Ficha 24

Archivos de Texto

1.] Introducción.

Hemos visto que en general en programación se acepta clasificar a los archivos en *archivos binarios* y *archivos de texto*, y hasta aquí nos hemos concentrado en la gestión de *archivos binarios*. También hemos indicado que en realidad *todo archivo es binario*, ya que en todo archivo finalmente se almacenan secuencias de bits que representan cualquier clase de información almacenada en el archivo [1]. Sin embargo, por razones de claridad y simplificación de terminología, es normal hablar de *archivos de texto* para referirse a ciertos tipos particulares de archivos binarios en los que cada byte del archivo se grabó con la intención real y efectiva de hacer que ese byte represente un caracter visualizable o forme parte de la representación de un caracter visualizable (y no, por ejemplo, de hacer que un conjunto de bytes agrupados representen un número entero o un número flotante).

Existen en ese sentido numerosos estándares de codificación binaria de caracteres. Esos estándares están basados en tablas de conversión en las que a cada caracter posible se le asocia un número, y la conversión a binario de ese número representa finalmente al caracter. La tabla más conocida, y también la más simple y una de las más antiguas usadas en la industria informática es la tabla *ASCII* (por *American Standard Code for Information Interchange*), que también se conoce como *ASCII 7* o también como *US-ASCII*. En el estándar *ASCII 7* cada caracter se representa exactamente con un byte, dejando el primer bit en 0 y usando el resto de los siete bits para representar el número de orden del caracter. Como sólo se usan siete bits, se pueden representar entonces no más de 128 caracteres (lo cual en un principio era consistente con las necesidades mínimas del mercado: letras del alfabeto inglés, dígitos, algunos signos de puntuación, algunos operadores y un puñado de caracteres de control para indicar saltos de línea, retornos de carro, saltos de página, etc.)

Sin embargo, con el correr del tiempo se hizo evidente que un conjunto tan reducido de caracteres era insuficiente, sobre todo con el advenimiento de los sistemas operativos de interfaz gráfica y el uso cada vez más globalizado de la informática. Así comenzaron entonces a aparecer variantes y expansiones del estándar *ASCII* original. Una de esas variantes se conoce como *ASCII 8* y emplea los ocho bits de un byte para representar caracteres, con lo cual el número total se extiende a 256 caracteres (y es la variante que se usó para incorporar caracteres especiales no incluidos en el idioma inglés pero comunes en otros idiomas (como los de base latina): la ñ del español, la ç del francés o el portugués, o las letras vocales acentuadas, por ejemplo.

Otras expansiones del estándar fueron claramente necesarias: con sólo 256 combinaciones no era posible incorporar caracteres de alfabetos no latinos (como el griego) o de alfabetos orientales (como el japonés, el chino o el coreano), ni representar caracteres icónicos tan

comunes y cotidianos hoy en día (como "las caritas felices" por ejemplo… ☺)[1]. Para lograr mayor amplitud en la representación, se planteó el estándar *Unicode* con fundamentos más complejos pero mucho más amplio que el estándar *ASCII*. Esencialmente, *Unicode* admite como un subconjunto válido al estándar *ASCII*, por lo que un texto codificado en *ASCII* será perfectamente legible por una aplicación o programa basado en *Unicode*.

A su vez, se han planteado tres formas de codificación de caracteres en base al estándar *Unicode*, designadas como *UTF-8*, *UTF-16* y *UTF-32* (y siendo *UTF* la abreviatura de *Unicode Trasformation Format* o *Formato de Transformación Unicode*), y a su vez estas tres formas dan lugar a numerosos esquemas técnicos de codificación cuya explicación escapa a los alcances de esta Ficha. En general, es suficiente con saber que mediante *Unicode* (en cualquiera de las tres formas de codificación que se emplee) se puede representar la totalidad de los caracteres existentes en todos los idiomas, incluidas lenguas muertas, lenguas orientales, lenguas latinas y no latinas y caracteres gráficos especiales.

El lenguaje Python emplea por default el estándar *Unicode* bajo la forma de codificación *UTF-8* para interpretación de texto [2]. Como el estándar *ASCII* está incluido como un subconjunto de *Unicode*, cualquier archivo de texto con formato *ASCII* puro será leído y gestionado sin problemas. Cuando se abre un archivo de texto en Python, se pueden leer y grabar cadenas de caracteres en ese archivo con relativa sencillez. Los caracteres serán interpretados al leerlos (o codificados al grabarlos) en formato *UTF-8*.

Un detalle que tiene su importancia en cuanto a la posibilidad de usar archivos de texto en distintas plataformas, es la forma en que se interpretan los *caracteres de fin de línea* que estén contenidos en un archivo de texto. Básicamente, un *caracter de fin de línea* es un *caracter especial de control*, que se inserta en un texto para que al ser leído se interprete de forma de producir un *cambio de renglón* o *salto de línea* cuando ese texto se muestre en pantalla.

Como sabemos, un *caracter de control* se representa con una barra (\) seguida de un caracter que es el que especifica de qué caracter de control se trata. El hecho es que en distintos sistemas operativos el caracter de control de salto de línea se representa en formas diferentes, trayendo luego problemas cuando el mismo archivo debe ser leído en distintas plataformas. En el sistema operativo Unix (así como en Linux) el caracter de fin de línea se representa con \n, mientras en Windows se representa con una pareja de caracteres de la forma \r\n (esto es, un retorno de carro (\r) seguido de un salto de línea (\n)).

Para evitar problemas debidos a esta incómoda situación, cuando en Python ***se lee un archivo de texto*** la acción por default es convertir en la cadena leída los caracteres de fin de línea que aparezcan (el \n de Unix o el par \r\n de Windows) simplemente en un \n. De esta

[1] Los modernos sistemas de codificación digital de caracteres parecerían cosa de magia para los hombres de las épocas previas al siglo XV en que apareció la primera imprenta de tipos móviles (atribuida a Johannes Gutenberg). En esos tiempos medievales los libros se escribían, copiaban e ilustraban a mano; y era muy común en Occidente que esa tarea fuera realizada por monjes en las bibliotecas de los monasterios cristianos. La película *The Name Of The Rose* (o *El Nombre de la Rosa*) de 1986, dirigida por *Jean-Jacques Annaud* y protagonizada por *Sean Connery*, está ambientada en una de esas bibliotecas y gira alrededor de una serie de crímenes que parecen sobrenaturales. El fraile Guillermo de Baskerville llega al convento para investigar esos misteriosos asesinatos y devolver la paz a los monjes, en un contexto tenebroso donde no falta la superstición, el éxtasis religioso, la Inquisición, la hipocresía y algún que otro romance *non sancto*... La película está basada en la novela original del mismo nombre del genial *Umberto Eco*, publicada en 1980.

forma, cualquiera sea la plataforma de origen del archivo, los saltos de renglón en la pantalla serán interpretados en forma consistente. Y en forma recíproca, cuando ***se graba en un archivo de texto*** la acción por default es convertir los caracteres \n que las cadenas a grabar contengan y grabar en el archivo los caracteres que correspondan según el sistema operativo huésped en ese momento (es decir, si se está trabajando en Windows y se graba una cadena de caracteres en un archivo de texto, los caracteres \n que esa cadena tenga se grabarán convertidos en parejas \r\n) [2].

Como esta acción eventualmente modifica el contenido original del archivo, el programador debe estar especialmente seguro que el archivo que está procesando es de texto, y abrirlo como tal (colocando la letra '**t**' en el modo de apertura, o no colocando ni la '**t**' ni la '**b**' (que como vimos, equivale por default a colocar una '**t**'). Si se abre en modo texto un archivo pensado como binario (por ejemplo, un archivo *.exe* o un *.jpg* o un *.pdf*), estos cambios podrían corromper el contenido del archivo, perdiendo datos o haciendo imposible abrirlo en forma normal. Si el archivo *no es de texto*, entonces, debería abrirse colocando la '**b**' en el modo de apertura, lo cual *desactiva* el reemplazo de caracteres de salto de línea [2].

2.] Uso de archivos de texto en Python.

La forma abrir y usar un archivo de texto en Python es esencialmente la misma que la explicada para archivos binarios. Un archivo de texto también debe abrirse para poder operar con él, y la apertura se hace con la misma función *open()* ya analizada para los archivos binarios. La diferencia, es que al indicar el modo de apertura deberá reemplazarse la letra 'b' que se agregaba para los archivos binarios, por una 't', o bien, no escribir ninguna de las dos letras (lo que por default implica que el archivo es de texto) [1]. El significado de los modos de apertura es exactamente el mismo que el de los equivalentes para archivos binarios (consulte la tabla de modos de apertura de la *Ficha 22*). La siguiente instrucción abre el archivo *'datos.txt'* como archivo de texto, en modo de grabación y lectura:

```
m = open('datos.txt', 'w+t')
```

y como se indicó, es lo mismo que:

```
m = open('datos.txt', 'w+')
```

Una vez que el archivo ha sido abierto, la variable *file object* (en nuestro caso, *m*) que representa al archivo dispone de varios métodos (tales como *write()*, *read()*, *readline()* y *readlines()*) que permiten grabar o leer cadenas de caracteres con sencillez desde ese archivo.

En ese sentido, surge aquí una diferencia práctica entre lo que hacíamos para grabar y leer registros en un archivo binario y lo que haremos para grabar o leer cadenas en un archivo de texto: para grabar o leer registros empleábamos el mecanismo de *serialización*, a través de las funciones *pickle.dump()* y *pickle.load()* para simplificar el proceso, ya que los métodos *read()* y *write()* provistos por las variables *file object* requerían un poco de trabajo extra para poder operar con registros o variables de estructura compleja.

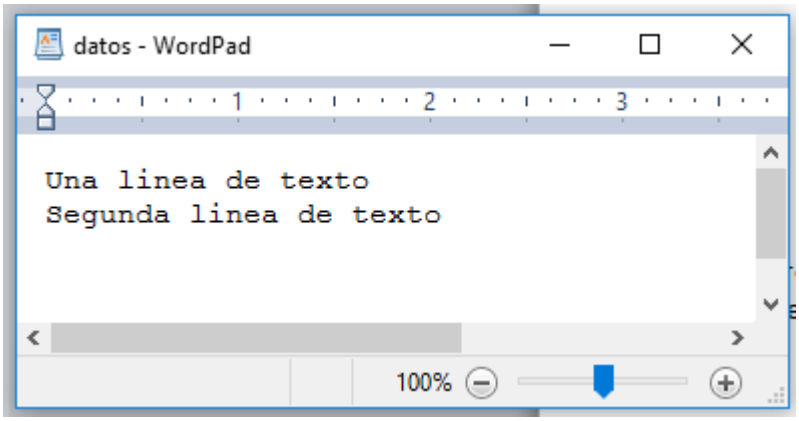
Sin embargo, esos mismos métodos permiten la grabación o la lectura de una cadena de caracteres en un archivo de texto en forma muy directa y muy simple. El método *write()* graba una cadena de caracteres en el archivo, y retorna la cantidad de caracteres que efectivamente grabó [2] [3] (en la función del ejemplo siguiente, el valor retornado por

write() está siendo ignorado). Note que el archivo debe estar abierto en un modo que permita grabaciones, y que este método ***no agrega*** un salto de línea al final de la cadena, por lo cual el programador debe incluir un caracter '\n' explícitamente si lo desea:

```
def grabar_lineas():
    m = open('datos.txt', 'wt')
    m.write('Una linea de texto\n')
    m.write('Segunda linea de texto')
    m.close()
```

El archivo *datos.txt* generado al invocar la función *grabar_lineas()* que acabamos de mostrar, contendrá dos líneas de texto, tal como se ve a continuación:

Figura 1: Contenido del archivo *datos.txt* creado con la función *grabar_lineas()* (*con* salto de línea).

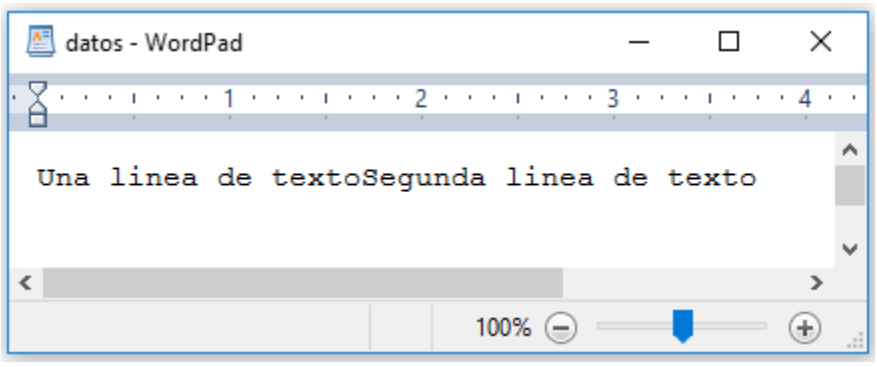


Como se indicó, el método *write()* no agrega un caracter de salto de línea al final de la cadena cuando la graba: es responsabilidad del programador agregar ese caracter cuando lo requiera. Si el ejemplo anterior se hubiese planteado así (sin el caracter '\n' al final de la primera cadena):

```
def grabar_lineas():
    m = open('datos.txt', 'wt')
    m.write('Una linea de texto')
    m.write('Segunda linea de texto')
    m.close()
```

entonces el contenido del archivo generado se vería así:

Figura 2: Contenido del archivo *datos.txt* creado con la función *grabar_lineas()* (*sin* salto de línea).



Igual que con los archivos binarios, El método *close()* se usa para cerrar la conexión con el archivo y liberar cualquier recurso que estuviese asociado a él. Luego de invocar a *close()*, la variable que representaba al archivo queda indefinida. Recuerde, además, que en Python los archivos son cerrados automáticamente cuando la variable usada para accederlos sale del ámbito en que fue definida. Por lo tanto, en la función anterior no es estrictamente necesario invocar a *close()*.

Por otra parte, el método *read()* (invocado sin parámetros) permite leer el *contenido completo del archivo*, retornándolo como una sola y única cadena de caracteres. Se puede pasar como parámetro un número entero positivo, en cuyo caso *read(n)* leerá y retornará exactamente *n* bytes desde el archivo. Los caracteres '\n' que el archivo pudiera tener *se preservan* y se copian en la cadena retornada:

```
def leer_todo():
    m = open('datos.txt', 'r')
    todo = m.read()
    print('Contenido completo:')
    print(todo)
    m.close()
```

Si el archivo *datos.txt* fue generado con un salto de línea entre la primera cadena y la segunda (como se muestra en la *Figura 1*), entonces la salida en pantalla producida por la invocación de la función *leer_todo()* que se acaba de mostrar será la que sigue:

```
Contenido completo:
Una linea de texto
Segunda linea de texto
```

Otro método útil para procesar archivos de texto es el método *readline()*, que permite leer *una línea simple del archivo*, retornándola como una cadena de caracteres. La lectura se iniciará desde donde se encuentre el *file pointer* en ese momento, *y terminará en el primer salto de línea que encuentre o al llegar al final del archivo*. Si no puede leer una nueva cadena desde el archivo, por ejemplo por haber llegado ya al final del mismo, el método *readline()* retorna una cadena vacía ('') [2] [3]:

```
def leer_lineas():
    m = open('datos.txt')
    l1 = m.readline()
    l2 = m.readline()
    print('\nContenido linea por linea:')
    print(l1, end='')
    print(l2)
    m.close()
```

Note que *readline() mantiene al final de la cadena retornada* el caracter '\n', el cual sólo es omitido en la línea final del archivo (si el mismo no termina a su vez con '\n'). Es por este motivo que en la función anterior hemos agregado el parámetro *end=''* en la segunda invocación a *print()*, para evitar que aparezca una línea en blanco innecesaria en la salida por consola.

Como el archivo *datos.txt* contiene dos líneas de texto separadas por un salto de línea, la función *leer_lineas()* invoca dos veces a *readline()* para poder recuperar las dos líneas: la primera invocación recupera la cadena *'Una línea de texto'* y sólo esa cadena, ya que la misma termina con un '\n'. El *file pointer* queda entonces apuntando al primer caracter de la segunda cadena, la cual es recuperada por la segunda invocación a *readline()*. La cadena

leída será entonces *'Segunda línea de texto'*, la cual será retornada sin un '\n' al final, ya que esa cadena finaliza en el mismo lugar donde termina el archivo (y sin '\n').

En el ejemplo anterior, el archivo *datos.txt* contenía sólo dos líneas, y para leerlo línea por línea en forma completa bastaban dos invocaciones a *readline()*. Pero si el archivo contuviese una cantidad mucho mayor de líneas y/o no se conociese esa cantidad, entonces una forma alternativa y simple de leer el archivo completo, línea por línea, consiste en usar un ciclo *for* que *itere sobre el contenido del file object*:

```
def leer_con_iterador():
    m = open('datos.txt')
    print('\nContenido con iterador:')
    for line in m:
        print(line, end='')
    m.close()
```

Lo anterior es simple y cómodo, pero a veces se requiere mayor control sobre el flujo de lectura. Se puede implementar un ciclo que vaya leyendo el archivo con *readline()*, verificando si esa función retornó o no la cadena vacía (indicador de que se ha llegado al final del archivo). En la función *lectura_controlada()* que sigue mostramos cómo hacer esto, y agregamos además un pequeño y útil proceso sobre la cadena leída, para eliminar el caracter de salto línea del final mediante un corte de índices [2] [3]:

```
def lectura_controlada():
    m = open('datos.txt')
    print('\nContenido con control de lectura:')
    while True:
        # intentar leer una linea...
        line = m.readline()

        # si se obtuvo una cadena vacia... cortar el ciclo y terminar...
        if line == '':
            break

        # si lo tenía, eliminar el caracter de salto de línea...
        if line[-1] == '\n':
            line = line[:-1]

        # procesar la cadena resultante...
        print(line)

    # cerrar antes de irnos...
    m.close()
```

Si la variable *line* es una cadena de caracteres, la expresión if not line: equivale a la expresión if line == "": con lo cual la función mostrada está detectando el final del archivo y cortando el ciclo con un *break*. La expresión line = line[:-1] está tomando *todo el contenido de la cadena line salvo el último caracter*, y asignando la cadena cortada nuevamente en *line*. El resultado de esto es que se elimina el caracter '\n' que *line* tenía al final.

Otro método disponible para leer desde un archivo de texto es *readlines()* (con una *s* al final…). Si es invocado sin parámetros, retorna *una lista* (un arreglo) que contiene a todas las líneas del archivo (***incluidos*** los '\n' al final de cada una). Se puede invocar al método pasándole como parámetro un valor numérico, en cuyo caso *readlines(n)* intentará recuperar hasta *n bytes* del archivo. En este caso, solo incluirá en la lista retornada las cadenas *completas* que haya logrado leer:

```
def list_lines():
    m = open('datos.txt', 'r')
    lista = m.readlines()
    print('\nLista de lineas contenidas:')
    print(lista)
    m.close()
```

Puede ver la implementación de todos los ejemplos y modelos que se han analizado hasta aquí en el modelo *test01.py* del proyecto *[F24] Archivos de Texto* que acompaña a esta Ficha.

3.] Reposicionamiento del file pointer en un archivo de texto en Python.

Para finalizar con el manejo de archivos de texto en Python, digamos que en forma similar a lo que ya hemos estudiado para los archivos binarios en Python pueden emplearse los mismos métodos *tell()* y *seek()* para consultar y cambiar el valor del *file pointer* de un archivo de texto, de forma de poder eventualmente realizar operaciones de lectura y/o grabación por acceso directo al byte en el cual se quiere comenzar (vea los ejemplos de uso y aplicación en el modelo *test02.py* en el proyecto *[F24] Archivos de Texto* que acompaña a esta Ficha).

Como sabemos, el método *tell()* retorna un número entero con el valor que en ese momento tenga el *file pointer* del archivo, medido en la cantidad de bytes desde el inicio del mismo y considerando al primer byte como en la posición 0(cero). Recuerde que si el *file pointer* está ubicado al final de un archivo (el primer byte inmediatamente luego del último que pertenece al archivo), entonces el valor retornado por *tell()* indica el tamaño en bytes del archivo (al igual que lo que ya vimos para archivos binarios) [1]:

```
m = open('prueba', 'w')
m.write('Universidad')
pos = m.tell()

# muestra: 11
print('Posicion del file pointer / Tamaño del archivo:', pos)
```

No confundir *tamaño del archivo en bytes*, con *cantidad de caracteres del archivo*. En el formato *Unicode* – esquema *UTF-8*, un caracter puede llegar a ser representado con 1, 2, 3 o 4 bytes dependiendo de cuál sea ese caracter, por lo que no necesariamente un tamaño de *k* bytes es equivalente a *k* caracteres en un archivo de texto.

El ya conocido método *seek()* permite cambiar el valor del *file pointer* también en un archivo de texto, aunque ahora existen algunas restricciones menores que distinguen su uso respecto de los archivos binarios.

Vimos que *seek()* en general recibe dos parámetros: el primero indica cuántos bytes debe moverse el *file pointer*, y el segundo indica desde donde se hace ese salto (el valor *io.SEEK_SET = 0* indica saltar desde el principio del archivo, el valor *io.SEEK_CUR = 1* indica saltar desde donde está el *file pointer* en ese momento y el valor *io.SEEK_END = 2* indica saltar desde el final). El valor por default del segundo parámetro es ***io.SEEK_SET = 0***, por lo cual por omisión se asume que los saltos son desde el inicio del archivo (revise la *Ficha 22* para repasar estos conceptos si lo necesita).

Pero note que si se está trabajando con *archivos de texto*, en general *solo pueden hacerse saltos desde el principio del archivo (si el primer parámetro es un valor diferente de cero, entonces el segundo parámetro obligatoriamente debe ser io.SEEK_SET)*. Si el segundo

parámetro enviado a *seek()* es *io.SEEK_END o io.SEEK_CUR, entonces el primer parámetro debe ser obligatoriamente igual a 0.* Si el archivo fue abierto como *archivo de texto*, *cualquier otra combinación de valores para el primer parámetro y el segundo parámetro de seek() provocará un error de tiempo de ejecución y el programa se interrumpirá* [2]:

```
def pruebas():
    m = open('prueba', 'w')
    m.write('Universidad')
    pos = m.tell()
    print('Posicion del file pointer / Tamaño del archivo:', pos)

    # reposicionamiento correcto:
    m.seek(4, io.SEEK_SET)

    pos = m.tell()
    print('Posicion del file pointer ahora:', pos)

    # reposicionamiento correcto:
    pos = m.seek(0, io.SEEK_END)

    print("Posicion del file pointer ahora: ", pos)

    # lo siguiente provoca un error...
    m.seek(3, io.SEEK_CUR)
```

Si la función anterior fuese ejecutada tal como está, el resultado será algo como:

```
Traceback (most recent call last):
  File "C:/Documentos/test02.py", line 31, in <module>, line 25, in main
    m.seek(3, io.SEEK_CUR)
io.UnsupportedOperation: can't do nonzero cur-relative seeks
```

Y si se ejecutase la siguiente instrucción en el mismo programa:

```
    # lo siguiente también provoca un error...
    m.seek(10, io.SEEK_END)
```

también obtendremos una situación error como la que sigue:

```
Traceback (most recent call last):
  File "C:/Documentos/test02.py", line 19, in main
    m.seek(10, io.SEEK_END)
io.UnsupportedOperation: can't do nonzero end-relative seeks
```

Está claro que si se invoca a *seek()* en esta forma:

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

el efecto será mover el *file pointer* al final del archivo, y eso es válido tanto en archivos de texto como binarios. Y si se hace algo como:

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

entonces el *file pointer* simplemente mantendrá el valor que ya tenía (tanto si el archivo es de texto como si es binario), sin cambios. Esto puede parecer poco útil, pero al menos no provoca un error si el archivo es de texto.

Existen otros métodos (y algunos atributos) incluidos en una variable/objeto de tipo *file object* en Python, ***aplicables tanto para archivos de texto como para archivos binarios***.

Algunos han sido presentados en fichas anteriores pero mostramos los más comunes en la tabla siguiente:

Tabla 1: Algunos métodos / atributos comunes incluidos en objetos tipo file object en Python.

Método o Atributo	Aplicación
closed	Es un atributo o campo (no un método) cuyo valor es *True* si el archivo está cerrado.
flush()	Vuelca al archivo los buffers de grabación, si corresponde. Aplicable sólo para archivos abiertos para grabar (no hace nada en caso contrario).
readable()	Retorna True si el archivo está disponible para ser leído.
truncate(size=None)	Trunca el contenido del archivo a una cantidad igual a *size* bytes. El archivo puede *aumentar* o *disminuir* su tamaño.
writable()	Retorna *True* si el archivo está disponible para ser grabado.
writelines(lines)	Graba el contenido de la lista *lines* en el archivo, por defecto ***sin*** incluir separadores de línea (por lo cual, el programador debe indicar ese separador al final de cada elemento de la lista *lines* si quiere los separadores).

El siguiente problema servirá para mostrar la forma general de aplicar técnicas esenciales de manejo de archivos de texto:

Problema 56.) *Desarrollar un programa con menú de opciones que permita abrir un archivo que se sabe es de texto, y proceda a operar con él a partir de las siguientes opciones:*

a.) *Cargar por teclado el nombre y la extensión de un archivo. Almacenar ese nombre en una variable fd de uso general para el resto de las opciones, y regresar al menú.*

b.) *Crear el archivo fd (si ya existía, limpiar su contenido y abrirlo vacío) y grabar en él un texto cargado por teclado por el usuario.*

c.) *Abrir el archivo fd, mostrar su contenido completo y permitir que el usuario cargue nuevo texto desde el teclado y se agregue al final del archivo. Si no existe el archivo, crearlo y también permitir que se grabe texto nuevo en el archivo, cargando ese texto por teclado.*

d.) *Comprobar si existe el archivo fd, y en ese caso abrirlo y mostrar su contenido completo. Si no existe, informar con un mensaje y retornar al menú.*

e.) *Comprobar si existe el archivo fd, y en ese caso abrirlo y determinar cuántas veces ese archivo contenía una palabra **x** que se carga por teclado antes de la consulta.*

f.) *Comprobar si existe el archivo fd, y en ese caso abrirlo y **duplicar su contenido**: agregar al final del archivo (sin eliminar el contenido anterior) una copia completa de su contenido actual.*

g.) *Comprobar si existe el archivo fd, y en ese caso abrirlo y **truncar su contenido** para que finalmente sólo contenga tantas líneas como indique la variable **cl** que se carga por teclado (de las que ya tenía el archivo), eliminando todo el resto. Si el archivo tenía menos de **cl** líneas de texto, no hacer nada y regresar al menú.*

Discusión y solución: El proyecto *[F24] Archivos de Texto* que acompaña a esta Ficha contiene un modelo *test03.py* con el programa completo que resuelve este caso de análisis.

La función que hemos designado como *main()* será la función o punto de entrada del programa. Contiene el menú de opciones y la declaración inicial de la variable *fd* en la que se almacenará el nombre físico del archivo. Si nunca se selecciona la opción 1, esa variable

quedará asignada por defecto con la cadena *'default.txt'* (y ese será el nombre del archivo a manejar en ese caso):

```
def main():
    # nombre físico por default del archivo...
    fd = 'default.txt'

    op = 0
    while op != 8:
        print('Opciones para gestión del archivo de texto:', fd)
        print('   1. Cargar nombre físico del archivo')
        print('   2. Crear archivo nuevo y grabar texto')
        print('   3. Abrir archivo y agregar texto al final')
        print('   4. Mostrar contenido completo del archivo')
        print('   5. Buscar y contar una palabra o cadena en el archivo')
        print('   6. Duplicar contenido del archivo (en el mismo archivo)')
        print('   7. Truncar contenido del archivo a dos líneas')
        print('   8. Salir')
        op = int(input('\t\tIngrese número de la opción elegida: '))
        print()

        if op == 1:
            fd = cargar_nombre()

        elif op == 2:
            crear_archivo(fd)

        elif op == 3:
            abrir_archivo(fd)

        elif op == 4:
            listado_completo(fd)

        elif op == 5:
            buscar_cadena(fd)

        elif op == 6:
            duplicar_contenido(fd)

        elif op == 7:
            truncar_archivo(fd)

        elif op == 8:
            pass


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

La función *cargar_nombre()* invocada al seleccionar la opción 1, es simple y directa: carga por teclado el nombre físico que se quiera utilizar desde ese momento, y lo retorna (con lo que a partir de ese momento, el valor de fd en la función *main()* cambia y con eso cambia también el nombre del archivo a utilizar en el resto del programa):

```
def cargar_nombre():
    fd = input('Ingrese nombre del archivo (con extensión si lo desea): ')
    print('Ok... nombre registrado...')
    print()
    return fd
```

La función *crear_archivo(fd)* toma como parámetro el nombre del archivo que debe utilizar, y procede a crearlo si no existía, o a abrirlo limpiar su contenido si ya existía. En ambos casos, a continuación usa un ciclo para cargar por teclado una secuencia de cadenas de

caracteres, y graba una por una esas cadenas en el archivo. Como la función *input()* que se usa para la carga por teclado de una cadena, elimina el salto de línea del final, entonces al grabar la cadena en el archivo con *m.write()* hemos agregado el caracter '\n' en forma explícita:

```
def crear_archivo(fd):
    print('Archivo a crear:', fd)
    m = open(fd, 'wt')

    print('Ingrese líneas de texto, presionando <Enter> al final de cada una.')
    print('Para terminar la carga, incluya los caracteres ".-" al final')
    print()

    line = input()
    while not line.endswith('.-'):
        m.write(line + '\n')
        line = input()

    # eliminar el último guión...
    line = line[:-1]

    # grabar la última línea...
    m.write(line + '\n')

    m.close()
    print()
    print('Archivo creado y guardado...')
    print()
```

Como la carga de datos se da por terminada cuando el usuario ingrese una cadena terminada con los caracteres '.-', entonces el ciclo de carga debe controlar que la cadena recién ingresada no termine con esos caracteres. El método *endswith(cad)* que viene incluido en toda variable de tipo cadena de caracteres permite chequear si la cadena con la que se invoca al método termina o no con los caracteres de la cadena *cad* tomada como parámetro, y ese método es el que se usa en el ciclo para controlar si debe continuar o detenerse.

Cuando esos caracteres aparecen en la última cadena line leída, el ciclo se detiene, pero esa última cadena debe todavía ser grabada en el archivo (el ciclo cortó antes de poder grabarla…) Para eso, se elimina el guión que la cadena tenía al final (*line = line[:-1]*), y se invoca a *write()* para grabarla, añadiendo el consabido '\n' al final.

La siguiente función es *abrir_archivo(fd)*, la cual abre el archivo cuyo nombre toma como parámetro, pero en modo 'a+' para crearlo si no existía pero preservar su contenido si existía. Lo primero que esta función hace es leer línea por línea el contenido del archivo por medio de un for iterador. Note que antes de comenzar la lectura, se movió el *file pointer* al inicio del archivo (*m.seek(0, io.SEEK_SET)*) ya que al abrir el archivo en modo 'a+' el *file pointer* queda ubicado al final, y eso haría que el ciclo de lectura no llegue a leer ninguna línea.

El resto de la función es similar a lo dicho para *crear_archivo()*: se cargan por teclado nuevas líneas y se graban en el archivo. Al hacerlo se agregarán al final de ese archivo debido a que está abierto en modo 'a':

```
def abrir_archivo(fd):
    print('Archivo a abrir:', fd)
    m = open(fd, 'a+t')

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

```
    print('Contenido actual del archivo:')
    print()
    for line in m:
        print(line, end='')

    print()
    print()
    print('Ingrese nuevas líneas de texto, para agregar al archivo,')
    print('presionando <Enter> al final de cada una.')
    print('Para terminar la carga, incluya los caracteres ".-" al final')
    print()

    line = input()
    while not line.endswith('.-'):
        m.write(line + '\n')
        line = input()

    # eliminar el guión del final...
    line = line[:-1]

    # grabar la última línea...
    m.write(line + '\n')

    m.close()
    print()
    print('Archivo modificado y guardado...')
    print()
```

Sigue la función *listado_completo(fd)*, que abre el archivo en modo de sólo lectura, y simplemente lo lee con un *for iterador* mostrando su contenido línea por línea. Como cada línea es recuperada del archivo ***incluyendo*** el caracter se salto de línea que tenía en ese archivo, entonces al mostrar cada cadena se agrega el parámetro *end=''* en la invocación a *print()*, evitando con esto que se hagan dos saltos de línea en lugar de uno al visualizar la cadena:

```
def listado_completo(fd):
    if not os.path.exists(fd):
        print('El archivo no existe... use opciones 2 o 3 para crearlo...')
        print()
        return

    print('Archivo a mostrar:', fd)
    print()
    m = open(fd, 'rt')

    print('Contenido del archivo:')
    print()
    for line in m:
        print(line, end='')

    m.close()
    print()
    print()
    print('Archivo visualizado...')
    print()
```

La función *buscar_cadena(fd)* abre el archivo y también lo lee línea por línea con un *for iterador*, pero ahora en lugar de mostrar cada cadena, invoca al método *count()* que esas cadenas contienen. Este método toma como parámetro otra cadena (aquí llamada *x*), y retorna la cantidad de veces que esa cadena *x* está contenida dentro de la cadena original con la cual se invoca al método. Los valores retornados por *count()* en cada vuelta del ciclo

se van acumulando en la variable *c*, y al finalizar el ciclo se muestra su valor (que será entonces la cantidad total de veces que *x* estaba en el archivo):

```
def buscar_cadena(fd):
    if not os.path.exists(fd):
        print('El archivo'no existe... use opciones 2 o 3 para crearlo...')
        print()
        return

    print('Archivo a controlar:', fd)
    m = open(fd, 'rt')

    c = 0
    x = input('Ingrese la cadena a buscar: ')
    for line in m:
        c += line.count(x)

    m.close()
    print()
    print('La cadena', x, 'fue encontrada', c, 'veces en el archivo')
    print()
```

La función *duplicar_contenido(fd)* es realmente simple: abre el archivo, lee todo su contenido de una sola vez mediante el método *read()*, y la cadena así obtenida la graba de inmediato al final del archivo… con lo cual el archivo duplica su tamaño y contiene ahora una copia replicada de su contenido anterior. Como el archivo debe ser leído y también grabado, el modo de apertura elegido es 'r+': al abrirlo, el *file pointer* apunta al inicio, el método *read()* lee desde allí todo el contenido y deja el *file pointer* al final, con lo que luego el método *write()* graba la cadena al final (no fueron necesarias en este caso, invocaciones a *seek()*):

```
def duplicar_contenido(fd):
    if not os.path.exists(fd):
        print('El archivo no existe... use opciones 2 o 3 para crearlo...')
        print()
        return

    print('Archivo a expandir:', fd)
    m = open(fd, 'r+t')

    # leer el contenido completo...
    todo = m.read()

    # grabar ese contenido al final...
    m.write(todo)

    m.close()
    print()
    print('Se duplicó y grabó el contenido del archivo')
    print()
```

La última función de este programa se llama *truncar_archivo(fd)*, y es la que lleva a cabo el requerimiento de *truncar* el contenido del archivo a una cantidad de líneas *cl* que se carga por teclado. La función abre el archivo en modo 'r+', y luego usa un ciclo *while* para leer el archivo con la función *readline()*, controlando si se cargó la cadena vacía o no.

Por lo que hemos ya indicado, si *readline()* carga una cadena vacía, significa que se ha llegado al final del archivo. Además, con cada lectura se incrementa en 1 el contador *c*, de forma que si el valor de *c* llegase a ser igual a *cl*, se habrá leído entonces la cantidad de líneas que se quiere que el archivo finalmente contenga. En ese momento se corta el ciclo con *break*, y la función continúa controlando si efectivamente el ciclo cortó porque se leyeron *cl*

líneas: en ese caso, el *file pointer* está apuntando al byte donde específicamente queremos que termine el archivo. Ese valor (recuperado con *m.tell()*) se pasa como parámetro al método ***truncate()***, el cual corta el contenido del archivo haciendo que desde ese momento en adelante termine justamente en ese byte.

```
def truncar_archivo(fd):
    if not os.path.exists(fd):
        print('El archivo no existe... use opciones 2 o 3 para crearlo...')
        print()
        return

    print('Archivo a truncar:', fd)
    m = open(fd, 'r+t')

    c = 0
    cl = int(input('En cuántas líneas quiere truncar el archivo?: '))
    line = m.readline()
    while line != '':
        c += 1
        if c == cl:
            break
        line = m.readline()

    if c == cl:
        m.truncate(m.tell())
        print('Se truncó el contenido del archivo a', cl, 'lineas')
    else:
        print('No se truncó el archivo... no hay suficientes líneas...')

    m.close()
    print()
```

El programa completo se muestra a continuación:

```
import io
import os.path


def cargar_nombre():
    fd = input('Ingrese el nombre del archivo (incluya si quiere una extensión): ')
    print('Ok... nombre registrado...')
    print()
    return fd


def crear_archivo(fd):
    print('Archivo a crear:', fd)
    m = open(fd, 'wt')

    print('Ingrese líneas de texto, presionando <Enter> al final de cada una.')
    print('Para terminar la carga, incluya los caracteres ".-" al final')
    print()

    line = input()
    while not line.endswith('.-'):
        m.write(line + '\n')
        line = input()

    # eliminar el último guión...
    line = line[:-1]

    # grabar la última línea...
    m.write(line + '\n')

    m.close()
    print()
```

```
    print('Archivo creado y guardado...')
    print()


def abrir_archivo(fd):
    print('Archivo a abrir:', fd)
    m = open(fd, 'a+t')

    m.seek(0, io.SEEK_SET)
    print('Contenido actual del archivo:')
    print()

    for line in m:
        print(line, end='')

    print()
    print()
    print('Ingrese nuevas líneas de texto, para agregar al archivo,')
    print('presionando <Enter> al final de cada una.')
    print('Para terminar la carga, incluya los caracteres ".-" al final')
    print()

    line = input()
    while not line.endswith('.-'):
        m.write(line + '\n')
        line = input()

    # eliminar el guión del final...
    line = line[:-1]

    # grabar la última línea...
    m.write(line + '\n')

    m.close()
    print()
    print('Archivo modificado y guardado...')
    print()


def listado_completo(fd):
    if not os.path.exists(fd):
        print('El archivo no existe... use las opciones 2 o 3 para crearlo...')
        print()
        return

    print('Archivo a mostrar:', fd)
    print()
    m = open(fd, 'rt')

    print('Contenido del archivo:')
    print()
    for line in m:
        print(line, end='')

    m.close()
    print()
    print()
    print('Archivo visualizado...')
    print()


def buscar_cadena(fd):
    if not os.path.exists(fd):
        print('El archivo no existe... use las opciones 2 o 3 para crearlo...')
        print()
        return

    print('Archivo a controlar:', fd)
```

```
    m = open(fd, 'rt')

    c = 0
    x = input('Ingrese la cadena a buscar: ')
    for line in m:
        c += line.count(x)

    m.close()
    print()
    print('La cadena', x, 'fue encontrada', c, 'veces en el archivo')
    print()


def duplicar_contenido(fd):
    if not os.path.exists(fd):
        print('El archivo no existe... use las opciones 2 o 3 para crearlo...')
        print()
        return

    print('Archivo a expandir:', fd)
    m = open(fd, 'r+t')

    # leer el contenido completo...
    todo = m.read()

    # grabar ese contenido al final...
    m.write(todo)

    m.close()
    print()
    print('Se duplicó y grabó el contenido del archivo')
    print()

def truncar_archivo(fd):
    if not os.path.exists(fd):
        print('El archivo no existe... use las opciones 2 o 3 para crearlo...')
        print()
        return

    print('Archivo a truncar:', fd)
    m = open(fd, 'r+t')

    c = 0
    cl = int(input('En cuántas líneas quiere truncar el archivo?: '))
    line = m.readline()
    while line != '':
        c += 1
        if c == cl:
            break
        line = m.readline()

    if c == cl:
        m.truncate(m.tell())
        print('Se truncó el contenido del archivo a', cl, 'lineas')
    else:
        print('No se truncó el archivo... no hay suficientes líneas...')

    m.close()
    print()


def main():
    # nombre físico por default del archivo...
    fd = 'default.txt'

    op = 0
    while op != 8:
        print('Opciones para gestión del archivo de texto:', fd)
```

```
        print('   1. Cargar nombre físico del archivo')
        print('   2. Crear archivo nuevo y grabar texto')
        print('   3. Abrir archivo y agregar texto al final')
        print('   4. Mostrar contenido completo del archivo')
        print('   5. Buscar y contar una palabra o cadena en el archivo')
        print('   6. Duplicar contenido del archivo (en el mismo archivo)')
        print('   7. Truncar contenido del archivo a dos líneas')
        print('   8. Salir')
        op = int(input('\t\tIngrese número de la opción elegida: '))
        print()

        if op == 1:
            fd = cargar_nombre()

        elif op == 2:
            crear_archivo(fd)

        elif op == 3:
            abrir_archivo(fd)

        elif op == 4:
            listado_completo(fd)

        elif op == 5:
            buscar_cadena(fd)

        elif op == 6:
            duplicar_contenido(fd)

        elif op == 7:
            truncar_archivo(fd)

        elif op == 8:
            pass


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

4.] Procesamiento de archivos de texto que contienen secuencias numéricas.

En muchas ocasiones se trabaja con archivos de texto que contienen series de números expresados como cadenas. Típicamente en estos casos, cada línea contiene un "número" y el salto de línea separa a un número del siguiente. Normalmente, cuando se necesita procesar un archivo así, la idea es tomar las cadenas que representan números en el archivo, convertirlas a números en el formato correcto (*int* o *float*), almacenar esos números en un arreglo, y finalmente retornar el arreglo (u operar con él).

La razón por la cual alguien guardaría números como cadenas en un archivo, es la de evitar *problemas de incompatibilidad de formatos numéricos*, si se prevé que distintos programadores usen lenguajes diferentes para procesar el mismo archivo. El formato de texto también tiene diversas variantes, pero es siempre más sencillo compatibilizar formatos de texto que numéricos, sobre todo si el archivo de texto fue generado usando un juego de caracteres estándar (ASCII, UTF-8, etc.)

En el proyecto *[F24] Archivos de Texto* ya citado, se incluye el modelo *test04.py* en cual aparecen dos funciones designadas como *save_numbers(n, f)* y *read_numbers(f)*. La primera toma como parámetro un número entero positivo *n* y el nombre *f* de un archivo, y graba en el archivo *f* un total de *n+1* líneas de texto que representan números, pero como cadenas. La

primera línea del archivo *f* es el propio valor de *n*, convertido a cadena de caracteres, a modo de indicador de la cantidad de números que contiene el archivo. Y las restantes líneas, son números aleatorios que pueden estar repetidos, pero de forma que todos pertenecen al intervalo *[1, n]* (ambos incluidos), y convertidos a cadenas de caracteres:

```
import random

def save_numbers(n, f='lista.txt'):
    # asegurar el nombre por default del archivo...
    if not isinstance(f, str):
        f = 'lista.txt'

    # crear el archivo de texto, en modo grabacion...
    m = open(f, 'w')

    # primera linea: la cantidad de numeros que tendrá el archivo...
    m.write(str(n) + '\n')

    # luego, n lineas: una por numero, en forma aleatoria en [1, n]...
    for i in range(n):
        # pedir un entero aleatorio entre 1 y n (incluidos ambos)
        x = random.randint(1, n)

        # grabar el numero como string, con salto de linea final...
        m.write(str(x) + '\n')

    m.close()
```

La segunda función (*read_numbers(f)*) abre el archivo *f* como archivo de texto en modo de solo lectura (*f* tiene por default el nombre *'lista.txt'*), lee la primera línea para saber cuántos números trae el archivo (y por añadidura, en qué rango vienen esos números…) y luego lee el resto de las líneas con un ciclo, elimina el salto de línea que cada una tenga al final, convierte la cadena resultante a un entero válido y agrega ese entero a un arreglo, por orden de llegada. Al finalizar el ciclo, la función retorna el arreglo:

```
def read_numbers(f='lista.txt'):
    # asegurar el nombre por default del archivo...
    if not isinstance(f, str):
        f = 'lista.txt'

    # el arreglo a retornar, inicialmente vacio...
    lista = []

    # abrir el archivo de texto en modo de solo lectura...
    m = open(f)

    # la primera linea del archivo nos da el valor de n...
    # ...sirva o no, leerla y eliminar el '\n' del final...
    ln = m.readline()
    ln = ln[:-1]

    # ...convertirla a int...
    n = int(ln)

    # leer el resto del archivo...
    while True:
        # leer el siguiente numero...
        xs = m.readline()
```

```
        # controlar fin de archivo...
        if xs == '':
            break

        # si lo tenía, eliminar el caracter de salto de línea...
        # ... y convertir a entero...
        if xs[-1] == '\n':
            xs = xs[:-1]
        x = int(xs)

        # agregar el entero al arreglo en orden de llegada...
        lista.append(x)

    # cerrar el archivo y retornar el arreglo...
    m.close()
    return lista
```

El siguiente programa completo (modelo *test04.py*) contiene estas dos funciones y una simple función *test()* que las invoca, para mostrar finalmente la forma general de uso de ambas. La misma función *test()* es la que actúa como punto de entrada del programa:

```
import random


def save_numbers(n, f='lista.txt'):
    # asegurar el nombre por default del archivo...
    if not isinstance(f, str):
        f = 'lista.txt'

    # crear el archivo de texto, en modo grabacion...
    m = open(f, 'w')

    # primera linea: grabar como string la cantidad de numeros que del archivo...
    m.write(str(n) + '\n')

    # luego n lineas: una por cada numero, generado en forma aleatoria en [1, n]...
    for i in range(n):
        # pedir un entero aleatorio entre 1 y n (incluidos ambos)
        x = random.randint(1, n)

        # grabar el numero convertido a string, con salto de linea al final...
        m.write(str(x) + '\n')

    m.close()


def read_numbers(f='lista.txt'):
    # asegurar el nombre por default del archivo...
    if not isinstance(f, str):
        f = 'lista.txt'

    # el arreglo a retornar, inicialmente vacio...
    lista = []

    # abrir el archivo de texto en modo de solo lectura...
    m = open(f)

    # la primera linea del archivo nos da el valor de n...
    # ...sirva o no, leerla y eliminar el '\n' del final...
    ln = m.readline()
    ln = ln[:-1]

    # ...convertirla a int...
    n = int(ln)
```

```
    # leer el resto del archivo...
    while True:
        # leer el siguiente numero...
        xs = m.readline()

        # controlar fin de archivo...
        if xs == '':
            break

        # si lo tenía, eliminar el caracter de salto de línea...
        # ... y convertir a entero...
        if xs[-1] == '\n':
            xs = xs[:-1]
        x = int(xs)

        # agregar el entero al arreglo en orden de llegada...
        lista.append(x)

    # cerrar el archivo y retornar el arreglo...
    m.close()
    return lista


def test():
    save_numbers(40)
    lis = read_numbers()
    print('La lista leida es:')
    print(lis)
    print('Cantidad de elementos:', len(lis))


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.