Programación en Python
TEMA 6
Ficheros de texto

Universidad Politécnica de Valencia

2021-2022

Índice

1 Persistencia 3

2 Abrir ficheros 3

3 Ficheros de texto y líneas 4

4 Lectura de ficheros 5

5 Búsqueda a través de un fichero 6

6 Permitiendo al usuario elegir el nombre de fichero 8

7 Utilizando try, except, y open 8

8 Escritura de ficheros 10

9 Depuración 10

10 Glosario 11

Ejercicios 11

1. Persistencia

Hasta ahora, hemos aprendido cómo escribir programas y comunicar nuestras intenciones a la *Unidad Central de Procesamiento* utilizando ejecuciones condicionales, funciones, e iteraciones. Hemos aprendido como crear y usar estructuras de datos en la *Memoria Principal*. La CPU y la memoria son los lugares donde nuestro software funciona y se ejecuta. Es donde toda la *inteligencia* ocurre.

Pero si recuerdas nuestras discusiones de arquitectura de hardware, una vez que la corriente se interrumpe, cualquier cosa almacenada ya sea en la CPU o en la memoria es eliminada. Así que hasta ahora nuestros programas han sido sólo una diversión pasajera para aprender Python.

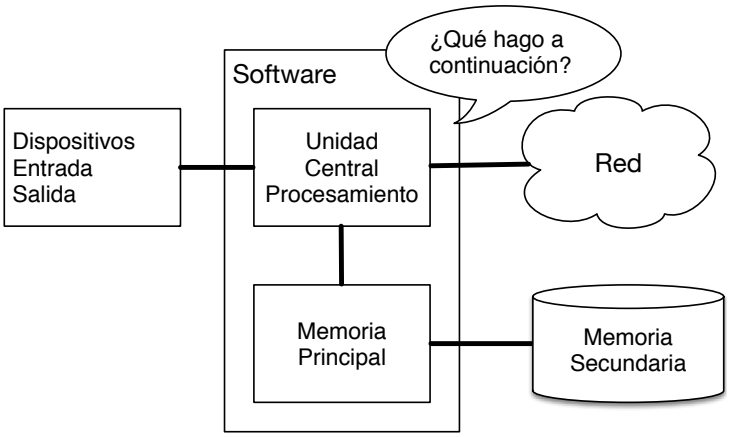

Figura 1: Memoria Secundaria

En este capítulo, vamos a comenzar a trabajar con *Memoria Secundaria* (o ficheros). La memoria secundaria no es eliminada cuando apagamos una computadora. Incluso, en el caso de una memoria USB, los datos que escribimos desde nuestros programas pueden ser retirados del sistema y transportados a otro sistema.

Nos vamos a enfocar principalmente en leer y escribir ficheros como los que creamos en un editor de texto. Más adelante veremos cómo trabajar con ficheros de bases de datos, que son ficheros binarios diseñados específicamente para ser leídos y escritos a través de software para manejo de bases de datos.

2. Abrir ficheros

Cuando queremos abrir o escribir un fichero (digamos, en el disco duro), primero debemos *abrir* el fichero. Al abrir el fichero nos comunicamos con el sistema operativo, el cual sabe dónde están almacenados los datos de cada fichero. Cuando abres un fichero, le estás pidiendo al sistema operativo que encuentre el fichero por su nombre y se asegure de que existe. En este ejemplo, abrimos el fichero *mbox.txt*, el cual debería estar almacenado en el mismo directorio en que estás localizado cuando inicias Python. Puedes descargar este fichero en Poliformat.

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

Si el **open** es exitoso, el sistema operativo nos devuelve un *manejador de fichero*. El manejador de fichero no son los datos contenidos en el fichero, sino un "manejador" *(handler)* que podemos usar para leer los datos. Obtendrás un manejador de fichero si el fichero solicitado existe y si tienes los permisos apropiados para leerlo.

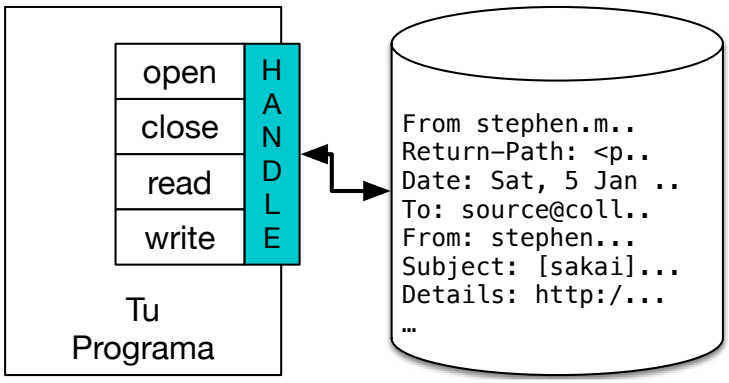

Figura 2: Un Manejador de fichero

Si el fichero no existe, **open** fallará con un mensaje de error y no obtendrás un manejador para acceder al contenido del fichero:

```
>>> manejador_fichero = 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 vamos a utilizar **try** y **except** para controlar de mejor manera la situación donde tratamos de abrir un fichero que no existe.

3. Ficheros de texto y líneas

Un fichero de texto puede ser considerado como una secuencia de líneas, así como una cadena de Python puede ser considerada como una secuencia de caracteres. Por ejemplo, este es un ejemplo de un fichero de texto que registra la actividad de correos de varias personas en un equipo de desarrollo de un proyecto de código abierto (open source):

```
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&rev=39772
...
```

El fichero completo de interacciones por correo está disponible en Poliformat con nombre `mbox.txt` y una versión reducida del fichero está disponible en `mbox-short.txt`.

Esos ficheros están en un formato estándar para un fichero que contiene múltiples mensajes de correo. Las líneas que comienzan con “From” separan los mensajes y las líneas que comienzan con “From:” son parte de esos mensajes. Para más información acerca del formato mbox, consulta https://es.wikipedia.org/wiki/Mbox.

Para separar el fichero en líneas, hay un carácter especial que representa el “final de una línea” llamado *salto de línea*.

En Python, representamos el *salto de línea* como una barra invertida-n en las cadenas. Incluso aunque esto parezca dos caracteres, realmente es un solo carácter. Cuando vemos la variable interactuando con el

intérprete, este nos muestra el `\n` en la cadena, pero cuando usamos **`print`** para mostrar la cadena, vemos la cadena separada en dos líneas debido al salto de línea.

```
>>> cosa = 'Hola\nMundo!'
>>> cosa
'Hola\nMundo!'
>>> print(cosa)
Hola
Mundo!
>>> cosa = 'X\nY'
>>> print(cosa)
X
Y
>>> len(cosa)
3
```

También puedes ver que el tamaño de la cadena `X\nY` es *tres* caracteres debido a que el separador de línea es un solo carácter.

Por tanto, cuando vemos las líneas en un fichero, necesitamos *imaginar* que ahí hay un carácter invisible llamado separador de línea al final de cada línea, el cual marca el final de la misma.

De modo que el separador de línea separa los caracteres del fichero en líneas.

4. Lectura de ficheros

Aunque el *manejador de fichero* no contiene los datos de un fichero, es bastante fácil utilizarlo en un bucle **`for`**—**`in`** para leer a través del fichero y contar cada una de sus líneas:

```
manejador_fichero = open('mbox-short.txt')
contador = 0
for linea in manejador_fichero:
    contador = contador + 1
print('Contador de líneas:', contador)
```

Podemos usar el manejador de ficheros como una secuencia en nuestro bucle **`for`**. Nuestro bucle **`for`** simplemente cuenta el número de líneas en el fichero y las imprime. La traducción aproximada de ese bucle al español es, "para cada línea en el fichero representado por el manejador de fichero, suma uno a la variable `count`."

La razón por la cual la función **`open`** no lee el fichero completo es porque el fichero puede ser muy grande, incluso con muchos gigabytes de datos. La sentencia **`open`** emplea la misma cantidad de tiempo sin importar el tamaño del fichero. De hecho, es el bucle **`for`** el que hace que los datos sean leídos desde el fichero.

Cuando el fichero es leído usando un bucle **`for`** de esta manera, Python se encarga de dividir los datos del fichero en líneas separadas utilizando el separador de línea. Python lee cada línea hasta el separador e incluye el separador como el último carácter en la variable `linea` para cada iteración del bucle **`for`**.

Debido a que el bucle **`for`** lee los datos línea a línea, éste puede leer eficientemente y contar las líneas en ficheros muy grandes sin quedarse sin memoria principal para almacenar los datos. El programa previo puede contar las líneas de cualquier tamaño de fichero utilizando poca memoria, puesto que cada línea es leída, contada, y después descartada.

Si sabes que el fichero es relativamente pequeño comparado al tamaño de tu memoria principal, puedes leer el fichero completo en una sola cadena utilizando el método `read` en el manejador de ficheros.

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

En este ejemplo, el contenido completo (todos los 94626 caracteres) del fichero *mbox-short.txt* son leídos directamente en la variable `inp`. Utilizamos el troceado de cadenas para imprimir los primeros 20 caracteres de la cadena de datos almacenada en `inp`.

Cuando el fichero es leído de esta forma, todos los caracteres incluyendo los saltos de línea son una cadena gigante en la variable `inp`. Es una buena idea almacenar la salida de `read` como una variable porque cada llamada a `read` vacía el contenido por completo:

```
>>> manejador = open('mbox-short.txt')
>>> print(len(manejador.read()))
94626
>>> print(len(manejador.read()))
0
```

Recuerda que esta forma de la función **`open`** solo debe ser utilizada si los datos del fichero son apropiados para la memoria principal del sistema. Si el fichero es muy grande para caber en la memoria principal, deberías escribir tu programa para leer el fichero en bloques utilizando un bucle **`for`** o **`while`**.

5. Búsqueda a través de un fichero

Cuando buscas a través de los datos de un fichero, un patrón muy común es leer el fichero, ignorar la mayoría de las líneas y solamente procesar líneas que cumplan con una condición particular. Podemos combinar el patrón de leer un fichero con métodos de cadenas para construir mecanismos de búsqueda sencillos.

Por ejemplo, si queremos leer un fichero y solamente imprimir las líneas que comienzan con el prefijo "From:", podríamos usar el método de cadenas *startswith* para seleccionar solo aquellas líneas con el prefijo deseado:

```
man_f = open('mbox-short.txt')
contador = 0
for linea in man_f:
    if linea.startswith('From:'):
        print(linea)
```

Cuando este programa se ejecuta, obtenemos la siguiente salida:

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

From: louis@media.berkeley.edu

From: zqian@umich.edu

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

La salida parece correcta puesto que las líneas que estamos buscando son aquellas que comienzan con "From:", pero ¿por qué estamos viendo las líneas vacías extras? Esto es debido al carácter invisible *salto de línea*. Cada una de las líneas leídas termina con un salto de línea, así que la sentencia `print` imprime la cadena almacenada en la variable `linea`, la cual incluye ese salto de línea, y después `print` agrega *otro* salto de línea, resultando en el efecto de doble salto de línea que observamos.

Podemos usar troceado de líneas para imprimir todos los caracteres excepto el último, pero una forma más sencilla es usar el método `rstrip`, el cual elimina los espacios en blanco del lado derecho de una cadena, tal como:

```
man_f = open('mbox-short.txt')
for linea in man_f:
    linea = linea.rstrip()
    if linea.startswith('From:'):
        print(linea)
```

Cuando este programa se ejecuta, obtenemos lo siguiente:

```
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 tus programas de procesamiento de ficheros se vuelven más complicados, quizá quieras estructurar tus bucles de búsqueda utilizando `continue`. La idea básica de un bucle de búsqueda es que estás buscando líneas "interesantes" e ignorando líneas "no interesantes". Y cuando encontramos una línea interesante, hacemos algo con ella.

Podemos estructurar el bucle para seguir el patrón de ignorar las líneas no interesantes así:

```
man_f = open('mbox-short.txt')
for linea in man_f:
    linea = linea.rstrip()
    # Ignorar 'líneas que no nos interesan'
    if not linea.startswith('From:'):
        continue
    # Procesar la línea que nos 'interesa'
    print(linea)
```

La salida del programa es la misma. En Español, las líneas no interesantes son aquellas que no comienzan con "From:", así que las saltamos utilizando `continue`. En cambio las líneas "interesantes" (aquellas que comienzan con "From:") las procesamos.

Podemos usar el método de cadenas `find` para simular la función de búsqueda de un editor de texto, que encuentra las líneas donde aparece la cadena de búsqueda en alguna parte. Puesto que `find` busca cualquier ocurrencia de una cadena dentro de otra y devuelve la posición de esa cadena o -1 si la cadena no fue encontrada, podemos escribir el siguiente bucle para mostrar las líneas que contienen la cadena "@uct.ac.za" (es decir, los que vienen de la Universidad de Cape Town en Sudáfrica):

```
man_f = open('mbox-short.txt')
for linea in man_f:
    linea = linea.rstrip()
    if linea.find('@uct.ac.za') == -1: continue
    print(linea)
```

Lo cual produce la siguiente salida:

```
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
...
```

Aquí utilizamos la forma contraída de la sentencia **if** donde ponemos el **continue** en la misma línea que el **if**. Esta forma contraída del **if** funciona de la misma manera que si el **continue** estuviera en la siguiente línea e indentado.

6. Permitiendo al usuario elegir el nombre de fichero

Definitivamente no queremos tener que editar nuestro código Python cada vez que queremos procesar un fichero diferente. Sería más útil pedir al usuario que introduzca el nombre del fichero cada vez que el programa se ejecuta, de modo que pueda usar nuestro programa en diferentes ficheros sin tener que cambiar el código.

Esto es sencillo de hacer leyendo el nombre de fichero del usuario utilizando **input** como se muestra a continuación:

```
nombre_f = input('Ingresa un nombre de un fichero: ')
man_f = open(nombre_f)
contador = 0
for linea in man_f:
    if linea.startswith('Subject:'):
        contador = contador + 1
print('Hay', contador, 'líneas de asunto (subject) en', nombre_f)
```

Leemos el nombre de fichero del usuario y lo guardamos en una variable llamada `man_f` y abrimos el fichero. Ahora podemos ejecutar el programa repetidamente en diferentes ficheros.

```
>>> %Run
  Ingresa un nombre de fichero: mbox.txt
  Hay 1797 líneas de asunto (subject) en mbox.txt
>>> %Run
  Ingresa un nombre de fichero: mbox-short.txt
  Hay 27 líneas de asunto (subject) en mbox-short.txt
```

Antes de mirar la siguiente sección, observa el programa anterior y pregúntate a ti mismo, "¿Qué error podría suceder aquí?" o "¿Qué podría nuestro amigable usuario hacer que cause que nuestro pequeño programa termine no exitosamente con un error, haciéndonos ver no-muy-geniales ante los ojos de nuestros usuarios?"

7. Utilizando try, except, y open

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

¿Qué tal si nuestro usuario escribe algo que no es un nombre de fichero?

```
>>> %Run
Ingresa un nombre de fichero: missing.txt
Traceback (most recent call last):
  File "search6.py", line 2, in <module>
    man_a = open(nfichero)
FileNotFoundError: [Errno 2] No such file or directory: 'missing.txt'
>>> %Run
Ingresa un nombre de fichero: na na boo boo
Traceback (most recent call last):
  File "search6.py", line 2, in <module>
    man_a = open(nfichero)
FileNotFoundError: [Errno 2] No such file or directory: 'na na boo boo'
```

No te rías. Los usuarios eventualmente harán cualquier cosa que puedan para estropear tus programas, sea a propósito o sin intenciones maliciosas. De hecho, una parte importante de cualquier equipo de desarrollo de software es una persona o grupo llamado *Quality Assurance* (Control de Calidad) (o QA en inglés) cuyo trabajo es probar las cosas más locas posibles en un intento de hacer fallar el software que el programador ha creado.

El equipo de QA (Control de Calidad) es responsable de encontrar los fallos en los programas antes de éstos sean entregados a los usuarios finales, que podrían comprar nuestro software o pagar nuestro salario por escribirlo. Así que el equipo de QA es el mejor amigo de un programador.

Ahora que vemos el defecto en el programa, podemos arreglarlo de forma elegante utilizando la estructura **try**/**except**. Necesitamos asumir que la llamada a **open** podría fallar y agregar código de recuperación para ese fallo, así:

```
nombre_f = input('Ingresa un nombre de archivo: ')
try:
    man_f = open(nombre_f)
except:
    print('No se puede abrir el archivo:', nombre_f)
    exit()
contador = 0
for linea in man_f:
    if linea.startswith('Subject:'):
        contador = contador + 1
print('Hay', contador, 'líneas de asunto (subject) en', nombre_f)
```

La función `exit` termina el programa. Es una función que llamamos que nunca retorna. Ahora cuando nuestro usuario (o el equipo de QA) introduzca algo sin sentido o un nombre de fichero incorrecto, vamos a "capturarlo" y recuperarnos de forma elegante:

```
>>> %Run
Ingresa un nombre de fichero: mbox.txt
Hay 1797 líneas de asunto (subject) en mbox.txt
>>> %Run
Ingresa un nombre de fichero: na na boo boo
No se puede abrir el fichero: na na boo boo
```

Proteger la llamada a **open** es un buen ejemplo del uso correcto de **try** y **except** en un programa de Python. Utilizamos el término "Pythónico" cuando estamos haciendo algo según el "estilo de Python". Podríamos decir que el ejemplo anterior es una forma Pythónica de abrir un fichero.

Una vez que estés más familiarizado con Python, puedes intercambiar opiniones con otros programadores de Python para decidir cuál de entre dos soluciones equivalentes a un problema es "más Pythónica". El

objetivo de ser "más Pythónico" engloba la noción de que programar es en parte ingeniería y en parte arte. No siempre estamos interesados sólo en hacer que algo funcione, también queremos que nuestra solución sea elegante y que sea apreciada como elegante por nuestros compañeros.

8. Escritura de ficheros

Para escribir en un fichero, tienes que abrirlo en modo "w" (de `write`, escritura) como segundo parámetro:

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

Si el fichero ya existía previamente, abrirlo en modo de escritura causará que se borre todo el contenido del fichero, así que ¡ten cuidado! Si el fichero no existe, un nuevo fichero es creado.

El método `write` del manejador de ficheros escribe datos dentro del fichero, devolviendo el número de caracteres escritos. El modo de escritura por defecto es texto para escribir (y leer) cadenas.

```
>>> linea1 = "Aquí está el zarzo,\n"
>>> fsal.write(linea1)
20
```

El manejador de fichero mantiene un seguimiento de dónde está, así que si llamas a `write` de nuevo, éste agrega los nuevos datos al final.

Debemos asegurarnos de gestionar los finales de las líneas conforme vamos escribiendo en el fichero, insertando explícitamente el carácter de salto de línea cuando queremos finalizar una línea. La sentencia **`print`** agrega un salto de línea automáticamente, pero el método `write` no lo agrega de forma automática.

```
>>> linea2 = 'el símbolo de nuestra tierra.\n'
>>> fsal.write(linea2)
30
```

Cuando terminas de escribir, tienes que cerrar el fichero para asegurarte que la última parte de los datos es escrita físicamente en el disco duro, de modo que no se pierdan los datos si la corriente eléctrica se interrumpe.

```
>>> fsal.close()
```

Podríamos cerrar los ficheros abiertos para lectura también, pero podemos ser menos rigurosos si sólo estamos abriendo unos pocos ficheros puesto que Python se asegura de que todos los ficheros abiertos sean cerrados cuando termina el programa. En cambio, cuando estamos escribiendo ficheros debemos cerrarlos de forma explícita para no dejar nada al azar.

9. Depuración

Cuando estás leyendo y escribiendo ficheros, puedes tener problemas con los espacios en blanco. Esos errores pueden ser difíciles de depurar debido a que los espacios, tabuladores, y saltos de línea son invisibles normalmente:

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

La función nativa **repr** puede ayudarte. Recibe cualquier objeto como argumento y devuelve una representación del objeto como una cadena. En el caso de las cadenas, representa los espacios en blanco con secuencias de barras invertidas:

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

Esto puede ser útil para depurar.

Otro problema que podrías tener es que diferentes sistemas usan diferentes caracteres para indicar el final de una línea. Algunos sistemas usan un salto de línea, representado como \\00n. Otros usan un carácter de retorno, representado con \\00r. Otros usan ambos. Si mueves ficheros entre diferentes sistemas, esas inconsistencias podrían causarte problemas.

Para la mayoría de los sistemas, hay aplicaciones que convierten de un formato a otro. Puedes encontrarlas (y leer más acerca de esto) en wikipedia.org/wiki/Newline. O también, por supuesto, puedes escribir una tu mismo.

10. Glosario

fichero de texto Una secuencia de caracteres almacenados en un dispositivo de almacenamiento permanente como un disco duro.

capturar (catch) Evitar que una excepción haga terminar un programa, usando las sentencias **try** y **except**.

control de calidad (QA) Una persona o equipo enfocado en asegurar la calidad en general de un producto. El Control de calidad (QA) es frecuentemente encargado de probar un software y encontrar posibles problemas antes de que el software sea lanzado.

pythónico Una técnica que funciona de forma elegante en Python. "Utilizar try y except es la forma *Pythónica* de gestionar los ficheros inexistentes".

salto de línea Un carácter especial utilizado en ficheros y cadenas para indicar el final de una línea.

Ejercicios

Exercise 1:

Escribe un programa que lea el fichero `mbox-short.txt` e imprima su contenido (línea por línea), todo en mayúsculas. Para testear el programa ejecútalo, y verifica que sale esto:

```
>>> %Run
Ingresa un nombre de fichero: 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 fichero `mbox-short.txt` desde Poliformat.

Exercise 2:

Escribe un programa que solicite un nombre de fichero y después lea ese fichero buscando las líneas que tengan la siguiente forma:

```
X-DSPAM-Confidence: 0.8475
```

**Cuando encuentres una línea que comience con "X-DSPAM-Confidence:" ponla aparte para extraer el número decimal de la línea. Cuenta esas líneas y después calcula el total acumulado de los valores de "spam-confidence". Cuando llegues al final del fichero, imprime el valor medio de "spam confidence".

```
>>> %Run
Ingresa un nombre de fichero: mbox.txt
Promedio spam confidence: 0.894128046745

>>> %Run
Ingresa un nombre de fichero: mbox-short.txt
Promedio spam confidence: 0.750718518519
```

Prueba tu programa con los ficheros `mbox.txt` y `mbox-short.txt` que están en Poliformat.

Exercise 3:

Algunas veces cuando los programadores se aburren o quieren divertirse un poco, agregan un inofensivo *Huevo de Pascua* a su programa. Modifica el programa que pregunta al usuario por el nombre de fichero para que imprima un mensaje divertido cuando el usuario escriba "na na boo boo" como nombre de fichero. El programa debería funcionar normalmente para cualquier fichero que exista o no exista. Aquí tres ejemplos para testear tu programa:

```
>>> %Run
Ingresa un nombre de fichero: mbox.txt
Hay 1797 líneas subject en mbox.txt

>>> %Run
Ingresa un nombre de fichero: inexistente.tyxt
El fichero no se puede abrir: inexistente.tyxt

>>> %Run
Ingresa un nombre de fichero: na na boo boo
NA NA BOO BOO PARA TI - Te he atrapado!
```

No te estamos aconsejando poner Huevos de Pascua en tus programas; es sólo un ejercicio.

Exercise 4:

Escribe un programa que lee el nombre de un fichero de texto del teclado y muestra por pantalla el texto codificado de forma que sólo las letras minúsculas se sustituyen por las siguientes.

Por ejemplo, si el fichero es:

message.txt

```
This is a secret message
```

```
>>> %Run
Introduzca un nombre de fichero: message.txt
Tijt jt b tfdsfu nfttbhf
```

Ejecuta tests de tu función creando diferentes ficheros de texto como entrada para tu programa y verifica que el resultado es la salida esperada.

Exercise 5:

Necesitamos anonimizar el fichero `interview.txt` en Poliformat. Este fichero contiene una entrevista con Bill, pero su nombre no puede aparecer y hay que cambiarlo a Pepito para anonimizar el fichero. Tu función tiene que devolver el nombre del fichero anonimizado.

Escribe una función `anonimizar` en Python que puede anonimizar este fichero. La función recibe 3 parámetros, el nombre de un fichero y dos strings `s1` y `s2`. En todo el fichero hay que cambiar las ocurencias del `s1` por el `s2`.

Puedes probar tu función manualmente usando el fichero `interview.txt`

```
>>> interview_file = "interview.txt"
>>> new_file = anonimizar(interview_file, "Bill", "Pepito")
>>> manej_new_file = open(new_file)
>>> for l in manej_new_file:
    print(l)

Pepito is a Syrian refugee. She arrived three days..
........
........
```

Exercise 6:

Para testear el ejercicio anterior de forma automatico, vamos a escribir otra función `en_fichero` que toma como argumento un nombre de un fichero y un string y devuelve `True` si el string aparece en el fichero y `False` si no esta.

La idea es testear que nuestra función `anonimizar` ha anonimizado bien el fichero, por ejemplo de esta manera:

```
@pytest.mark.parametrize("testcase, in1, in2, in3",[
(1, "interview.txt", "Bill", "Pepito")
])

def test_anonimizar(testcase, in1, in2, in3):
    assert not (en_fichero(anonimizar(in1, in2, in3),in2)),\
           "caso {0}".format(testcase)
```

Crear 2 ficheros de texto más para anonimizar y añade 2 casos de test.

Exercise 7:

Escribir una función que pida un número entero entre 1 y 20 y guarde en un fichero con el nombre tabla-div-n.txt la tabla de división de ese número, done n es el número introducido. El formato del fichero para n es 3 sería:

```
 3  :  3 =  1
 6  :  3 =  2
 9  :  3 =  3
12  :  3 =  4
.
.
.
.
.
30  :  3 = 10
```

Prueba a mano tu función con algunos valores.

Exercise 8:

Para hacer un poco más facil el testing del ejercico anterior, y para no tener que estar abriendo y cerrando ficheros todo el tiempo, vamos a escribir una función `print_tabla` que pida un número n entero entre 1 y 10, lea el fichero tabla-div-n.txt generado con la tabla de división de ese número y la muestre por pantalla. Si el fichero no existe debe mostrar un mensaje por pantalla informando de ello.

Exercise 9:

Al final decidimos automatizar el testing de la función para hacer aun más fácil el testing del ejercicio anterior, y para no tener que estar abriendo y cerrando ficheros todo el tiempo. Vamos a escribir una función `check_tabla` que pida un número n entero entre 1 y 10, lea el fichero tabla-div-n.txt generado con la tabla de división de ese número y devuelve `True` si el fichero es correcto y `False` si no. Si el fichero no existe debe mostrar un mensaje por pantalla informando de ello.

```
@pytest.mark.parametrize("testcase, entrada",[
(1, 3),
(2, 5),
(4, 9),
(10, 11)
])

def test_check_tabla(testcase, entrada):
    tabla_division(entrada)
    assert check_tabla(entrada),"caso {0}".format(testcase)
```