archivo(fd)

        elif op == 5:
            crear_archivo_algunos(v, fd)

        elif op == 6:
            v = crear_arreglo_todos(fd)

        elif op == 7:
            v = crear_arreglo_algunos(fd)

        elif op == 8:
            pass


if __name__ == '__main__':
    test()
```

Bibliografía

[1] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.

[2] Python Software Foundation, "Python Documentation," 2021. [Online]. Available: https://docs.python.org/3/.

[3] M. Pilgrim, "Dive Into Python - Python from novice to pro", Nueva York: Apress, 2004.