El manejo de archivos y de texto

El manejo de archivos suele causar problemas a los principiantes, aunque nunca entendí porqué sucede esto. Desde el punto de vista de la programación un archivo no difiere en nada de los que utilizamos en un procesador de texto o en cualquier otra aplicación: simplemente lo *abrimos*, ejecutamos algún tipo de operación sobre él y luego lo volvemos a *cerrar*.

Sin embargo, la diferencia más importante es que en un programa accedemos a los archivos de manera *secuencial*, es decir, se lee una línea por vez desde el comienzo del archivo. En la práctica, un procesador de texto realiza la misma operación, sólo que mantiene el archivo en la memoria mientras uno trabaja en él y luego lo guarda en el disco una vez finalizada la edición. Otra diferencia a tener en cuenta es que podemos abrir un archivo para ser leído o para ser escrito. Podemos escribir en un archivo al crearlo de la nada (o sobreescribiendo uno ya existente) o agregando información al final de un archivo preexistente a la manera de un apéndice.

Otra característica importante es que podemos volver al comienzo del archivo mientras lo estamos procesando.

Archivos - Entrada y Salida

Veamos un ejemplo. Supongamos que existe un archivo llamado menu.txt que contiene una lista de comidas:

```
ensalada de tomate
papas fritas
pizza
```

Ahora vamos a escribir un programa que lea el archivo y nos muestre su contenido (como los comandos 'cat' en Unix o 'type' en DOS:

```
# Primero abrimos el archivo en modo lectura (r)
inp = open("menu.txt","r")
# leemos el archivo, colocamos el contenido en una lista
# e imprimimos cada item
for linea in inp.readlines():
    print linea
# Ahora lo cerramos
inp.close()
```

Nota 1: `open()` requiere dos argumentos. El primero es el nombre del archivo (que puede ser pasado como una variable o directamente como una cadena de caracteres, como hicimos en el ejemplo). El segundo es el *modo de acceso*. El modo determina en qué forma se abrirá el archivo: para lectura (r) o para escritura (w). El modo también indica el tipo de contenido del archivo; así, agregando "b" a la "r" o "w" indicaremos que el archivo es binario (por ejemplo, `open(na,"rb")`; de lo contrario se da por supuesto de que se trata de un archivo en formato ASCII de texto.

Nota 2: Leemos y cerramos el archivo mediante funciones prefijadas con la variable de archivo. Esta notación se conoce como *invocación de métodos* y es nuestro primer vistazo a la *Programación Orientada a Objetos*. Por ahora no te preocupes, simplemente fijate que esto se relaciona de algún modo con lo que hemos visto en el uso de los módulos. Podés pensar en una variable de archivo como una referencia a un módulo que contiene una serie de funciones que operan sobre los archivos y que importamos automáticamente cada vez que creamos esta variable.

Consideremos ahora qué ocurre con los archivos más extensos. En primer lugar deberemos leer el archivo de a una línea por vez (en Python utilizaremos `readline()` en lugar de `readlines()`. También deberemos usar una variable cuenta_lineas que se incremente con cada línea leída; evaluaremos esta variable para ver si llegó a 25 (el número de líneas en la pantalla) y en ese caso pedirle al usuario que pulse una tecla ("Enter" por ejemplo) para continuar mostrando el listado, habiendo previamente puesto en cero la variable cuenta_lineas. Estas son las consignas, te queda la codificación a vos como ejercicio...

Realmente esto es todo. Abrís un archivo, leés el contenido y lo manipulás en la forma que vos quieras. Cuando hayas terminado, sólo debés cerrar el archivo y a otra cosa. Vamos a crear en Python nuestra propia versión del comando copy (copiar): simplemente abrimos un archivo en modo lectura, creamos uno nuevo en modo escritura y escribimos en este último las líneas del primero:

```
# Equivalente de COPY MENU.TXT MENU.BAK

# Abrimos los archivos para lectura (r) y escritura (w)
inp = open("menu.txt","r")
outp = open("menu.bak","w")

# leemos el archivo, copiamos el contenido en una lista
# y copiamos ésta en el nuevo archivo
for linea in inp.readlines():
    outp.write(linea)

print "1 archivo copiado..."

# Ahora cerramos los archivos
inp.close()
outp.close()
```

¿Te diste cuenta de que agregamos una instrucción print sólo para que el usuario viera que algo había ocurrido? Este tipo de *retroalimentación con el usuario* es siempre una buena idea.

Una última vuelta de tuerca sería agregar datos al final de una archivo existente. Una forma de realizar esto sería abrir el archivo en modo lectura, recuperar su contenido en una lista, agregar los nuevos datos en la lista y escribir la lista completa en una nueva versión del archivo anterior. Si el archivo es pequeño esta no es una mala idea, pero si el archivo es extenso (digamos unos 100Mb), nos quedaremos sin memoria para manejar la lista. Afortunadamente hay otra solución: el modo `"a"` nos permite agregar nuevos datos directamente al final del archivo. Mejor aun, si el archivo no existe, creará uno nuevo como si hubiésemos especificado `"w"`.

Como ejemplo supongamos que tenemos un archivo donde guardamos mensajes de error, y por supuesto no queremos borrar los mensajes que tenemos sino ir acumulándolos de manera secuencial:

```
def logError(msg):
   err = open("Errores.log","a")
   err.write(msg)
   err.close()
```

En la vida real, probablemente limitemos de algún modo el tamaño del archivo. Una técnica común es crear un nombre de archivo basado en la fecha, de tal modo que cuando cambia la fecha automáticamente creamos un nuevo archivo. Este procedimiento facilita mucho las tareas del administrador del sistema al agrupar los errores por fecha, pudiendo eliminar aquellos más antiguos que ya no son necesarios.

Contando palabras

Vamos a rever el programa que contaba palabras tal como prometí en la sección anterior. Nuestro *Pseudo Código* era algo así como:

```
def numpalabras(s):
    lista = split(s) # lista con cada una de las palabras
    return len(lista) # devuelve la cantidad de items en la lista

for linea in archivo:
    total = total + numpalabras(linea) # acumulamos el total de cada línea
print "El archivo tiene %d palabras" % total
```

Ahora que ya sabemos cómo obtener las líneas de un archivo, veamos en mayor detalle nuestra función `numpalabras()`. Primero queremos crear una lista con las palabras de una línea. Si revisamos la documentación de referencia de Python encontraremos el módulo `string` que nos aporta una función denominada `split`. Esta función separa una cadena de caracteres en una lista de campos delimitados por un espacio en blanco (o por cualquier otro caracter que nosotros hayamos definido previamente). Finalmente, gracias a la documentación encontramos una función predeterminada llamada `len()` la cual devuelve la cantidad de elementos contenidos en una lista. En nuestro caso, estos elementos serán las palabras.

Ahora el código final:

```
import string
def numpalabras(s):
    lista = string.split(s) # necesitamos prefijar a split() con el módulo string
    return len(lista) # devuelve el número de elementos en lista

inp = open("menu.txt","r")
total = 0  # inicializamos total en cero, también creamos la variable

for linea in inp.readlines():
    total = total + numpalabras(linea) # acumula el total de cada línea
print "El archivo tiene %d palabras" % total

inp.close()
```

Este programa tiene ciertas limitaciones, ya que si en el texto apareciera el símbolo "&" lo contaría como una palabra (algo no tan grave, ya que de hecho implica la conjunción "y"). También, sólo puede ser utilizado con un único archivo llamado `"menu.txt"`. Esto puede solucionarse si obtenemos el nombre del archivo desde la línea de comando (`argv[1]`) o por medio de la instrucción `raw_input()` que vimos en la sección 'Hablándole al usuario'. Dejo estas mejoras como un ejercicio para los lectores.

BASIC y Tcl

Tanto BASIC como Tcl presentan sus propios mecanismos para manejar archivos, los cuales no son muy diferentes del utilizado por Python. Por esta razón me limitaré a mostrarles un ejemplo.

La versión del BASIC

BASIC utiliza el concepto de *canales* para identificar a los archivos. Estos canales deben numerarse, lo que hace bastante tedioso el manejo de archivos en Basic. Podemos evitar esto utilizando la función `FREEFILE` que nos devuelve el siguiente número de canal disponible en el sistema. Si guardamos este dato en una variable no nos confundiremos acerca de la correspondencia entre estos canales y los archivos respectivos.

```
INFILE = FREEFILE
OPEN "TEST.DAT" FOR INPUT AS INFILE
REM Comprobamos final de archivo (EOF), luego
REM leemos la línea del archivo y la imprimimos
```

```
DO WHILE NOT EOF(INFILE)
    LINE INPUT #INFILE, LINEA
    PRINT LINEA
CLOSE #INFILE
```

La versión de Tcl

Ya te habrás acostumbrado al mecanismo:

```
set infile [open "Test.dat" r]
while { [gets $infile line] >= 0} {
     puts $line
     }
close $infile
```

Anterior Próxima Contenido

Si tenés sugerencias o dudas podés enviar un email en inglés a: alan.gauld@btinternet.com o en español a: manilio@zoommail.com