

# 8 Listas

## Una lista es una secuencia

Como una cadena, una **lista** es una secuencia de valores. En una cadena, los valores son caracteres; en una lista, pueden ser de cualquier tipo. Los valores en la lista se denominan **elementos**.

Hay varias formas de crear una nueva lista; lo más simple es encerrar los elementos entre corchetes (`[` y `]`):

```
[10, 20, 30, 40]  
['crunchy frog', 'ram bladder', 'lark vomit']
```

El primer ejemplo es una lista de cuatro enteros. El segundo es una lista de tres cadenas. Los elementos de una lista no tienen que ser del mismo tipo. La siguiente lista contiene una cadena, un decimal, un entero y... ¡otra lista!:

```
['spam', 2.0, 5, [10, 20]]
```

Una lista dentro de otra lista está **anidada**.

Una lista que no contiene elementos se llama una lista vacía; puedes crear una con corchetes vacíos, `[]`.

```
empty_list = []
```

Como es de esperar, puedes asignar valores de lista a variables.

## Las listas son mutables

La sintaxis para acceder a los elementos de una lista es la misma que para acceder a los caracteres de una cadena: el operador de corchete. La expresión dentro de los paréntesis especifica el índice. Recuerda que los índices comienzan por 0:

```
>>> print(cheeses[0])  
Cheddar
```

A diferencia de las cadenas, las listas son mutables porque puede cambiar el orden de los elementos en una lista o reasignar un elemento en una lista. Cuando el operador de corchetes aparece en el lado izquierdo de una asignación, identifica el elemento de la lista que se asignará.

```
>>> numbers = [17, 123]  
>>> numbers[1] = 5  
>>> print(numbers)  
[17, 5]  
>>>
```

El elemento one-eth de `numbers`, que era 123, ahora es 5.

Puede pensar en una lista como una relación entre índices y elementos. Esta relación se llama **mapeo**; cada índice "mapea" a uno de los elementos.

Los índices de lista funcionan de la misma manera que los índices de cadena:

- Cualquier `entero` puede ser usado como un índice.
- Si intentas leer o escribir un elemento que no existe, obtienes un 'IndexError'.
- Si un índice tiene un valor negativo, cuenta hacia atrás desde el final de la lista.

El operador `in` también funciona en listas.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']  
>>> 'Edam' in cheeses  
True  
>>> 'Brie' in cheeses  
False  
>>>
```

## Atravesando una lista

La forma más común de recorrer los elementos de una lista es con un bucle `for`. La sintaxis es la misma que para las cadenas:

```
for cheese in cheeses:  
    print(cheese)
```

Esto funciona bien si solo necesitas leer los elementos de la lista. Pero si deseas escribir o actualizar los elementos, necesitas los índices. Una forma común de hacerlo es combinar las funciones `range` y `len`:

```
for i in range(len(numbers)):  
    numbers[i] = numbers[i] * 2
```

Este bucle atraviesa la lista y actualiza cada elemento. `len` devuelve el número de elementos en la lista. `range` devuelve una lista de índices de 0 a **n-1**, donde **n** es la longitud de la lista. Cada vez que pasa por el bucle, `i` obtiene el índice del siguiente elemento. La declaración de asignación en el cuerpo usa `i` para leer el valor antiguo del elemento y para asignar el nuevo valor.

Un bucle `for` sobre una lista vacía nunca ejecuta el cuerpo:

```
for x in empty:  
    print('This never happens.')
```

Aunque una lista puede contener otra lista, la lista anidada todavía cuenta como un elemento único. La longitud de esta lista es cuatro:

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

## Operaciones de lista

El operador `+` concatena las listas:

```
>>> a = [1, 2, 3]  
>>> b = [4, 5, 6]  
>>> c = a + b  
>>> print(c)  
[1, 2, 3, 4, 5, 6]  
>>>
```

Del mismo modo, el operador `*` repite una lista un número determinado de veces:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>>
```

El primer ejemplo se repite cuatro veces. El segundo ejemplo repite la lista tres veces.

## Listar segmentos

El operador de corte también funciona en listas:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
>>>
```

Si omites el primer índice, la división comienza al principio. Si omites el segundo, la rebanada va al final. Entonces, si omite ambos, la porción es una copia de toda la lista.

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Dado que las listas son mutables, a menudo es útil hacer una copia antes de realizar operaciones de pegado, huso o mutilación de listas.

Un operador de sector en el lado izquierdo de una asignación puede actualizar varios elementos:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> print(t)
['a', 'x', 'y', 'd', 'e', 'f']
```

```
>>>
```

# Métodos de lista

Python proporciona métodos que operan en listas. Por ejemplo, `append` agrega un nuevo elemento al final de una lista:

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print(t)
['a', 'b', 'c', 'd']
```

`extend` toma una lista como argumento y agrega todos los elementos:

```
>>> t = ['a', 'b', 'c']
>>> t.extend(['d', 'f'])
>>> print(t)
['a', 'b', 'c', 'd', 'f']
```

Este ejemplo deja `t2` sin modificar.

`sort` organiza los elementos de la lista de bajo a alto:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> print(t)
['a', 'b', 'c', 'd', 'e']
```

La mayoría de los métodos de lista son nulos; modifican la lista y devuelven `None`. Si accidentalmente escribes `t = t.sort ()`, quedarás decepcionado con el resultado.

# Eliminando elementos

Hay varias formas de eliminar elementos de una lista. Si conoces el índice del elemento que deseas, puede usar `pop`:

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> print(t)
['a', 'c']
>>> print(x)
b
>>>
```

`pop` modifica la lista y devuelve el elemento que fue eliminado. Si no proporcionas un índice, elimina y devuelve el último elemento.

Si no necesitas el valor eliminado, puede usar el operador `del`:

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> print(t)
['a', 'c']
>>>
```

Si conoces el elemento que deseas eliminar (pero no el índice), puedes usar `remove`:

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> print(t)
['a', 'c']
>>>
```

El valor de retorno de `remove` es `None`.

Para eliminar más de un elemento, puedes usar `del` con un índice de división:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> print(t)
['a', 'f']
>>>
```

Como es habitual, la división selecciona todos los elementos hasta el segundo índice, pero sin incluirlo.

# Listas y funciones

Hay una serie de funciones incorporadas que se pueden usar en listas que te permiten mirar rápidamente una lista sin escribir sus propios bucles:

```
>>> nums = [3, 41, 12, 9, 74, 15]
>>> print(len(nums))
6
>>> print(max(nums))
74
>>> print(min(nums))
3
>>> print(sum(nums))
154
>>> print(sum(nums)/len(nums))
25.666666666666668
```

La función `sum()` solo funciona cuando los elementos de la lista son números. Las otras funciones (`max()`, `len()`, etc.) funcionan con listas de cadenas y otros tipos que pueden ser comparables.

Podríamos reescribir un programa anterior que calculó el promedio de una lista de números ingresados por el usuario usando una lista.

Primero, el programa para calcular un promedio sin una lista:

<https://trinket.io/embed/python3/e02540c91c>

En este programa, tenemos las variables `count` y `total` para mantener el número y el total acumulado de los números del usuario, ya que repetidamente pedimos al usuario un número.

Simplemente podríamos recordar cada número cuando el usuario lo ingresó y usar funciones integradas para calcular la suma y el conteo al final.

<https://trinket.io/embed/python3/d5f9bf95ff>

Hacemos una lista vacía antes de que comience el ciclo, y luego cada vez que tenemos un número, lo agregamos a la lista. Al final del programa, simplemente calculamos la suma de los números en la lista y la dividimos por el recuento de los números en la lista para obtener el promedio.

## Listas y cadenas

Una cadena es una secuencia de caracteres y una lista es una secuencia de valores, pero una lista de caracteres no es lo mismo que una cadena. Para convertir de una cadena a una lista de caracteres, puedes usar `list`:

```
>>> s = 'spam'
>>> t = list(s)
>>> print(t)
['s', 'p', 'a', 'm']
```

Como `list` es el nombre de una función incorporada, debes evitar usarla como nombre de variable. También evito la letra `l` porque se parece demasiado al número `1`. Así que por eso uso `t`.

La función `list` rompe una cadena en letras individuales. Si deseas dividir una cadena en palabras, puede usar el método `split`:

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> print(t)
['pining', 'for', 'the', 'fjords']
>>> print(t[2])
the
```

Una vez que hayas usado `split` para dividir la cadena en una lista de palabras, puedes usar el operador de índice (corchete) para mirar una palabra en particular en la lista.

Puedes llamar a `split` con un argumento opcional llamado **delimiter** que especifica qué caracteres usar como límites de palabras. El siguiente ejemplo usa un guión como delimitador:



```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> s.split(delimiter)
['spam', 'spam', 'spam']
>>>
```

`join` es el inverso de `split`. Toma una lista de cadenas y concatena los elementos. `join` es un método de cadena, por lo que debes invocarlo en el delimitador y pasar la lista como parámetro:

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> delimiter.join(t)
'pining for the fjords'
```

En este caso, el delimitador es un carácter de espacio, por lo que `join` pone un espacio entre las palabras. Para concatenar cadenas sin espacios, puedes usar la cadena vacía "" como un delimitador.

## Parseando líneas

Por lo general, cuando estamos leyendo un archivo queremos hacer algo en las líneas que no sea simplemente imprimir toda la línea. A menudo queremos encontrar las "líneas interesantes" y luego **analizar** la línea para encontrar algunas **partes** interesantes de la línea. ¿Qué pasaría si quisiéramos imprimir el día de la semana desde esas líneas que comienzan con "From"?

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
```

El método de `split` es muy efectivo cuando se enfrenta a este tipo de problemas. Podemos escribir un pequeño programa que busque líneas donde la línea comience con "From", `dividir` esas líneas, y luego imprimir la tercera palabra en la línea:

<https://trinket.io/embed/python3/106858f1a8>

Aquí también usamos la forma contraída de la declaración `if` donde colocamos el `continue` en la misma línea que el `if`. Esta forma contraída de `if` funciona de la misma manera que si `continue` estuviera en la siguiente línea y con sangría.

El programa produce la siguiente salida:

```
Sat
Fri
Fri
Fri
...
```

Más adelante, aprenderemos técnicas cada vez más sofisticadas para elegir las líneas en las que trabajar y cómo separamos esas líneas para encontrar la información exacta que estamos buscando.

## Objetos y valores

Si ejecutamos estas sentencias de asignación:

```
a = 'banana'
b = 'banana'
```

sabemos que `a` y `b` se refieren a una cadena, pero no sabemos si se refieren a la *misma* cadena. Hay dos estados posibles:



Imagen - Variables y objetos

En un caso, `a` y `b` se refieren a dos objetos diferentes que tienen el mismo valor. En el segundo caso, se refieren al mismo objeto.

Para verificar si dos variables se refieren al mismo objeto, puede usar el operador `is`.

```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

En este ejemplo, Python solo creó un objeto de cadena, y tanto `a` como `b` se refieren a él.



Pero cuando creas dos listas, obtienes dos objetos:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
>>>
```

En este caso, diríamos que las dos listas son **equivalentes**, porque tienen los mismos elementos, pero no **idénticas**, porque no son el mismo objeto. Si dos objetos son idénticos, también son equivalentes, pero si son equivalentes, no son necesariamente idénticos.

Hasta ahora, hemos estado utilizando indistintamente "objeto" y "valor", pero es más preciso decir que un objeto tiene un valor. Si ejecuta `a = [1,2,3]`, `a` se refiere a un objeto de lista cuyo valor es una secuencia particular de elementos. Si otra lista tiene los mismos elementos, diríamos que tiene el mismo valor.

## Aliasing

Si `a` se refiere a un objeto y usted asigna `b = a`, entonces ambas variables se refieren al mismo objeto:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b
[1, 2, 3]
>>> a[2] = 4
>>> b
[1, 2, 4]
```

La asociación de una variable con un objeto se denomina **referencia**. En este ejemplo, hay dos referencias al mismo objeto.

Un objeto con más de una referencia tiene más de un nombre, por lo que decimos que el objeto tiene un **alias**.

Si el objeto con alias es mutable, los cambios realizados con un alias afectan al otro:

```
>>> b[0] = 17
>>> print(a)
[17, 2, 3]
```

Aunque este comportamiento puede ser útil, es propenso a errores. En general, es más seguro evitar los alias cuando trabaja con objetos mutables.

Para objetos inmutables como cadenas, el aliasing no es tan problemático.

## Lista de argumentos

Cuando pasas una lista a una función, la función obtiene una referencia a la lista. Si la función modifica un parámetro de lista, la función ve el cambio. Por ejemplo, `delete_head` elimina el primer elemento de una lista:

```
def delete_head(t):
    del t[0]
```

Así es como se usa:

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> print(letters)
['b', 'c']
>>>
```

El parámetro `t` y la variable `letters` son alias para el mismo objeto.

Es importante distinguir entre operaciones que modifican listas y operaciones que crean nuevas listas. Por ejemplo, el método `append` modifica una lista, pero el operador `+` crea una nueva lista:

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> print(t1)
[1, 2, 3]
>>> print(t2)
None
```

```
>>> t3 = t1 + [3]
>>> print(t3)
[1, 2, 3, 3]
>>> t2 is t3
False
>>>
```

Esta diferencia es importante cuando escribes funciones que se supone que modifican las listas. Por ejemplo, esta función **no** elimina el encabezado de una lista:

```
def bad_delete_head(t):
    t = t[1:]          # WRONG!
```

El operador de sector crea una nueva lista y la asignación hace que `t` se refiera a ella, pero nada de eso tiene ningún efecto en la lista que se pasó como argumento.

Una alternativa es escribir una función que crea y devuelve una nueva lista. Por ejemplo, `tail` devuelve todos menos el primer elemento de una lista:

```
def tail(t):
    return t[1:]
```

Esta función deja la lista original sin modificar. Así es como se usa:

```
>>> letters = ['a', 'b', 'c']
>>> rest = tail(letters)
>>> print(rest)
['b', 'c']
```

### Ejercicio 1:

Escriba una función llamada `chop` que tome una lista y la modifique, eliminando el primer y último elemento y devuelva `None`.

Luego escribe una función llamada `middle` que tome una lista y devuelva una nueva lista que contiene todos los elementos, excepto el primero y el último.

## Depuración

El uso descuidado de listas (y otros objetos mutables) puede llevar a largas horas de depuración. Aquí hay algunos escollos comunes y maneras de evitarlos:

No olvides que la mayoría de los métodos de lista modifican el objeto y devuelven `None`. Esto es lo contrario de los métodos de cadena, que devuelven una cadena nueva y dejan el original inalterado.

Si estás acostumbrado a escribir un código de cadena como este:

```
word = word.strip()
```

Es tentador escribir un código de lista como este:

```
t = t.sort () # WRONG!
```

Debido a que `sort` devuelve `None`, la siguiente operación que realice con `t` es probable que falle.

Antes de utilizar los métodos de lista y los operadores, debes leer la documentación detenidamente y luego probarlos en modo interactivo. Los métodos y operadores que las listas comparten con otras secuencias (como cadenas) se documentan en

<https://docs.python.org/3.7/library/stdtypes.html#string-methods>. Los métodos y operadores que solo se aplican a secuencias mutables están documentados en

<https://docs.python.org/3.7/library/stdtypes.html#mutable-sequence-types>.

Escoge una sintaxis y quédate con ella.

Parte del problema con las listas es que hay demasiadas formas de hacer las cosas. Por ejemplo, para eliminar un elemento de una lista, puedes usar `pop`, `remove`, `del`, o incluso una asignación de sector.

Para agregar un elemento, puede usar el método `append` o el operador `+`. Pero no olvides que estos son correctos:

```
t.append(x)
t = t + [x]
```

Y estos son erróneos:



```
t.append([x])          # WRONG!  
t = t.append(x)        # WRONG!  
t + [x]                # WRONG!  
t = t + x              # WRONG!
```

Prueba cada uno de estos ejemplos en modo interactivo para asegurarse de que comprendes lo que hacen. Observa que solo el último causa un error en tiempo de ejecución; los otros tres son legales, pero no hacen lo que queremos.

Haz copias para evitar el aliasing.

Si deseas utilizar un método como `sort` que modifica el objeto, pero también necesita conservar la lista original, puede hacer una copia.

```
orig = t[:]
t.sort()
```

En este ejemplo, también podrías usar la función incorporada `sorted`, que devuelve una nueva lista ordenada y deja el original inalterado. ¡Pero en ese caso, debes evitar usar `sorted` como nombre de variable!

Listas, `split` y archivos

Cuando leemos y analizamos archivos, hay muchas oportunidades de encontrar información que puede bloquear nuestro programa, por lo que es una buena idea revisar el patrón **guardian** cuando se trata de escribir programas que leen un archivo y buscan una "aguja en el pajar".

Revisemos nuestro programa que busca el día de la semana en las líneas de nuestro archivo:

```
From stephen.marquard@uct.ac.zaSatJan 5 09:14:16 2008
```

Ya que estamos dividiendo esta línea en palabras, podríamos prescindir del uso de `startswith` y simplemente mirar la primera palabra de la línea para determinar si estamos interesados en ella. Podemos usar `continue` para omitir las líneas que no tienen "From" como la primera palabra de la siguiente manera:

```
fhand = open('mbox-short.txt')
for line in fhand:
    words = line.split()
```

```
if words[0] != 'From' : continue  
print(words[2])
```

Esto parece mucho más simple y ni siquiera necesitamos hacer el `rstrip` para eliminar la nueva línea al final del archivo. Pero, ¿es mejor?

```
python search8.py  
Sat  
Traceback (most recent call last):  
  File "search8.py", line 5, in <module>  
    if words[0] != 'From' : continue  
IndexError: list index out of range
```

Funciona y vemos el día desde la primera línea (Sat), pero luego el programa falla con un error de rastreo. ¿Qué salió mal? ¿Qué datos desordenados hicieron que nuestro programa elegante, inteligente y muy Pythonic fallara?

Puedes mirarlo fijamente durante un largo tiempo y resolverlo o pedirle ayuda a alguien, pero el enfoque más rápido e inteligente es agregar una declaración `print()`. El mejor lugar para agregar la declaración de impresión es justo antes de la línea donde falló el programa e imprimir los datos que parecen estar causando la falla.

Ahora este enfoque puede generar una gran cantidad de líneas de salida, pero al menos tendrás inmediatamente alguna pista sobre el problema en cuestión. Así que agregamos una impresión de la variable `words` justo antes de la línea cinco. Incluso agregamos un prefijo "Debug:" a la línea para que podamos mantener nuestra salida regular separada de nuestra salida de depuración.

```
for line in fhand:  
    words = line.split()  
    print('Debug:', words)  
    if words[0] != 'From' : continue  
    print(words[2])
```

Cuando ejecutamos el programa, una gran cantidad de salida se desplaza fuera de la pantalla, pero al final, vemos nuestra salida de depuración y el rastreo, por lo que sabemos lo que sucedió justo antes del rastreo.

```
Debug: ['X-DSPAM-Confidence:', '0.8475']  
Debug: ['X-DSPAM-Probability:', '0.0000']
```



```
Debug: []  
Traceback (most recent call last):  
  File "search9.py", line 6, in <module>  
    if words[0] != 'From' : continue  
IndexError: list index out of range
```

Cada línea de depuración está imprimiendo la lista de palabras que obtenemos cuando "dividimos" la línea en palabras. Cuando el programa falla, la lista de palabras está vacía `[]`. Si abrimos el archivo en un editor de texto y miramos el archivo, en ese punto se ve como sigue:

```
X-DSPAM-Result: Innocent  
X-DSPAM-Processed: Sat Jan  5 09:14:16 2008  
X-DSPAM-Confidence: 0.8475  
X-DSPAM-Probability: 0.0000  
  
Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
```

¡El error ocurre cuando nuestro programa encuentra una línea en blanco! Por supuesto, hay "cero palabras" en una línea en blanco. ¿Por qué no pensamos en eso cuando estábamos escribiendo el código? Cuando el código busca la primera palabra (`words[0]`) para verificar si coincide con "From", obtenemos un error de "índice fuera de rango".

Por supuesto, este es el lugar perfecto para agregar un código **guardian** para evitar marcar la primera palabra si la primera palabra no está allí. Hay muchas maneras de proteger este código. Elegiremos verificar el número de palabras que tenemos antes de mirar la primera palabra:

```
fhand = open('mbox-short.txt')  
count = 0  
for line in fhand:  
    words = line.split()  
    # print 'Debug:', words  
    if len(words) == 0 : continue  
    if words[0] != 'From' : continue  
    print(words[2])
```

Primero comentamos la declaración de impresión de depuración en lugar de eliminarla, en caso de que nuestra modificación falle y tengamos que volver a depurar. Luego agregamos una declaración que verifique si tenemos cero palabras y, si es así, usamos `continue` para saltar a la siguiente línea del archivo.



Podemos pensar que las dos afirmaciones "continue" nos ayudan a refinar el conjunto de líneas que son "interesantes" para nosotros y que queremos procesar un poco más. Una línea que no tiene palabras es "poco interesante" para nosotros, así que saltamos a la siguiente línea. Una línea que no tiene "From" como su primera palabra no nos interesa, así que la omitimos.

El programa modificado se ejecuta con éxito, así que quizás sea correcto. Nuestra declaración guardiana asegura que `words[0]` nunca fallarán, pero tal vez no sea suficiente. Cuando estamos programando, siempre debemos pensar: "¿Qué podría salir mal?"

## Ejercicios

**Ejercicio 2:** Averigua qué línea del programa anterior aún no está correctamente protegida. Intenta construir un archivo de texto que haga que el programa falle y luego modifícalo para que la línea esté bien protegida y pruébalo para asegurarte de que manejas tu nuevo archivo de texto.

**Ejercicio 3:** reescribe el código guardián en el ejemplo anterior sin dos declaraciones 'if'. En su lugar, usa una expresión lógica compuesta utilizando el operador lógico `and` con una sola instrucción `if`.

**Ejercicio 4:** descarga una copia del archivo desde <http://www.py4e.com/code3/romeo.txt>

Escribe un programa para abrir el archivo `romeo.txt` y léelo línea por línea. Para cada línea, divide la línea en una lista de palabras usando la función `split`.

Para cada palabra, verifica si la palabra ya está en una lista. Si la palabra no está en la lista, agrégala a la lista.

Cuando se complete el programa, ordena e imprime las palabras resultantes en orden alfabético.

```
```Enter file: romeo.txt ['Arise', 'But', 'It', 'Juliet', 'Who', 'already', 'and', 'breaks', 'east', 'envious',
'fair', 'grief', 'is', 'kill', 'light', 'moon', 'pale', 'sick', 'soft', 'sun', 'the', 'through', 'what', 'window',
'with', 'yonder']```
```

**\*\*Ejercicio 5:\*\*** Escribe un programa para leer los datos del buzón de correo y cuando encuentre la línea que comienza con "From", dividirá la línea en palabras usando la función `split`. Estamos interesados en quién envió el mensaje, que es la segunda palabra en la línea From.

`De stephen.marquard@uct.ac.za sábado 5 de enero 09:14:16 2008`

Analizará la línea Desde e imprimirá la segunda palabra para cada línea From, también contará el número de líneas From (no From:) e imprimirá un recuento al final.

Esta es una buena salida de muestra con algunas líneas eliminadas:

```
```python
python fromcount.py
Enter a file name: mbox-short.txt
stephen.marquard@uct.ac.za
louis@media.berkeley.edu
zqian@umich.edu

[...some output removed...]

ray@media.berkeley.edu
cwen@iupui.edu
cwen@iupui.edu
cwen@iupui.edu
There were 27 lines in the file with From as the first word
```

**Ejercicio 6:** Reescribe el programa que solicita al usuario una lista de números e imprime el máximo y el mínimo de los números al final cuando el usuario ingresa "listo". Escribe el programa para almacenar los números que el usuario ingresa en una lista y use las funciones `max()` y `min()` para calcular los números máximo y mínimo después de que se complete el ciclo.

```
Enter a number: 6
Enter a number: 2
Enter a number: 9
Enter a number: 3
Enter a number: 5
Enter a number: done
Maximum: 9.0
Minimum: 2.0
```



Revision #1

Created 5 April 2025 10:19:36 by Javier Quintana

Updated 5 April 2025 11:10:14 by Javier Quintana