ARCHIVOS

Existen dos formas básicas de acceder a un archivo, una es utilizarlo como un archivo de texto, que procesaremos línea por línea; la otra es tratarlo como un archivo binario, que procesaremos byte por byte.

En Python, para abrir un archivo usaremos la función open, que recibe el nombre del archivo a abrir.

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

Esta función intentará abrir el archivo con el nombre indicado. Si tiene éxito, devolverá una variable que nos permitirá manipular el archivo de diversas maneras.

La operación más sencilla a realizar sobre un archivo es leer su contenido. Para procesarlo línea por línea, es posible hacerlo de la siguiente forma:

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

Esto funciona ya que cada archivo que se encuentre abierto tiene una posición asociada, que indica el último punto que fue leído.

Cada vez que se lee una línea, avanza esa posición. Es por ello que `readline()` devuelve cada vez una línea distinta y no siempre la misma.

La siguiente estructura es una forma equivalente a la vista en el ejemplo anterior.

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

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

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

```
líneas = archivo.readlines()
```

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

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

Para cerrar un archivo simplemente se debe llamar a:

```
archivo.close()
```

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

```
# Código: numera_líneas.py: Imprime las lineas de un
archivo con su numero
archivo = open("archivo.txt")
i = 1
for linea in archivo:
    linea = linea.rstrip("\\n")
    print " %4d: %s" % (i, linea)
    i+=1
archivo.close()
```

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

NOTA
Los archivos de texto son sencillos de manejar, pero existen por lo menos tres formas distintas de marcar un fin de línea.

En Unix tradicionalmente se usa el caracter `\n` (valor de ASCII 10, definido como nueva línea) para el fin de línea, mientras que en Macintosh el fin de línea se solía representar como un `\r` (valor ASCII 13, definido como retorno de carro) y en Windows se usan ambos caracteres `\r\n`.

Si bien esto es algo que hay que tener en cuenta en una diversidad de casos, en particular en Python por omisión se maneja cualquier tipo de fin de línea como si fuese un `\n`, salvo que se le pida lo contrario. Para manejar los caracteres de fin de línea *a mano* se puede poner una `U` en el parámetro modo que le pasamos a `open`.

Otra opción para hacer exactamente lo mismo seria utilizar la función de Python `enumerate(secuencia). Esta función devuelve un contador por cada uno de los elementos que se recorren, puede usarse con cualquier tipo de secuencia, incluyendo archivos. La versión equivalente se muestra a continuación.

```
# Código: numera_líneas2.py: Imprime las líneas de
un archivo con su número

archivo = open("archivo.txt")
for i, línea in enumerate(archivo):
    línea = línea.rstrip("\\n")
    print " %4d: %s" % (i, línea)
archivo.close()
```

Modo de apertura de los archivos

La función `open` recibe un parámetro opcional para indicar el modo en que se abrirá el archivo. Los tres modos de apertura que se pueden especificar son:

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

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

NOTA
Si un archivo no existe y se lo intenta abrir en modo lectura, se generará un error; en cambio si se lo abre para escritura, Python se encargará de crear el archivo al momento de abrirlo, ya sea con `w`, `a`, `w+` o con `a+`).
En caso de que no se especifique el modo, los archivos serán abiertos en modo sólo lectura (`r`).

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

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

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

O mediante listas de cadenas:

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

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

```
# Código: genera_saludo.py: Genera el archivo saludo.py
saludo = open("saludo.py", "w")
saludo.write("""
print "Hola Mundo"
""")
saludo.close()
```

El ejemplo que se muestra contiene un programa Python que a su vez genera el código de otro programa Python.

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

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

Agregar información a un archivo

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

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

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

En este módulo se utiliza el módulo de Python `datetime` para obtener la fecha y hora actual que se guardará en los archivos. Es importante notar que en el módulo mostrado no se abre o cierra un archivo en particular, sino que las funciones están programadas de modo tal que puedan ser utilizadas desde otro programa.
Se trata de un módulo genérico que podrá ser utilizado por diversos programas, que requieran la funcionalidad de registrar los posibles errores o eventos que se produzcan durante la ejecución.
Para utilizar este módulo, será necesario primero llamar a `abrir_log` para abrir el archivo de log, luego llamar a `guardar_log` por cada mensaje que se quiera registrar, y finalmente llamar a `cerrar_log` cuando se quiera concluir la registración de mensajes.
Por ejemplo, en el Código 11.5 se muestra un posible programa que utiliza el módulo de log incluido anteriormente.

```
# Código: log.py: Módulo para manipulación de archivos de log
#! /usr/bin/env python
# encoding: latin1

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

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

def guardar_log(archivo_log, mensaje):
```

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

def cerrar_log(archivo_log):
    """ Cierra el archivo de log.
    Pre: el archivo de log ha sido abierto correctamente.
    Post: el archivo de log se ha cerrado. """
    guardar_log(archivo_log, "Fin del registro de errores")
    archivo_log.close()
# Código 11.5: usa_log.py: Módulo que utiliza el módulo de
log
#!/usr/bin/env python
#encoding latin1

import log

# Constante que contiene el nombre del archivo de log a
utilizar
ARCHIVO_LOG = "mi_log.txt"

def main():
    # Al comenzar, abrir el log
    archivo_log = log.abrir_log(ARCHIVO_LOG)
    # ...
    # Hacer cosas que pueden dar error
    if error:
        guardar_log(archivo_log, "ERROR: "+error)
    # ...
    # Finalmente cerrar el archivo
    log.cerrar_log(archivo_log)

main()
```

Este código, que incluye el módulo `log` mostrado anteriormente, muestra una forma básica de utilizar un archivo de log. Al iniciarse el programa se abre el archivo de log, de forma que queda registrada la fecha y hora de inicio. Posteriormente se realizan tareas varias que podrían provocar errores, y de haber algún error se lo guarda en el archivo de log. Finalmente, al terminar el programa, se cierra el archivo de log, quedando registrada la fecha y hora de finalización.

El archivo de log generado tendrá la forma:

```
2015-04-10 15:20:32.229556 Iniciando registro de errores
2015-04-10 15:20:50.721415 ERROR: no se pudo acceder al
recurso
2015-04-10 15:21:58.625432 ERROR: formato de entrada inválido
2015-04-10 15:22:10.109376 Fin del registro de errores
```

Manipular un archivo en forma binaria

No todos los archivos son archivos de texto, y por lo tanto no todos los archivos pueden ser procesados por líneas. Existen archivos en los que cada byte tiene un significado particular, y es necesario manipularlos conociendo el formato en que están los datos para poder procesar esa información.
Para abrir un archivo y manejarlo de forma binaria es necesario agregarle una `b` al parámetro de modo.
Para procesar el archivo de a bytes en lugar de líneas, se utiliza la función `contenido = archivo.read(n)` para leer `n` bytes y `archivo.write(contenido)`, para escribir contenido en la posición actual del archivo.

NOTA

La `b` en el modo de apertura viene de *binario*, por el sistema de numeración binaria, ya que en el procesador de la computadora la información es manejada únicamente mediante ceros o unos (bits) que conforman números binarios.
Si bien no es necesaria en todos los sistemas (en general el mismo sistema detecta que es un archivo binario sin que se lo pidamos), es una buena

costumbre usarla, por más que sirva principalmente como documentación.
Al manejar un archivo binario, es necesario poder conocer la posición actual en el archivo y poder modificarla. Para obtener la posición actual se utiliza `archivo.tell()`, que indica la cantidad de bytes desde el comienzo del archivo.
Para modificar la posición actual se utiliza `archivo.seek(inicio, desde)` , que permite desplazarse una cantidad `inicio` de bytes en el archivo, contando `desde` el comienzo del archivo, `desde` la posición actual o `desde` el final.