Ficha 26

Archivos: Gestión ABM

1.] Introducción.

Hemos indicado que un archivo puede contener datos del cualquier tipo que necesite el programador, pero que será muy común que se requiera almacenar *registros* de algún tipo, de forma que luego se procesen esos registros en base a alguna estrategia. Hemos visto que muchas veces se puede copiar algunos o todos los registros del archivo a un arreglo en memoria y trabajar luego con el arreglo (que será la forma que más comúnmente emplearemos en los ejercicios y problemas que quedan).

Pero en la práctica real lo más común es que dado un archivo de registros el programador necesite operar *directamente con el archivo*, sin pasar sus registros a un arreglo. En ese sentido, son básicamente tres las operaciones que permanentemente suelen aplicarse sobre su contenido: la operación de *agregar un registro* al archivo (operación que suele designarse como *alta de un registro*, o simplemente una *alta*), la operación de *eliminar un registro* (que se conoce como *baja de un registro* o simplemente una *baja*), y la operación de *modificar el contenido de un registro que ya existía* (operación que se conoce como *modificación de un registro*) [1].

En los programas o modelos de programas que hasta aquí hemos desarrollado para mostrar el manejo de archivos, hemos visto esquematizada de una forma u otra la operación de alta de un registro. Sin embargo, es común que en un sistema de gran envergadura y complejidad se incluyan subsistemas completamente dedicados a realizar el *soporte de mantenimiento* de cada archivo que el sistema usa. Cuando hablamos de un *subsistema de soporte de mantenimiento de un archivo*, nos referimos a un programa (o a una librería o a un módulo de un programa) que sea capaz de realizar las tres operaciones básicas que hemos citado sobre un archivo (*altas*, *bajas* y *modificaciones*), más alguna otra operación adicional que tenga utilidad, como por ejemplo la visualización por pantalla del contenido del archivo (en forma completa o en forma parcial), o la búsqueda de un registro particular, entre otras[1]. Un programa de esta clase suele designarse como ***programa ABM***, siendo *ABM*

[1] El mantenimiento de un archivo o de una base de datos para un sistema informático crítico no es un tema para tomar a la ligera. La pérdida de esos datos, la modificación negligente de los mismos y/o la intervención malintencionada en los mecanismos de control del sistema puede tener consecuencias muy dañinas, como se reflejó en la película del 2007 *Live Free or Die Hard* (conocida aquí como *Duro de Matar 4.0*), dirigida por *Len Wiseman* y protagonizada por *Bruce Willis*. La película muestra el accionar de una banda de hackers ultra-tecnológicos contra todo el sistema informático de soporte de servicios públicos y financieros de los Estados Unidos, por medio de un ataque informático masivo designado como *fire sale* (o *venta de liquidación*). Un policía "chapado a la antigua" y un hacker "obligado a hacer de bueno" intentan desactivar el ataque, y la película estalla (literalmente…) en un impresionante "crescendo" de escenas de acción. Y todo por un "hacker malo"…

una abreviatura formada por las primeras letras de las palabras ***a**ltas*, ***b**ajas* y ***m**odificaciones*.

El hecho es que a la hora de desarrollar un ***programa ABM*** surgen algunos inconvenientes conceptuales que conviene estudiar en detalle para no cometer errores involuntarios y para no caer en pérdidas de eficiencia en los planteos que se realicen. A continuación, hacemos un breve análisis de cuales son esos problemas y las formas de resolverlos. En toda la explicación que sigue, supondremos que nuestro objetivo es diseñar un ***programa ABM*** para un archivo que contenga registros con datos de *estudiantes* de una carrera universitaria, con campos para el número de *legajo*, el *nombre* y el *promedio* de cada estudiante. Los detalles de código fuente irán apareciendo a medida que se vayan dejando en claro algunos aspectos prácticos.

2.] Bajas en un archivo de registros.

Básicamente, los problemas comienzan cuando se quiere plantear la forma de hacer una *baja de un registro*. La idea es que el programa debe pedir que se cargue por teclado el valor de la *clave de identificación del registro*[2] que se quiere borrar (en nuestro caso, si suponemos que en el archivo almacenaremos registros con datos de estudiantes, podemos tomar como clave al campo o atributo *legajo*). Luego, el programa debe buscar un registro con esa clave en el archivo, y en caso de encontrarlo, proceder a eliminar el registro completo.

Hasta aquí, todo parece fácil... pero el hecho es que el registro está grabado en el archivo, y no existe una instrucción en Python que permita eliminar un registro del archivo en forma similar a lo que hace la instrucción *del* para eliminar un componente de un arreglo representado en una variable de tipo *list*. Frente a esto, hay dos vías de solución clásicas [1]:

a.) La primera es usar un *segundo archivo* (designado como *archivo temporal*), inicialmente vacío, y proceder de la forma siguiente: El archivo original se abre en modo de lectura ('rb'), y el archivo temporal en modo de grabación pero de forma que si el archivo existía se elimine su contenido ('wb'). Con un ciclo *se recorre y se lee (con pickle.load()) hasta el final el archivo original*. En cada vuelta del ciclo se toma el registro leído desde archivo original y se verifica si el campo clave (el legajo en nuestro caso) coincide con el valor buscado. Si el valor *no* coincide, entonces simplemente se toma ese registro y se lo graba (con *pickle.dump()*) en el archivo *temporal*. El único registro que *no se copia* al temporal, es aquel cuyo campo clave coincida con el valor buscado.

Al finalizar el ciclo, el archivo temporal será una copia exacta del archivo *original*, salvo que no contendrá al registro que se quería borrar. Entonces lo único que queda por hacer es *destruir el archivo original* (borrarlo del *sistema de archivos* o *file system* del disco), y cambiar el nombre del archivo temporal para que ahora tome el nombre que tenía el original.

Este proceso completo efectivamente permite eliminar de un archivo a un registro, aunque para ello trabaja transitoriamente con un archivo temporal que se usa a modo de réplica del original.

[2] En general, se entiende como *clave* de un registro al campo (o conjunto de campos) cuyos valores permiten identificar en *forma única* a un registro en el archivo. El número de legajo suele ser una clave válida en la Universidad, puesto que para cada alumno el número de legajo es único. Normalmente, son buenos candidatos para claves de registro los campos cuyos valores no pueden (o no deben...) repetirse, pero en algunas ocasiones se usan claves que admitan valores repetidos.

En general, cualquier operación que permita eliminar un registro de un archivo de forma que el registro no ocupe más lugar dentro del archivo, se conoce como una ***baja física*** de registro.

El proceso de *baja física* hace que el registro realmente sea eliminado, disminuyendo el tamaño del archivo original, pero como contrapartida puede verse claramente que el mecanismo completo insume mucho tiempo, ya que todos los registros del archivo original deben ser leídos y grabados uno por uno en el archivo temporal (salvo el que se quiere eliminar). En archivos pequeños este tiempo puede ser despreciable, pero en situaciones reales con archivos muy grandes, la demora será importante.

b.) La otra forma de hacer una baja consiste en no tratar de eliminar el registro físicamente, pues como se indicó esto lleva mucho tiempo. Lo que se puede hacer es una variante conocida como ***baja lógica*** (o ***marcado lógico***) del registro: en vez de eliminarlo físicamente, *simplemente se lo marca* usando un campo o atributo especialmente definido a modo de *bandera*: si este campo vale *False*, significa que el registro está *marcado como eliminado* y por lo tanto no debe tenerse en cuenta en ninguna operación que se realice sobre el archivo, pero si vale *True*, *entonces el registro está activo* y por lo tanto debe ser procesado cuando se recorra el archivo.

Cuando un registro nuevo se graba en el archivo, se hace de forma que ese atributo de marcado lógico valga *True*, y sólo se cambia ese valor cuando se desea eliminar el registro. De esta forma, el proceso de *baja lógica* consiste en buscar el registro con la clave dada usando un ciclo, leerlo con *pickle.load()*, cambiar el atributo de marcado lógico a *False*, y *volver a grabar el registro en el mismo lugar original* usando *pickle.dump()*.

Ahora bien: si se aplica siempre el proceso de baja lógica, puede ocurrir que en algún momento el archivo quede *disperso*: por más que los registros se marcan como borrados, *siguen ocupando lugar físico* dentro del archivo. Con el tiempo, los registros marcados pero no borrados implican un tiempo de procesamiento extra que equilibra el tiempo que se pudiera haber ganado con las bajas lógicas en lugar de las físicas, además del uso innecesario del espacio de disco para registros que no se usan.

En la práctica, y para plantear una solución de equilibrio entre las ventajas de ambas estrategias, se suele aplicar un *esquema combinado*: cuando se requiere hacer bajas en el momento (por ejemplo, en casos de atención frente al público en tiempo real), se aplica un proceso de *baja lógica* para no perder tiempo. Y luego, en algún momento no crítico del ciclo de uso del archivo (por ejemplo, a la hora del cierre o de la apertura de la organización que usa el sistema, si es que en algún momento el mismo se cierra), el encargado del mantenimiento del sistema efectúa ***un*** proceso de *bajas físicas*, pero eliminando de una sola pasada todos los registros que quedaron marcados. Este último proceso suele llamarse *depuración* del archivo (o también *compactación* u *optimización del espacio físico* del archivo).

Cabe aclarar que el proceso de *baja lógica*, planteado en la forma que aquí se hizo, requiere recorrer el archivo hasta encontrar el registro buscado. En el peor caso, si el registro se encuentra muy al final, eso supondría que el archivo debería ser leído casi completamente, y por ello se perdería básicamente el mismo tiempo que con una baja física (en la *baja física* se agrega el tiempo extra de grabar el registro en el archivo secundario). Si el proceso de *baja lógica* se hace en base a un recorrido secuencial del archivo, entonces en promedio no hay diferencia significativa de tiempo con respecto a la *baja física*.

Entonces, ¿por qué insistir? El hecho es que la técnica para realizar *bajas lógicas* realmente permite ganar tiempo si el archivo está organizado de forma que permita accesos rápidos a sus registros individuales. Esto puede lograrse con métodos de organización muy conocidos, pero que escapan al alcance de este curso, como la *organización de claves para acceso directo*, la *indexación*, o la *búsqueda por dispersión* (llamada más comúnmente *hashing*). En todo caso, lo que aquí se pretende es mostrar al estudiante la forma en que puede hacerse una *baja lógica*, y el mismo estudiante podrá oportunamente adaptar la técnica para situaciones más provechosas [1].

El programa *abm.py* que viene incluido en el proyecto *[F26] Gestión ABM* anexo a esta Ficha, aplica estas ideas sobre un archivo con registros de estudiantes. El planteo y análisis detallado del programa y las tecnicas que se aplican en él, se verá en las secciones que siguen.

3.] Altas en un archivo de registros.

Una vez resuelto (o al menos, diseñado) el tema de las *bajas* (usando un campo de marcado lógico y/o combinando con un proceso de eliminación física posterior), entonces el proceso de *alta* resulta simple. Si se asume que en el archivo no se permitirán registros con claves repetidas (lo cual es muy común), entonces los pasos a seguir son los siguientes: se carga por teclado el valor del campo clave del registro a dar de alta. Usando un ciclo se recorre el archivo y en cada iteración se lee el registro actual usando *pickle.load()*. Con cada registro así leído desde el archivo, se compara su clave con el valor cargado por teclado.

Si se encuentra algún registro cuya clave coincida con este último, *pero que además no esté marcado como eliminado*, entonces la operación de *alta* se rechaza (pues ya existe un registro activo con esa clave). Si se llega al final del archivo y no se encontró repetida la clave, entonces se terminan de cargar por teclado los campos del registro a dar de alta, *se asigna en True su campo de marcado lógico*, y se graba el registro *al final* del archivo usando *pickle.dump()* (típicamente, el archivo debería estar abierto en modo 'a+b': es necesario poder leerlo para buscar el registro y saber si estaba repetido o no, y es necesario poder grabar el nuevo registro si fuese el caso, pero agregándolo al final).

Si en el archivo *se admiten registros con claves repetidas*, entonces el proceso de *alta* es mucho más simple pues ahora no es necesaria la fase de búsqueda para saber si el registro está repetido. Simplemente se abre el archivo en modo 'ab', se cargan por teclado los datos del nuevo registro, se pone su campo de marcado lógico en *True*, y se graba el registro sin más trámite (se grabará al final, porque el archivo se abrió en modo 'ab'). En el citado modelo *abm.py* que viene en el proyecto anexo a esta Ficha, *se supone que las repeticiones no se permiten*, y se deja como tarea para el alumno el adaptar ese programa para aceptar claves repetidas [1].

4.] Modificaciones y listados en un archivo de registros.

Otra tarea muy común es la de pedir la *modificación* de uno o más campos de un registro específico dentro del archivo. Para esto, el proceso es similar al de la *baja lógica*: se carga por teclado el valor de la clave del registro que se quiere modificar. Se busca dicho registro en el archivo usando un ciclo y leyendo y comparando uno por uno. Si se encuentra un registro cuya clave coincida con la buscada y tenga en *True* en el campo de marcado lógico,

se muestran los campos del mismo por pantalla, y con un pequeño menú de opciones se pide al operador que elija los campos que quiere modificar, cargando por teclado los valores nuevos.

Como esta modificación se está haciendo en memoria principal, todavía queda *volver a grabar el registro* en el disco pero *en su posición original*, para lo cual se usa *seek()* para volver atrás el *file pointer*, y se vuelve a grabar el registro con *pickle.dump()*. Decimos que este proceso es similar al de una *baja lógica*, porque en ambos se busca el registro, se lo sube a memoria, se modifica su contenido, y se vuelve a grabar en la posición original. La diferencia es que en la *baja* se cambia solo el valor del campo de marcado lógico, y en la *modificación* se cambia el resto de los campos (y no el campo de marcado lógico) [1].

Si bien el proceso general es el que acabamos de describir, en la práctica surge un inconveniente adicional debido al hecho de usar serialización para la grabación de registros. Cuando un registro se graba mediante *pickle.dump()* en la operación de alta, la función *pickle.dump()* transforma el registro en una secuencia de bytes de cierto tamaño *k*, y graba esa secuencia en el archivo. Pero en Python una variable ocupará tantos bytes como sea necesario para poder representar en binario el valor asignado en ella. Un número entero pequeño (por caso, el 10 o el 28) ocupará menos bytes que un número más grande (por caso, el 50000). Y una cadena de caracteres ocupará más bytes mientras más caracteres tenga.

Todo lo anterior quiere decir que dos registros del mismo tipo (por ejemplo, dos registros de tipo *Estudiante*) *no necesariamente ocuparán la misma cantidad de bytes*. Esto no es un problema en memoria principal, pero sí implica algunos inconvenientes cuando esos registros se graban en un archivo y se pretende aplicar un *proceso ABM* en ese archivo.

¿Cuál es concretamente el potencial problema? Suponga que un registro *r1* fue grabado en cierta posición *p1* del archivo cuando fue dado de alta, y también suponga que ese registro ocupaba una cantidad *k1* de bytes. Por lo tanto, el siguiente registro *r2* comenzará en la posición *p2 = p1 + k1*. Suponga ahora que se modifica el contenido del registro *r1*, pero al hacerlo suponga también que los nuevos valores de sus campos *aumentan* el tamaño del registro *r1* al valor *k2* (> *k1*). Cuando el registro *r1* se vuelva a grabar en su posición original *p1*, en lugar de ocupar *k1* bytes ocupará *k2* bytes, y como *k2 > k1*, entonces el registro *r1* ocupará bytes que antes le pertenecían en el archivo al registro *r2* (es decir, el registro *r1* ahora ocupa bytes que están más allá de la posición *p2 = p1 + k1*).

La consecuencia de esto es que se perderán bytes del registro *r2*, y se perderá también la sincronización en el proceso de lectura secuencial cuando se aplique *pickle.load()*: Cuando se intente leer el registro *r2*, el *file pointer* no estará ubicado en *p2 = p1 + k1* sino en la posición *p3 = p1 + k2*, y desde esa posición *pickle.load()* no podrá ya recuperar el registro original. El proceso completo terminará con un *error de runtime* cuando *pickle.load()* intente seguir leyendo y se encuentre de forma inesperada con el final del archivo.

Hay varias maneras de evitar estos inconvenientes. La forma más tradicional consiste en *forzar a cada registro a ocupar el mismo espacio k en bytes* cuando se grabe en el archivo, eliminando de esta forma todo problema (ya que todos los bloques que se graben, ahora serán del mismo tamaño *k*). En nuestro caso, y dado que Python modifica el tamaño de cada variable en función del valor que toma, aplicar esta solución no es tan simple ni tan directo y por lo tanto se deben formular algunas restricciones.

Por lo pronto, si un campo de un registro es de tipo cadena de caracteres haremos que esa cadena se *ajuste a una cantidad de caracteres prefijada por el programador*, llenándola con espacios en blanco a la derecha si fuese necesario. Por ejemplo, si el registro *r* tiene un campo llamado *nombre* asignado con la cadena *'Luis'* en ese campo, y cuando se quiera grabar el registro se decide que el *nombre* ocupe siempre 30 caracteres, se podría hacer en la forma siguiente:

```
r.nombre = 'Luis'
r.nombre = r.nombre.ljust(30)
```

El pequeño script anterior usa el método *ljust()* contenido en cualquier variable de tipo cadena de caracteres, para justificar hacia la izquierda el contenido de la cadena forzándola a tener tantos caracteres como indica el número tomado como parámetro. Si ese parámetro vale 30, se agregarán al final de la cadena tantos espacios en blanco como hagan falta para completar los 30 caracteres pedidos. Por lo tanto, si la cadena asignada era 'Luis', se agregarán 26 espacios en blanco a la derecha (la cadena quedará con 30 caracteres de largo, justificada hacia la izquierda).

Si el método *ljust()* se invoca pasando sólo un parámetro numérico (como en el ejemplo), entonces se asume que el relleno debe hacerse con *espacios en blanco*. Opcionalmente, se puede enviar un segundo parámetro indicando cuál es el caracter de relleno que debería usarse. Lo anterior, en este caso, sería lo mismo que:

```
r.nombre = 'Luis'
r.nombre = r.nombre.ljust(30, ' ')
```

El método *ljust()* ajusta el contenido de la variable para llegar a la cantidad de caracteres pedida, y finalmente retorna la nueva cadena.

Si el registro contiene campos numéricos de tipo entero, Python asignará a ese campo tantos bytes como necesite para representar en binario ese valor. Por lo tanto, otra vez, podríamos tener problemas con el cambio de tamaño de un registro si el mismo estaba grabado en el archivo y luego se modifica el valor de un campo de tipo entero (ese campo podría ahora ocupar más bytes que en el registro original). De nuevo, hay muchas formas de solucionar este problema, y el programador deberá esforzarse en cada caso para aplicar la mejor solución. Una estrategia es directamente manejar ese campo como una cadena de caracteres de tamaño fijo que sólo contenga dígitos, y cuando se necesite usar el valor en forma de número, convertirlo a número por medio de la función *int()*. Otra, es validar el valor de ese campo para asegurar que siempre tenga un valor en un intervalo conocido, y asegurarse que todos los números en ese intervalo tengan el mismo tamaño en bytes.

En general, la operación de modificación del contenido de un registro se hace sobre los campos que *no son la clave del registro* o *no forman parte de la clave del registro*. Si se desea modificar el campo clave, debe hacerse primero la baja del registro, y luego volver a darlo de alta con la nueva clave. En el ABM que estamos planteando para el archivo de estudiantes, el campo clave es el *legajo*, que es de tipo entero. Pero como no permitiremos la modificación de este campo, los problemas potenciales por el cambio de tamaño de ese campo no se producirán.

Finalmente, si el registro contiene algún campo numérico de tipo *float* entonces el programador no deberá preocuparse: los valores de tipo *float* se representan siempre con *doble precisión* (o sea, 8 bytes por número), y cuando un *float* es serializado mediante

pickle.dump() su representación se expande a 12 bytes por número en el archivo. En nuestro caso, el registro que representa a un estudiante contiene el campo *promedio*, de tipo *float*, que por lo ya indicado no será un problema en cuanto a posible cambio de tamaño.

La operación general de *modificación* es la última de las tres más básicas que se esperan en un *programa ABM*. Sin embargo, como dijimos, este tipo de programas puede incluir muchas otras opciones de gestión, la mayoría en forma de tareas de búsqueda y/o de listados de contenido.

Un proceso de *listado completo* es aquel que simplemente muestra el contenido completo del archivo, y se plantea en forma simple: el archivo se abre en modo de solo lectura (rb), con un ciclo se leen uno por uno sus registros y se muestran en pantalla a medida que se leen. Obviamente, los registros que se muestran son sólo aquellos que en el momento de ser leídos no estén marcados como eliminados. Y por otra parte, un *listado parcial* (también conocido como *listado con filtro*) es aquel en el cual se recorre y se lee todo el archivo, pero sólo se muestran en pantalla los registros que cumplen con cierta condición (por ejemplo, mostrar sólo los registros de los alumnos cuyo promedio sea mayor o igual a 7). Como las necesidades de consulta de un archivo suelen ser muchas y todas ellas con distintos criterios de filtrado, el menú de un *programa ABM* suele tener también muchísimas opciones y variantes. En el modelo que acompaña a esta Ficha hemos incluido los procesos esenciales (altas, bajas lógicas, modificaciones, depuración) y un par de opciones de listado de contenido (una para un listado completo, y otra para un listado con filtro). Vea la sección siguiente de esta Ficha.

5.] Desarrollo de un programa completo de gestión ABM.

Se propone ahora el estudio detallado de la estructura de un programa completo de *gestión ABM* que aplique los conceptos desarrollados en forma conceptual en la primera parte de esta Ficha. Formalmente, lo enunciamos como un problema a modo de caso de análisis:

Problema 60.) *Desarrollar un programa controlado por menú de opciones, que permita realizar en forma completa la gestión ABM de un archivo de registros de estudiantes de una carrera universitaria. Por cada estudiante, prevea tres campos para el legajo, el nombre y el promedio (además del campo de marcado lógico para las bajas lógicas). El programa debe incluir opciones que permitan:*

a.) *Realizar altas de registros en el archivo.*

b.) *Realizar bajas lógicas.*

c.) *Realizar la modificación de los datos de un registros.*

d.) *Mostrar el contenido completo del archivo.*

e.) *Mostrar los datos de los estudiantes con promedio mayor o igual a 7.*

f.) *Realizar la depuración del archivo (proceso de bajas físicas).*

Discusión y solución: El proyecto *[F26] Gestión ABM* que acompaña a esta Ficha contiene un modelo *abm.py* con el programa completo que resuelve este caso de análisis.

El programa *abm.py* comienza con la ya tradicional importación de módulos y la definición de la clase *Estudiante*, más los métodos *__init__()* y *__str__()* ya conocidos para manejar

objetos de esa clase, y dos funciones de validación de carga para los atributos *legajo* y *promedio*.

Note que la función constructora *__init__()* agrega e inicializa los campos en un objeto pero dentro de ese conjunto de campos ya incluye el atributo llamado *activo* de tipo *boolean*, al cual inicializa directamente con el valor *True* cuando un objeto es creado (sin tomar su valor como parámetro). Ese atributo *activo* es el que usaremos como *campo de marcado* para las *bajas lógicas* [1]:

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


class Estudiante:
    def __init__(self, leg, nom, prom):
        self.legajo = leg
        self.nombre = nom
        self.promedio = prom
        self.activo = True

    def __str__(self):
        r = ''
        r += '{:<20}'.format('Legajo: ' + str(self.legajo))
        r += '{:<30}'.format('Nombre: ' + self.nombre.strip())
        r += '{:<20}'.format('Promedio: ' + str(self.promedio))
        return r


def validar_legajo(inf, sup):
    n = inf - 1
    while n < inf or n > sup:
        n = int(input('Valor entre ' + str(inf) + ' y ' + str(sup) + ': '))
        if n < inf or n > sup:
            print('Era entre', inf, 'y', sup, '... cargue de nuevo...')
    return n


def validar_promedio():
    n = -1.0
    while n < 0.0 or n > 10.0:
        n = float(input('Valor entre 0 y 10 (puede tener decimales): '))
        if n < 0 or n > 10:
            print('Error... era entre 0 y 10... cargue de nuevo...')
    return n
```

Note también que el método *__str__()* convierte a cadena de caracteres a todos los atributos de un objeto, salvo el valor del atributo *activo*, que es un atributo de uso interno y no es necesario que se muestre luego al usuario final. Además, antes de concatenar a la cadena de salida el valor del atributo *nombre*, la función usa el método *strip()* que elimina los espacios en blanco que ese *nombre* pudiese tener al principio o al final (recuerde que para asegurar que el tamaño del registro se mantenga constante, al hacer la alta se ajustará la cantidad de caracteres del atributo *nombre* para agregar espacios en blanco y llegar a 30 caracteres).

Lo siguiente en el programa, y para facilitar los procesos, es el desarrollo de una función que permite *buscar* un registro con una clave dada dentro del archivo, y retorne la dirección de

ese registro si lo encuentra (por dirección del registro entendemos el número del byte dentro del archivo en el cual comienza el registro encontrado). Esa función se llama *buscar()* y es usada en las funciones de *altas*, *bajas lógicas* y *modificaciones*. Su estructura se muestra a continuación (el nombre del archivo es *estudiantes.utn* y se supone que ese nombre está almacenado en una variable ***global FD*** (creada y definida en la función *main()* que contiene el menú de opciones y el lanzamiento del programa):

```
def buscar(m, leg):
    global FD
    t = os.path.getsize(FD)

    fp_inicial = m.tell()
    m.seek(0, io.SEEK_SET)

    posicion = -1
    while m.tell() < t:
        fp = m.tell()
        est = pickle.load(m)
        if est.activo and est.legajo == leg:
            posicion = fp
            break

    m.seek(fp_inicial, io.SEEK_SET)
    return posicion
```

La función toma dos parámetros: el primero (*m*) es la variable *file object* que representa al archivo que se está gestionando. La función supone que ese archivo ya viene abierto, y además supone que viene abierto en un modo que permite lecturas. Si estos supuestos no se cumplen, *la función provocará un error de runtime y el programa se interrumpirá*.

El segundo parámetro (*leg*) es un número entero con el número de legajo que se quiere buscar en el archivo. La función buscará un registro con ese legajo. En caso de encontrarlo, detendrá la búsqueda y retornará el número del byte interno del archivo en el cual comienza el registro que se acaba de encontrar (por lo tanto, retornará en este caso un número mayor o igual a cero). Si el archivo no contiene ningún registro cuyo legajo coincida con *leg*, la función retornará el valor -1 (el cual entonces debe ser chequeado y entendido como un flag avisando que la búsqueda no tuvo éxito). La variable local *posicion* se usa para almacenar el valor a retornar: comienza valiendo -1 y si luego el registro es encontrado, su valor cambia para contener el número de su byte de inicio.

Como la función necesita leer el archivo desde el inicio (pues de lo contrario podrían quedar registros sin revisar), se usa el método *seek()* para llevar el file pointer al inicio del archivo (el byte número cero). Como se vio en la ficha anterior, esto puede hacerse así [2] [3]:

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

Luego de esto comienza el ciclo de lectura del archivo. Se aplica el ya conocido mecanismo de usar un ciclo *while* y controlar que el valor actual del *file pointer* sea menor que el tamaño en bytes del archivo. En cada iteración del ciclo, se lee el registro actual con pickle.load(), pero teniendo la precaución de almacenar previamente en la variable local *fp* el valor que en ese momento (antes de la lectura) tenía el *file pointer*. De esta forma, si el registro que luego se lee con *pickle.load()* fuese efectivamente el registro que se estaba buscando, la variable *fp* contendrá la dirección o número del byte donde ese registro comenzaba, y sólo deberá copiarse su valor en la variable *posicion* para retornarlo al final (recuerde que cada vez que

se lee o graba en un archivo, el valor del *file pointer* se actualiza para quedar apuntando al byte inmediatamente siguiente a aquel en el cual terminó la lectura o la grabación, por lo cual el valor retornado por *tell()* después de leer no será la dirección del registro leído, sino la del siguiente…) [1].

Un detalle final: como la función recibe el archivo ya abierto, el *file pointer* del mismo está ya posicionado en algún byte particular. Y es de esperar que el programador que haya invocado a la función buscar() espere que el archivo no sólo vuelva abierto, sino que además vuelva con el file pointer apuntando al mismo lugar donde el mismo estaba antes de invocar a la función (aunque en este caso el contexto no exige que se cumple ese requisito). Por ese motivo, antes de volver el *file pointer* al inicio del archivo y comenzar el ciclo de lectura, se invoca a *tell()* y se almacena la posición actual del *file pointer* en la variable *fp_inicial*. Cuando el ciclo termina, y antes de retornar el resultado final, se vuelve a usar *seek()* para llevar el *file pointer* a la posición indicada por *fp_inicial*, dejando el archivo tal como estaba antes de comenzar la búsqueda.

La función *alta()* es la encargada de agregar nuevos registros al archivo. Su estructura es la que sigue:

```
def alta():
    global FD
    m = open(FD, 'a+b')

    print()
    print('Legajo del estudiante a registrar (cargue 0 para salir): ')
    leg = validar_legajo(0, 99999)
    while leg != 0:
        # buscamos el registro con ese legajo...
        pos = buscar(m, leg)
        if pos == -1:
            # no estaba repetido... lo cargamos por teclado...
            nom = input('Nombre: ')

            # ...ajustar a 30 caracteres, llenando con blancos al final...
            nom = nom.ljust(30, ' ')

            print('Promedio...')
            pro = validar_promedio()

            est = Estudiante(leg, nom, pro)

            # ...lo grabamos...
            pickle.dump(est, m)

            # ...volcamos el buffer de escritura
            # para que el sistema operativo REALMENTE
            # grabe en disco el registro...
            m.flush()

            print('Registro grabado en el archivo...')

        else:
            print('Legajo repetido... alta rechazada...')

        print()
        print('Otro legajo a registrar (cargue 0 para salir): ')
        leg = validar_legajo(0, 99999)
```

```
    m.close()

    print()
    print('Operación de altas finalizada...')
    input('Presione <Enter> para seguir...')
```

La idea es la que ya hemos explicado en la sección 3 de esta ficha. El archivo se abre en modo 'a+b', de forma que si el mismo no existía, será creado, y en caso de existir, su contenido será preservado. Cuando se pida grabar (con *pickle.dump()*) el nuevo registro será agregado al final.

La función usa un ciclo de carga por doble lectura para leer por teclado un legajo (controlando que sea diferente de cero). Si es así, se usa la función *buscar()* ya explicada, para determinar si existe o no un registro con ese legajo. Si ya existiese un registro con ese legajo, la operación de alta se rechaza (no admitiremos alumnos con registros duplicados) y se carga otro legajo. Sólo si el legajo no estuviese repetido, se carga el resto de los campos y se graba con *pickle.dump()* el nuevo registro. Recuerde que el método *__init__()* asigna el valor *True* en el campo activo de marcado lógico.

Al cargar por teclado el *nombre* del estudiante, se usa el ya citado método *ljust()* para agregar espacios en blanco al final del atributo *nombre*, ajustar el tamaño del atributo a 30 caracteres, y evitar así futuros problemas de cambio de tamaño cuando se modifique ese objeto.

El único detalle extraño es que luego de invocar a *pickle.dump()* para grabar el registro, se está invocando a otro método *flush()* que hasta aquí no habíamos utilizado. El motivo: cuando un archivo se abre para grabación, el verdadero responsable de grabar los nuevos datos en el mismo es el *sistema operativo* (y no el lenguaje de alto nivel que se esté usando). Pero por razones de eficiencia y de mejor aprovechamiento del tiempo de trabajo del procesador, el sistema operativo utiliza una zona de memoria intermedia (o *buffer*) en la cual va guardando los datos que el programa solicita grabar. Sólo cuando ese buffer está lleno, el sistema operativo va al disco y efectivamente graba los datos en el archivo. Mientras tanto, esos datos no están realmente grabados… por lo que en este caso, cuando la función *buscar()* es invocada para determinar si existe o no un registro con legajo repetido podría llegar a informar que un registro que *se acaba de agregar* no existe, simplemente porque sus datos aún no fueron grabados en el archivo.

Cuando se invoca al método *close()*, todos los datos que el sistema operativo tenía en el buffer del archivo se vuelcan al mismo. Pero en el caso de la función *altas()*, el archivo no se cierra hasta terminar el ciclo de carga por doble lectura (se graban varios registros antes de cerrar el archivo). Para no tener que cerrar y volver a abrir el archivo cada vez que se graba un registro, se usa el método *flush()*: este método simplemente vuelca el contenido del buffer del archivo y lo graba en el mismo en forma efectiva, sin tener que esperar a que ese buffer se llene o que se invoque a *close()* [2].

Note finalmente que el problema que acabamos de describir sólo se da cuando el archivo se abre en *modo de grabación* (de hecho, *flush()* no tiene ningún efecto si el archivo está abierto en modo de sólo lectura).

La función *baja()* lleva a cabo el proceso de *baja lógica* (eliminación por marcado lógico de un registro):

```
def baja():
    global FD
    if not os.path.exists(FD):
        print('El archivo', FD, 'no existe...')
        print()
        return

    m = open(FD, 'r+b')

    print()
    print('Legajo del estudiante a borrar (cargue 0 para salir): ')
    leg = validar_legajo(0, 99999)
    while leg != 0:
        # buscamos el registro con ese legajo...
        pos = buscar(m, leg)
        if pos != -1:
            # encontrado... procedemos a cargarlo...
            m.seek(pos, io.SEEK_SET)
            est = pickle.load(m)

            # ...mostramos el registro tal como estaba...
            print()
            print('El registro actualmente grabado es:')
            print(est))

            # ...chequemos si el usario está seguro de lo que hace...
            r = input('Está seguro de querer borrarlo (s/n)?: ')
            if r in ['s', 'S']:
                # lo marcamos como borrado, y ya...
                est.activo = False

                # ...reubicamos el file pointer...
                m.seek(pos, io.SEEK_SET)

                # ...y lo volvemos a grabar...
                pickle.dump(est,m)

                print()
                print('Registro eliminado del archivo...')

        else:
            print('Ese registro no existe en el archivo...')

        print()
        print('Otro legajo de estudiante a borrar (cargue 0 para salir): ')
        leg = validar_legajo(0, 99999)

    m.close()

    print()
    print('Operación de bajas finalizada...')
    input('Presione <Enter> para seguir...')
```

Esta función también usa un ciclo de carga por lectura doble para poder hacer varias operaciones de baja de registros si fuese necesario. Se carga un legajo, y se invoca a la función *buscar()* otra vez. Si no existe un registro con ese legajo, la baja no puede hacerse y se pide un nuevo legajo.

Si el legajo existe, se usa *seek()* para posicionar el *file pointer* en el byte donde comienza ese registro (valor que fue retornado por *buscar()* en ese caso), y se lee el registro con

pickle.load(). Antes de eliminarlo, se muestra el registro en pantalla y se pregunta al usuario si está seguro de querer eliminarlo. Si el usuario confirma la operación, simplemente se cambia el valor del campo *activo* por el valor *False*, y se vuelve a grabar el registro en la misma posición original.

Sólo hay que recordar que cuando el registro fue leído, el *file pointer* se movió al registro siguiente, por lo que antes de volver a grabarlo se debe hacer retroceder el *file pointer* a la posición donde el registro comienza: esto requiere una segunda invocación a *seek()* (a la misma posición retornada por *buscar()*) y recién entonces invocar a *pickle.dump()* para volver a grabar el registro [1].

La función *modificacion()* es la que permite cambiar el valor de alguno de los campos de un registro. Es muy similar a la función *baja()*, pero implica algo más de trabajo en la interfaz de usuario (incluyendo *el ajuste a 30 caracteres del campo nombre* si este fuese vuelto a leer desde el teclado):

```
def modificacion():
    global FD
    if not os.path.exists(FD):
        print('El archivo', FD, 'no existe...')
        print()
        return

    m = open(FD, 'r+b')

    print()
    print('Legajo del estudiante a modificar (cargue 0 para salir): ')
    leg = validar_legajo(0, 99999)
    while leg != 0:
        # buscamos el registro con ese legajo...
        pos = buscar(m, leg)
        if pos != -1:
            # encontrado... procedemos a cargarlo...
            m.seek(pos, io.SEEK_SET)
            est = pickle.load(m)

            # ...mostramos el registro tal como estaba...
            print()
            print('El registro actualmente grabado es:')
            print(est))

            # ...modificamos el valor de los campos...
            op = 0
            while op != 3:
                print('1. Modificar nombre.')
                print('2. Modificar promedio.')
                print('3. Terminar modificaciones.')
                op = int(input('\t\tIngrese opción: '))

                if op == 1:
                    nom = input('Nuevo nombre: ')
                    est.nombre = nom.ljust(30, ' ')

                elif op == 2:
                    print('Nuevo promedio:')
                    est.promedio = validar_promedio()

                elif op == 3:
                    pass
```

```
            # ...registro modificado en memoria...
            # ...ahora nos volvemos a su posición en el archivo...
            m.seek(pos, io.SEEK_SET)

            # ...y volvemos a grabar el registro modificado...
            pickle.dump(est, m)

            print()
            print('Los datos del registro se actualizaron...')

        else:
            print('Ese registro no existe en el archivo...')

        print('Otro legajo a modificar (cargue 0 para salir): ')
        leg = validar_legajo(0, 99999)

    m.close()

    print()
    print('Operación de modificaciones finalizada...')
    input('Presione <Enter> para seguir...')
```

Esta función trabaja en la misma forma que la función *baja()*. La diferencia es que ahora, cuando se muestra el registro encontrado, en lugar de preguntar al usuario si está seguro de querer borrarlo se muestra un pequeño menú de opciones en el cual se le da a elegir al operador el campo cuyo valor quiere cambiar. Cuando esos cambios han sido realizados, el registro se vuelve a grabar en la misma posición en la que fue leído, teniendo en cuenta los mismos detalles ya señalados en cuanto al *file pointer* para la función *baja()* (aunque en este caso, obviamente, el valor del campo de marcado lógico se mantiene en *True*) [1].

La función encargada de depurar el archivo y proceder a eliminar físicamente los registros marcados como eliminados, es la función *depuracion()*:

```
def depuracion():
    global FD
    if not os.path.exists(FD):
        print('El archivo', FD, 'no existe...')
        print()
        return

    tbm = os.path.getsize(FD)

    original = open(FD, 'rb')
    temporal = open('temporal.dat', 'wb')

    print('Procediendo a optimizar el archivo', FD)
    input('Presione <Enter> para seguir...')

    while original.tell() < tbm:
        # cargar un registro del archivo original...
        est = pickle.load(original)

        # ...si no estaba marcado, grabarlo en el archivo temporal...
        if est.activo:
            pickle.dump(est, temporal)

    # cerrar ambos archivos...
    original.close()
```

```
    temporal.close()

    # eliminar el archivo original...
    os.remove(FD)

    # y renombrar el temporal...
    os.rename('temporal.dat', FD)

    print('Optimización terminada... se eliminaron los registros marcados')
    input('Presione <Enter> para seguir...')
```

El archivo *original* se abre en modo de sólo lectura ('rb'), y el *temporal* en modo de creación y eliminación de contenido anterior ('wb'). Un ciclo *while* lee uno por uno los registros del *original*, y a cada uno cuyo campo *activo* valga *True* lo graba a su vez en el *temporal*.

Cuando el ciclo termina, ambos archivos se cierran. Se procede entonces a eliminar el archivo original, con la función *os.remove()*. Esta toma como parámetro el *nombre físico del archivo* (y *no* la variable *file object* para gestión del archivo) que se quiere eliminar y simplemente procede a removerlo del sistema de archivos del disco. La función *os.remove()* no se limita a vaciar el contenido del archivo (cosa que se puede hacer abriendo el mismo en modo 'wb'), sino que lo elimina efectivamente de su carpeta contenedora [2].

Una vez eliminado el *original*, se debe renombrar el *temporal* para que asuma el nombre físico que tenía el *original*. Esto se hace con la función *os.rename()*, la cual toma dos parámetros: el *viejo nombre físico* del archivo a renombrar, y el *nuevo nombre físico* que se quiere dar al archivo. La función cambia el *viejo nombre* del archivo en el sistema de archivos, y lo reemplaza por el *nuevo nombre*. Note que tanto la función *os.remove()* como la función *os.rename()* requieren que los archivos con los que se va a trabajar *estén cerrados* (no se puede eliminar o renombrar un archivo que esté en uso) y ambas pertenecen al *módulo os* que fue convenientemente importado al inicio del programa [2].

Las funciones encargadas de mostrar el contenido del archivo (en forma completa o en forma filtrada) son directas y sencillas, por lo que dejamos su análisis para el estudiante. Lo mismo vale para la función *main()* que contiene el menú de opciones y es el punto de entrada del programa. El programa completo se muestra a continuación:

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


class Estudiante:
    def __init__(self, leg, nom, prom):
        self.legajo = leg
        self.nombre = nom
        self.promedio = prom
        self.activo = True

    def __str__(self):
        r = ''
        r += '{:<20}'.format('Legajo: ' + str(self.legajo))
        r += '{:<30}'.format('Nombre: ' + self.nombre.strip())
        r += '{:<20}'.format('Promedio: ' + str(self.promedio))
        return r


def validar_legajo(inf, sup):
```

```
    n = inf - 1
    while n < inf or n > sup:
        n = int(input('Valor entre ' + str(inf) + ' y ' + str(sup) + ': '))
        if n < inf or n > sup:
            print('Era entre', inf, 'y', sup, '... cargue de nuevo...')
    return n


def validar_promedio():
    n = -1.0
    while n < 0.0 or n > 10.0:
        n = float(input('Valor entre 0 y 10 por favor (puede tener
decimales): '))
        if n < 0 or n > 10:
            print('Error... se pidió entre 0 y 10... cargue de nuevo...')
    return n


def buscar(m, leg):
    global FD
    t = os.path.getsize(FD)

    fp_inicial = m.tell()
    m.seek(0, io.SEEK_SET)

    posicion = -1
    while m.tell() < t:
        fp = m.tell()
        est = pickle.load(m)
        if est.activo and est.legajo == leg:
            posicion = fp
            break

    m.seek(fp_inicial, io.SEEK_SET)
    return posicion


def alta():
    global FD
    m = open(FD, 'a+b')

    print()
    print('Legajo del estudiante a registrar (cargue 0 para salir): ')
    leg = validar_legajo(0, 99999)
    while leg != 0:
        # buscamos el registro con ese legajo...
        pos = buscar(m, leg)
        if pos == -1:
            # no estaba repetido... lo cargamos por teclado...
            nom = input('Nombre: ')

            # ...ajustamos a 30 caracteres, rellenando con blancos...
            nom = nom.ljust(30, ' ')

            print('Promedio...')
            pro = validar_promedio()

            est = Estudiante(leg, nom, pro)

            # ...lo grabamos...
            pickle.dump(est, m)
```

```
            # ...volcamos el buffer de escritura
            # para que el sistema operativo REALMENTE
            # grabe en disco el registro...
            m.flush()

            print('Registro grabado en el archivo...')

        else:
            print('Legajo repetido... alta rechazada...')

        print()
        print('Otro legajo de estudiante a registrar (con cero termina): ')
        leg = validar_legajo(0, 99999)

    m.close()

    print()
    print('Operación de altas finalizada...')
    input('Presione <Enter> para seguir...')


def baja():
    global FD
    if not os.path.exists(FD):
        print('El archivo', FD, 'no existe. Use la opción 1 para crearlo')
        print()
        return

    m = open(FD, 'r+b')

    print()
    print('Legajo del estudiante a borrar (cargue 0 para salir): ')
    leg = validar_legajo(0, 99999)
    while leg != 0:
        # buscamos el registro con ese legajo...
        pos = buscar(m, leg)
        if pos != -1:
            # encontrado... procedemos a cargarlo...
            m.seek(pos, io.SEEK_SET)
            est = pickle.load(m)

            # ...mostramos el registro tal como estaba...
            print()
            print('El registro actualmente grabado es:')
            print(est)

            # ...chequemos si el usario está seguro de lo que hace...
            r = input('Está seguro de querer borrarlo (s/n)?: ')
            if r in ['s', 'S']:
                # lo marcamos como borrado, y ya...
                est.activo = False

                # ...reubicamos el file pointer...
                m.seek(pos, io.SEEK_SET)

                # ...y lo volvemos a grabar...
                pickle.dump(est, m)

                print()
                print('Registro eliminado del archivo...')
```

```
        else:
            print('Ese registro no existe en el archivo...')

        print()
        print('Otro legajo de estudiante a borrar (cargue 0 para salir): ')
        leg = validar_legajo(0, 99999)

    m.close()

    print()
    print('Operación de bajas finalizada...')
    input('Presione <Enter> para seguir...')


def modificacion():
    global FD
    if not os.path.exists(FD):
        print('El archivo', FD, 'no existe. Use la opción 1 para crearlo')
        print()
        return

    m = open(FD, 'r+b')

    print()
    print('Legajo del estudiante a modificar (cargue 0 para salir): ')
    leg = validar_legajo(0, 99999)
    while leg != 0:
        # buscamos el registro con ese legajo...
        pos = buscar(m, leg)
        if pos != -1:
            # encontrado... procedemos a cargarlo...
            m.seek(pos, io.SEEK_SET)
            est = pickle.load(m)

            # ...mostramos el registro tal como estaba...
            print()
            print('El registro actualmente grabado es:')
            print(est)

            # ...modificamos el valor de los campos...
            op = 0
            while op != 3:
                print('1. Modificar nombre.')
                print('2. Modificar promedio.')
                print('3. Terminar modificaciones.')
                op = int(input('\t\tIngrese opción: '))

                if op == 1:
                    nom = input('Nuevo nombre: ')
                    est.nombre = nom.ljust(30, ' ')

                elif op == 2:
                    print('Nuevo promedio:')
                    est.promedio = validar_promedio()

                elif op == 3:
                    pass

            # ...registro modificado en memoria...
            # ...ahora nos volvemos a su posición en el archivo...
```

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

            # ...y volvemos a grabar el registro modificado...
            pickle.dump(est, m)

            print()
            print('Los datos se actualizaron en el archivo...')

        else:
            print('Ese registro no existe en el archivo...')

        print('Otro legajo de estudiante a modificar (con 0 termina): ')
        leg = validar_legajo(0, 99999)

    m.close()

    print()
    print('Operación de modificaciones finalizada...')
    input('Presione <Enter> para seguir...')


def depuracion():
    global FD
    if not os.path.exists(FD):
        print('El archivo', FD, 'no existe. Use la opción 1 para crearlo')
        print()
        return

    tbm = os.path.getsize(FD)

    original = open(FD, 'rb')
    temporal = open('temporal.dat', 'wb')

    print('Optimizando el archivo', FD, '(eliminación registros borrados)')
    while original.tell() < tbm:
        # cargar un registro del archivo original...
        est = pickle.load(original)

        # ...y si no estaba marcado, grabarlo en el archivo temporal...
        if est.activo:
            pickle.dump(est, temporal)

    # cerrar ambos archivos...
    original.close()
    temporal.close()

    # eliminar el archivo original...
    os.remove(FD)

    # y renombrar el temporal...
    os.rename('temporal.dat', FD)

    print('Terminado... registros marcados eliminados...')
    input('Presione <Enter> para seguir...')


def listado_completo():
    global FD
    if not os.path.exists(FD):
        print('El archivo', FD, 'no existe. Use la opción 1 para crearlo')
        print()
```

```
        return

    tbm = os.path.getsize(FD)

    m = open(FD, 'rb')

    print('Listado general de estudiantes registrados:')
    while m.tell() < tbm:
        est = pickle.load(m)
        if est.activo:
            print(est)

    m.close()

    print()
    input('Presione <Enter> para seguir...')


def listado_filtrado():
    global FD
    if not os.path.exists(FD):
        print('El archivo', FD, 'no existe. Use la opción 1 para crearlo')
        print()
        return

    tbm = os.path.getsize(FD)

    m = open(FD, 'rb')

    print('Listado de estudiantes con promedio mayor o igual a 7:')
    while m.tell() < tbm:
        est = pickle.load(m)
        if est.activo and est.promedio >= 7:
            print(est)

    m.close()

    print()
    input('Presione <Enter> para seguir...')


def main():
    global FD
    FD = 'estudiantes.utn'

    op = 0
    while op != 7:
        print('Opciones ABM del archivo de estudiantes')
        print('   1. Alta de estudiantes')
        print('   2. Baja de estudiantes')
        print('   3. Modificación de estudiantes')
        print('   4. Listado completo de estudiantes')
        print('   5. Listado de estudiantes con promedio >= 7')
        print('   6. Depuración del archivo de estudiantes')
        print('   7. Salir')
        op = int(input('\t\tIngrese número de la opción elegida: '))
        print()

        if op == 1:
            alta()
```

```
        elif op == 2:
            baja()

        elif op == 3:
            modificacion()

        elif op == 4:
            listado_completo()

        elif op == 5:
            listado_filtrado()

        elif op == 6:
            depuracion()

        elif op == 7:
            pass


# script principal...
if __name__ == '__main__':
    main()
```

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.