7 Archivos

Persistencia

Hasta ahora, hemos aprendido cómo escribir programas y comunicar nuestras intenciones a la **Unidad de procesamiento central** mediante la ejecución condicional, las funciones y las iteraciones. Hemos aprendido cómo crear y usar estructuras de datos en la **Memoria principal**. La CPU y la memoria son donde nuestro software funciona y se ejecuta. Es donde sucede todo el "pensamiento".

Pero si recuerdas de nuestras discusiones sobre la arquitectura del hardware, una vez que se apaga la alimentación, cualquier cosa almacenada en la CPU o la memoria principal se borra. Hasta ahora, nuestros programas solo han sido ejercicios transitorios y divertidos para aprender Python.

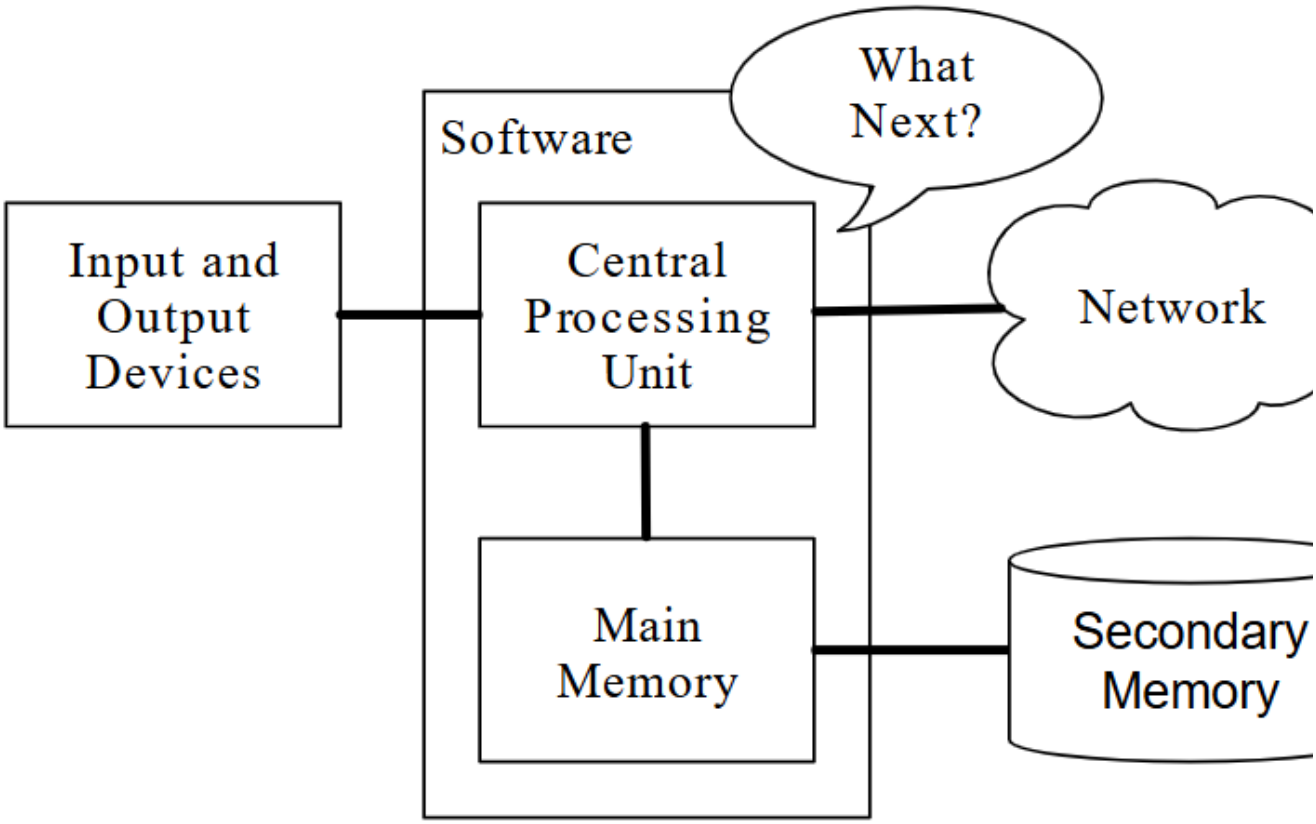

Imagen - Memoria secundaria

En este capítulo, comenzamos a trabajar con **Memoria secundaria** (o archivos). La memoria secundaria no se borra cuando se apaga la alimentación. O en el caso de una unidad flash USB, los datos que escribimos de nuestros programas se pueden eliminar del sistema y transportar a otro sistema.

Nos centraremos principalmente en leer y escribir archivos de texto como los que creamos en un editor de texto. Más adelante veremos cómo trabajar con archivos de base de datos que son archivos binarios, diseñados específicamente para ser leídos y escritos a través del software de base de datos.

Apertura de archivos

Cuando queremos leer o escribir un archivo (digamos en su disco duro), primero debemos **abrir** el archivo. Al abrir el archivo se comunica con su sistema operativo, que sabe dónde se almacenan los datos de cada archivo. Cuando abres un archivo, le estás pidiendo al sistema operativo que busque el archivo por su nombre y se asegure de que exista. En este ejemplo, abrimos el archivo `mbox.txt`, que se debe almacenar en la misma carpeta en la que se encuentra cuando inicia Python. Puedes descargar este archivo desde www.py4e.com/code3/mbox.txt

```
>>> fhand = open('mbox.txt')
>>> print(fhand)
<_io.TextIOWrapper name='mbox.txt' mode='r' encoding='cp1252'>
```

Si el `open` tiene éxito, el sistema operativo nos devuelve un **archivo manejador**. El identificador de archivo no es la información real contenida en el archivo, sino que es un "identificador" que podemos usar para leer los datos. Se le otorga un identificador si el archivo solicitado existe y tienes los permisos adecuados para leer el archivo.

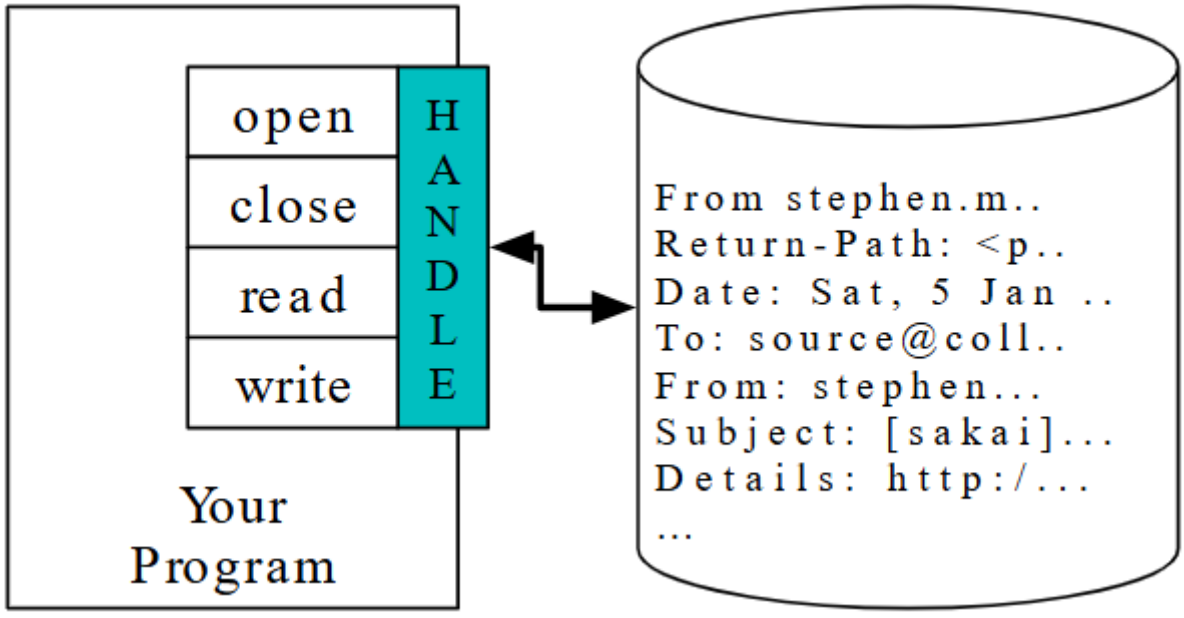

Imagen - A File Handle

Si el archivo no existe, `open` fallará con un rastreo y no obtendrás un identificador para acceder al contenido del archivo:

```
>>> fhand = open('stuff.txt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'stuff.txt'
```

Más adelante usaremos `try` y `except` para lidiar con la situación en la que intentamos abrir un archivo que no existe.

Archivos y líneas de texto

Un archivo de texto puede considerarse como una secuencia de líneas, al igual que una cadena de Python puede considerarse como una secuencia de caracteres. Por ejemplo, esta es una muestra de un archivo de texto que registra la actividad de correo de varias personas en un equipo de desarrollo de proyectos de código abierto:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
Date: Sat, 5 Jan 2008 09:12:18 -0500
To: source@collab.sakaiproject.org
From: stephen.marquard@uct.ac.za
Subject: [sakai] svn commit: r39772 - content/branches/
Details: http://source.sakaiproject.org/viewsvn/?view=rev&amp;rev=39772
...
```

El archivo completo de interacciones de correo está disponible en

www.py4e.com/code3/mbox.txt

y una versión abreviada del archivo está disponible en

www.py4e.com/code3/mbox-short.txt

Estos archivos están en un formato estándar para un archivo que contiene varios mensajes de correo. Las líneas que comienzan con "From" separan los mensajes y las líneas que comienzan con "From:" son parte de los mensajes. Para obtener más información sobre el formato mbox, consulte en.wikipedia.org/wiki/Mbox.

Para dividir el archivo en líneas, hay un carácter especial que representa el "final de la línea" llamado el carácter **nueva línea**.

En Python, representamos el carácter **nueva línea** como una barra invertida-n en constantes de cadena. A pesar de que se ve como dos caracteres, en realidad es un solo caracter. Cuando miramos la variable al ingresar "cosas" en el intérprete, nos muestra el `\n` en la cadena, pero cuando usamos `print` para mostrar la cadena, vemos la cadena dividida en dos líneas por el caracter de **nueva línea**.

```
>>> stuff = 'Hello\nWorld!'
>>> stuff
'Hello\nWorld!'
>>> print(stuff)
Hello
World!
>>> stuff = 'X\nY'
>>> print(stuff)
X
Y
>>> len(stuff)
3
```

También puede ver que la longitud de la cadena `X\nY` es **tres** caracteres porque el carácter de nueva línea es un solo carácter.

Entonces, cuando miramos las líneas en un archivo, necesitamos **imaginarnos** que hay un carácter invisible especial llamado `nueva línea` que marca el final de la línea.

Así que el caracter de nueva línea separa los caracteres del archivo en líneas.

Leyendo archivos

Mientras que el **manejador de archivo** no contiene los datos para el archivo, es bastante fácil construir un bucle `for` para leer y contar cada una de las líneas en un archivo:

https://trinket.io/embed/python3/971f4a72f3

Podemos usar el identificador de archivo como el iterable en nuestro bucle `for`. Nuestro bucle `for` simplemente cuenta el número de líneas en el archivo y las imprime. La traducción aproximada del bucle `for` al inglés es, "para cada línea en el archivo representado por el identificador de archivo, incrementa la variable `count`.

La razón por la que la función `abrir` no lee el archivo completo es que el archivo puede ser bastante grande con muchos gigabytes de datos. La instrucción `open` toma la misma cantidad de tiempo independientemente del tamaño del archivo. El bucle `for` en realidad hace que los datos se lean del archivo.

Cuando se lee el archivo utilizando un bucle `for` de esta manera, Python se encarga de dividir los datos en el archivo en líneas separadas utilizando el carácter de nueva línea. Python lee cada línea asignándola a la variable `line` en cada iteración del bucle `for`.

Debido a que el bucle `for` lee los datos una línea cada vez, puede leer y contar de manera eficiente las líneas en archivos muy grandes sin quedarse sin memoria principal para almacenar los datos. El programa anterior puede contar las líneas en cualquier archivo de tamaño usando muy poca memoria ya que cada línea se lee, se cuenta y luego se descarta.

Si sabes que el archivo es relativamente pequeño en comparación con el tamaño de tu memoria principal, puedes leer todo el archivo en una cadena usando el método `read` en el identificador de archivo.

```
>>> fhand = open('mbox-short.txt')
>>> inp = fhand.read()
>>> print(len(inp))
94626
>>> print(inp[:20])
From stephen.marquar
```

En este ejemplo, el contenido completo (todos los 94626 caracteres) del archivo `mbox-short.txt` se lee directamente en la variable `inp`. Utilizamos el corte de cadena para imprimir los primeros 20 caracteres de los datos de cadena almacenados en `inp`.

Cuando el archivo se lee de esta manera, todos los caracteres, incluidas todas las líneas y los caracteres de nueva línea, son una cadena grande en la variable **inp**. Recuerda que la función `read` solo debe usarse si los datos del archivo caben cómodamente en la memoria principal de su ordenador.

Si el archivo es demasiado grande para la memoria principal, debes escribir tu programa para leer el archivo en trozos usando un bucle `for` o `while`.

Buscando a través de un archivo

Cuando buscas datos en un archivo, es un patrón muy común leer un archivo, ignorando la mayoría de las líneas y solo procesando líneas que cumplen con una condición particular. Podemos combinar el patrón para leer un archivo con métodos de cadena para construir mecanismos de búsqueda simples.

Por ejemplo, si quisiéramos leer un archivo y solo imprimir líneas que comenzaran con el prefijo "De:", podríamos usar el método de cadena **startswith** para seleccionar solo aquellas líneas con el prefijo deseado:

https://trinket.io/embed/python3/064568c5b1

Cuando este programa se ejecuta, obtenemos el siguiente resultado:

```
From: stephen.marquard@uct.ac.za

From: louis@media.berkeley.edu

From: zqian@umich.edu

From: rjlowe@iupui.edu
...
```

El resultado se ve muy bien ya que las únicas líneas que estamos viendo son aquellas que comienzan con "De:", pero ¿por qué vemos las líneas en blanco adicionales? Esto se debe a ese carácter invisible **de nueva línea**. Cada una de las líneas termina con una nueva línea, por lo que la instrucción `print` imprime la cadena en la línea de la variable que incluye una nueva línea y luego `print` agrega **otra** nueva línea, resultando en el efecto de doble espaciado.

Podríamos usar el corte de línea para imprimir todo menos el último carácter, pero un enfoque más simple es usar el método **rstrip** que elimina el espacio en blanco del lado derecho de una cadena de la siguiente manera:

https://trinket.io/embed/python3/8436ab534f

Cuando este programa se ejecuta, obtenemos el siguiente resultado:

```
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: cwen@iupui.edu
...
```

A medida que los programas de procesamiento de archivos se vuelven más complicados, es posible que desees estructurar tus bucles de búsqueda utilizando `continue`. La idea básica del bucle de búsqueda es que estás buscando líneas "interesantes" y que omites líneas "no interesantes". Cuando encontramos una línea interesante, hacemos algo con esa línea.

Podemos estructurar el bucle para seguir el patrón de omitir líneas no interesantes como sigue:

https://trinket.io/embed/python3/c175b5a15f

La salida del programa es la misma. En inglés, las líneas no interesantes son aquellas que no comienzan con "From:", que omitimos usando `continue`. Para las líneas "interesantes" (es decir, aquellas que comienzan con "From:") realizamos el procesamiento pertinente.

Podemos usar el método de cadena `find` para simular una búsqueda de editor de texto que encuentra líneas donde la cadena de búsqueda está en cualquier lugar de la línea. Dado que `find` busca una aparición de una cadena dentro de otra cadena y devuelve la posición de la cadena o -1 si no se encontró la cadena, podemos escribir el siguiente bucle para mostrar las líneas que contienen la cadenav "@uct.ac.za" (es decir, provienen de la Universidad de Ciudad del Cabo en Sudáfrica):

https://trinket.io/embed/python3/25e3e59816

Que produce el siguiente resultado:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
X-Authentication-Warning: set sender to stephen.marquard@uct.ac.za using -f
From: stephen.marquard@uct.ac.za
Author: stephen.marquard@uct.ac.za
From david.horwitz@uct.ac.za Fri Jan  4 07:02:32 2008
X-Authentication-Warning: set sender to david.horwitz@uct.ac.za using -f
From: david.horwitz@uct.ac.za
Author: david.horwitz@uct.ac.za
...
```

Permitiendo que el usuario elija el nombre del archivo

Realmente no queremos tener que editar nuestro código Python cada vez que deseamos procesar un archivo diferente. Sería más útil pedirle al usuario que ingrese la cadena del nombre del archivo cada vez que se ejecute el programa para que puedan usar nuestro programa en diferentes archivos sin cambiar el código Python.

Esto es bastante simple de hacer leyendo el nombre del archivo del usuario usando `input` de la siguiente manera:

https://trinket.io/embed/python3/e92683e754

Leemos el nombre del archivo del usuario, lo colocamos en una variable llamada `fname` y abrimos ese archivo. Ahora podemos ejecutar el programa repetidamente en diferentes archivos.

```
python search6.py
Enter the file name: mbox.txt
```

```
There were 1797 subject lines in mbox.txt

python search6.py
Enter the file name: mbox-short.txt
There were 27 subject lines in mbox-short.txt
```

Antes de echar un vistazo a la siguiente sección, observa el programa anterior y pregúntate: "¿Qué podría salir mal aquí?" o "¿Qué podría hacer nuestro usuario amigable que haría que nuestro pequeño y agradable programa fallara en su ejecución, mostrándonos no tan geniales a los ojos de nuestros usuarios?"

Usando `try, except,` y `open`

Te dije que no miraras. Esta es tu última oportunidad.

¿Qué pasa si nuestro usuario escribe algo que no es un nombre de archivo?

```
python search6.py
Enter the file name: missing.txt
Traceback (most recent call last):
  File "search6.py", line 2, in <module>
    fhand = open(fname)
FileNotFoundError: [Errno 2] No such file or directory: 'missing.txt'

python search6.py
Enter the file name: na na boo boo
Traceback (most recent call last):
  File "search6.py", line 2, in <module>
    fhand = open(fname)
FileNotFoundError: [Errno 2] No such file or directory: 'na na boo boo'
```

No te rías. Los usuarios eventualmente harán todo lo posible para romper tus programas. De hecho, una parte importante de cualquier equipo de desarrollo de software es una persona o grupo llamado **Aseguradora de calidad** (o control de calidad para abreviar) cuyo trabajo consiste en hacer las cosas más locas posibles en un intento de romper el software que el programador ha creado.

El equipo de control de calidad es responsable de encontrar las fallas en los programas antes de que entreguemos el programa a los usuarios finales que pueden estar comprando el software o pagando nuestro salario para escribir el software. Así que el equipo de control de calidad es el mejor amigo del programador.

Así que ahora que vemos la falla en el programa, podemos arreglarlo con elegancia usando la estructura `try`/`except`. Debemos suponer que la llamada `open` puede fallar y agregar un código de recuperación cuando el `open` falla de la siguiente manera:

https://trinket.io/embed/python3/ee7849d86b

La función `exit` termina el programa. Es una función que llamamos que nunca vuelve. Ahora, cuando nuestro usuario (o equipo de control de calidad) escribe tonterías o nombres de archivos incorrectos, los "capturamos" y recuperamos con gracia:

```
python search7.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

python search7.py
Enter the file name: na na boo boo
File cannot be opened: na na boo boo
```

Proteger la llamada `open` es un buen ejemplo del uso correcto de `try` y `except` en un programa Python. Usamos el término "Pythonic" cuando estamos haciendo algo a la "manera Python". Podríamos decir que el ejemplo anterior es la forma Pythonic de abrir un archivo.

Una vez que adquieras más experiencia en Python, puede comprometerse con otros programadores de Python para decidir cuál de las dos soluciones equivalentes a un problema es "más Pythonic". El objetivo de ser "más Pythonic" capta la noción de que la programación es parte ingeniería y parte arte. No siempre estamos interesados en hacer que algo funcione, también queremos que nuestra solución sea elegante y que nuestros pares la aprecien como elegante.

Escribiendo archivos

Para escribir un archivo, debes abrirlo con el modo "w" (write) como segundo parámetro:

```
>>> fout = open('output.txt', 'w')
>>> print(fout)
<_io.TextIOWrapper name='output.txt' mode='w' encoding='cp1252'>
```

Si el archivo ya existe, abrirlo en modo de escritura borra los datos antiguos y comienza de nuevo, ¡así que ten cuidado! Si el archivo no existe, se crea uno nuevo.

El método `write` del objeto de identificador de archivo coloca datos en el archivo, devolviendo el número de caracteres escritos. El modo de escritura predeterminado es texto para escribir (y leer) cadenas.

```
>>> line1 = "This here's the wattle,\n"
>>> fout.write(line1)
24
```

Nuevamente, el objeto de archivo mantiene un registro de dónde está, por lo que si llama a `write` de nuevo, agrega los nuevos datos al final.

Debemos asegurarnos de administrar los extremos de las líneas a medida que escribimos en el archivo insertando explícitamente el carácter de nueva línea cuando queremos terminar una línea. La instrucción `print` agrega automáticamente una nueva línea, pero el método `write` no agrega la nueva línea automáticamente.

```
>>> line2 = 'the emblem of our land.\n'
>>> fout.write(line2)
24
```

Cuando hayas terminado de escribir, debes cerrar el archivo para asegurarte de que el último bit de datos se haya escrito físicamente en el disco y que no se pierda si se corta la alimentación.

```
>>> fout.close()
```

Podríamos cerrar los archivos que abrimos para leer también, pero podemos ser un poco descuidados si solo abrimos algunos archivos ya que Python se asegura de que todos los archivos abiertos se cierren cuando finalice el programa. Cuando estamos escribiendo archivos, queremos cerrar explícitamente los archivos para no dejar nada al azar.

depuración

Cuando estás leyendo y escribiendo archivos, puedes tener problemas con el espacio en blanco. Estos errores pueden ser difíciles de depurar porque los espacios, las pestañas y las nuevas líneas son normalmente invisibles:

```
>>> s = '1 2\t 3\n 4'
>>> print(s)
1 2  3
 4
```

La función incorporada `repr` puede ayudar. Toma cualquier objeto como un argumento y devuelve una representación de cadena del objeto. Para cadenas, representa caracteres de espacio en blanco con secuencias de barra invertida:

```
>>> print(repr(s))
'1 2\t 3\n 4'
```

Esto puede ser útil para la depuración.

Otro problema que podrías encontrar es que los diferentes sistemas usan caracteres diferentes para indicar el final de una línea. Algunos sistemas usan `\n`. Otros utilizan un carácter de retorno, representado como `\r`. Algunos usan ambos. Si mueves archivos entre diferentes sistemas, estas inconsistencias pueden causar problemas.

Para la mayoría de los sistemas, hay aplicaciones para convertir de un formato a otro. Puede encontrarlos (y leer más sobre este problema) en wikipedia.org/wiki/Newline. O, por supuesto, podrías escribir uno tú mismo.

Ejercicios

Ejercicio 1 escribe un programa para leer un archivo e imprime el contenido del mismo (línea por línea) todo en mayúsculas. Ejecutando el programa se verá como sigue:

```
python shout.py
Enter a file name: mbox-short.txt
FROM STEPHEN.MARQUARD@UCT.AC.ZA SAT JAN  5 09:14:16 2008
RETURN-PATH: <POSTMASTER@COLLAB.SAKAIPROJECT.ORG>
RECEIVED: FROM MURDER (MAIL.UMICH.EDU [141.211.14.90])
     BY FRANKENSTEIN.MAIL.UMICH.EDU (CYRUS V2.3.8) WITH LMTPA;
```

```
      SAT, 05 JAN 2008 09:14:16 -0500
```

Puedes descargar el archivo desde

www.py4e.com/code3/mbox-short.txt

Ejercicio 2 escribe un programa para solicitar un nombre de archivo. Luego lee el archivo y busca las líneas del formulario:

`X-DSPAM-Confidence: 0.8475`

Cuando encuentra una línea que comienza con "X-DSPAM-Confidence:", separa la línea para extraer el número de punto flotante en la línea. Cuenta estas líneas y luego calcula el total de los valores de confianza de correo no deseado de estas líneas. Cuando llegues al final del archivo, imprime la confianza promedio del spam.

```
Enter the file name: mbox.txt
Average spam confidence: 0.894128046745

Enter the file name: mbox-short.txt
Average spam confidence: 0.750718518519
```

Prueba tu programa en los archivos `mbox.txt` y `mbox-short.txt`.

Ejercicio 3 A veces, cuando los programadores se aburren y quieren divertirse un poco, agregan un **Huevo de Pascua** inofensivo a su programa. Modifica el programa que solicita al usuario el nombre del archivo para que imprima un mensaje divertido cuando el el usuario escriba el nombre exacto del archivo "na na boo boo". El programa debería comportarse normalmente para todos los demás archivos que existen y no existen. Aquí hay una muestra de ejecución del programa:

```
python egg.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

python egg.py
Enter the file name: missing.tyxt
File cannot be opened: missing.tyxt

python egg.py
```

```
Enter the file name: na na boo boo
NA NA BOO BOO TO YOU - You have been punk'd!
```

No te alentamos a que pongas Huevos de Pascua en tus programas; esto es solo un ejercicio

---

Revision #1
Created 2025-04-05 09:30:45 CEST by Javier Quintana
Updated 2025-04-05 09:33:50 CEST by Javier Quintana