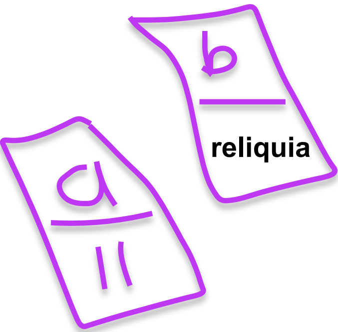


Chapter 8

Listas

R = 5i+6j+7k
R =(5,6,7)

8.1 Una lista es una secuencia

Así como una cadena, una *lista* es una secuencia de valores. En una cadena, los valores son caracteres; en una lista, pueden ser cualquier tipo. Los valores en una lista son llamados *elementos* o a veces *ítems*.

Hay varias formas de crear una nueva lista; la más simple es encerrar los elementos en corchetes (“[" y “]”):

```
[10, 20, 30, 40]
['rana crujiente', 'vejiga de carnero', 'vómito de alondra']
```

El primer ejemplo es una lista de 4 enteros. La segunda 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 flotante, un entero, y (¡mira!) otra lista:

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

Una lista dentro de otra lista está *anidada*.

Una lista que no contiene elementos es llamada una lista vacía; puedes crear una con corchetes vacíos, [].

Como puedes ver, puedes asignar los valores de una lista a variables:

```
>>> quesos = ['Cheddar', 'Edam', 'Gouda']
>>> numeros = [17, 123]
>>> vacia = []
>>> print(quesos, numeros, vacia)
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

8.2 Las listas son mutables

La sintaxis para accesar elementos de una lista es la misma que para accesar los caracteres de una cadena: el operador corchete. La expresión dentro de los corchetes especifíca el índice. Recordemos que los índices empiezan en 0:

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

A diferencia de las cadenas, las listas son mutables porque pueden cambiar el orden de los elementos en una lista o reasignar un elemento en una lista. Cuando el operador corchete aparece en el lado izquierdo de una asignación, éste identifica el elemento de la lista que será asignado.

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

El elemento en la posición uno de `numeros`, el cual solía ser 123, es ahora 5.

Puedes pensar en una lista como una relación entre índices y elementos. Esta relación es llamada *mapeo*; cada índice "mapea a" uno de los elementos.

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

- Cualquier forma de entero puede ser utilizada como índice.
- Si tratas de leer o escribir un elemento que no existe, obtendrás un `IndexError`.
- Si un índice tiene un valor negativo, éste cuenta hacia atrás desde el final de la lista.

El operador `in` funciona también en listas.

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

8.3 Recorriendo 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 queso in quesos:
    print(queso)
```

Esto funciona bien si solamente necesitas leer los elementos de la lista. Pero si quieres escribir o actualizar los elementos, necesitas los índices. Una forma común de hacer eso es combinando las funciones `range` y `len`:

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

Este bucle recorre la lista y actualiza cada elemento. `len` regresa el número de elementos en una lista. `range` regresa una lista de índices desde 0 hasta $n - 1$, donde n es la longitud de la lista. Cada vez que pasa a través del recorrido, `i` obtiene el índice del siguiente elemento. La sentencia de asignación dentro del bucle utiliza `i` para leer el valor original del elemento y asignar un nuevo valor.

Un bucle `for` a través de una lista vacía nunca ejecuta el código contenido en el cuerpo:

```
for x in vacia:
    print('Esto nunca sucede.')
```

Aunque una lista puede contener otra lista, las listas anidadas siguen contando como un solo elemento. El tamaño de esta lista es cuatro:

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

8.4 Operaciones de listas

El operador `+` concatena listas:

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

De igual forma, el operador `*` repite una lista un determinado número de veces:

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

En el primer ejemplo se repite cuatro veces. En el segundo ejemplo se repite la lista tres veces.

8.5 Rebanado de listas

El operador de rebanado 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, el rebanado comienza desde el inicio de la lista. Si omites el segundo, el rebanado se va hasta el final. Así que si omites ambos, el rebanado es una copia de la lista completa.

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

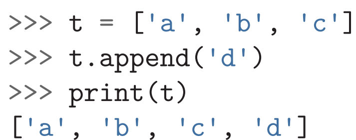
Como las listas son mutables, a veces es útil hacer una copia antes de hacer operaciones que doblan, pegan, o cortan listas.

Un operador de rebanado al lado izquierdo de una asignación puede actualizar múltiples elementos:

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

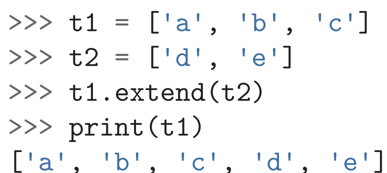
8.6 Métodos de listas

Python provee 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:



```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> print(t1)
['a', 'b', 'c', 'd', 'e']
```

Este ejemplo deja `t2` sin modificar.

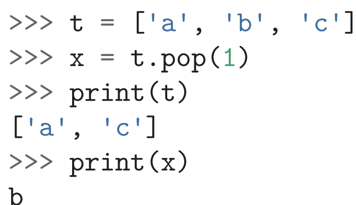
`sort` ordena los elementos de la lista de menor a mayor:

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

La mayoría de métodos no regresan nada; modifican la lista y regresan `None`. Si accidentalmentes escribes `t = t.sort()`, vas a decepcionarte con el resultado.

8.7 Eliminando elementos

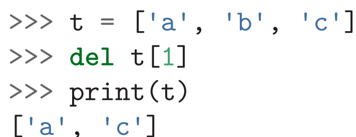
Hay varias formas de eliminar elementos de una lista. Si sabes el índice del elemento que quieres, puedes usar `pop`:



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

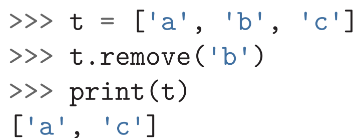
`pop` modifica la lista y regresa el elemento que fue removido. Si no provees un índice, la función elimina y retorna el último elemento.

Si no necesitas el valor removido, puedes usar el operador `del`:



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

Si sabes qué elemento quieres remover (pero no sabes 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 remover más de un elemento, puedes usar `del` con un índice de rebanado:

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

Como siempre, el rebanado selecciona todos los elementos hasta, pero excluyendo, el segundo índice.

8.8 Listas y funciones

Hay un cierto número funciones internas que pueden ser utilizadas en las listas que te permiten mirar rápidamente a través de una lista sin escribir tus 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
```

La función `sum()` solamente 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 comparados entre sí.

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

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

```
total = 0
contador = 0
while (True):
    inp = input('Ingresa un número: ')
    if inp == 'fin': break
    valor = float(inp)
    total = total + valor
    contador = contador + 1

promedio = total / contador
print('Promedio:', promedio)

# Código: https://es.py4e.com/code3/avenum.py
```

En este programa, tenemos las variables `contador` y `total` para almacenar la cantidad y el total actual de los números del usuario según el usuario va ingresando los números repetidamente.

Podríamos simplemente recordar cada número como el número lo ingresó, y utilizar funciones internas para calcular la suma y el total de números al final.

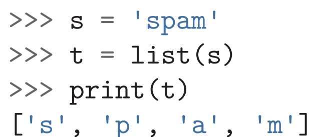
```
numlista = list()
while (True):
    inp = input('Ingresa un número: ')
    if inp == 'fin': break
```

```
    valor = float(inp)
    numlista.append(valor)

promedio = sum(numlista) / len(numlista)
print('Promedio:', promedio)

# Código: https://es.py4e.com/code3/avelist.py
```

Creamos una lista vacía antes de que comience el bucle, y luego cada vez que tengamos 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 total de números en la lista para obtener el promedio.

8.9 Listas y cadenas

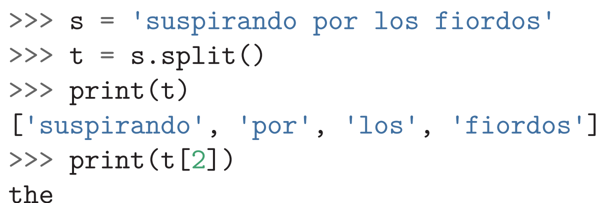
Una cadena es una secuencia de caracteres y una lista es una secuencia de valores, pero una lista de caracteres no es lo mismo que una cadena. Para convertir una cadena en una lista de caracteres, puedes usar `list`:



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

Debido a que `list` es el nombre de una función interna, debes evitar usarla como un nombre de variable. Yo trato de evitar también la letra "l" porque se parece mucho al número "1". Así que por eso utilizo "t".

La función `list` divide una cadena en letras individuales. Si quieres dividir una cadena en palabras, puedes utilizar el método `split`:



```
>>> s = 'suspirando por los fiordos'
>>> t = s.split()
>>> print(t)
['suspirando', 'por', 'los', 'fiordos']
>>> print(t[2])
the
```

Una vez que hayas utilizado `split` para dividir una cadena en una lista de palabras, puedes utilizar el operador índice (corchetes) para ver una palabra en particular en la lista.

Puedes llamar `split` con un argumento opcional llamado *delimitador* que especifica qué caracteres usar para delimitar las palabras. El siguiente ejemplo utiliza un guión medio como delimitador:

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

join es el inverso de split. Este toma una lista de cadenas y concatena los elementos. join es un método de cadenas, así que tienes que invocarlo en el delimitador y pasar la lista como un parámetro:

```
>>> t = ['suspirando', 'por', 'los', 'fiordos']
>>> delimiter = ' '
>>> delimiter.join(t)
'suspirando por los fiordos'
```

En este caso el delimitador es un caracter de espacio, así que join agrega un espacio entre las palabras. Para concatenar cadenas sin espacios, puedes usar la cadena vacía, “”, como delimitador.

8.10 Analizando líneas

Normalmente cuando estamos leyendo un archivo queremos hacer algo con las líneas que no sea solamente imprimir las líneas como son. Frecuentemente queremos encontrar las “líneas interesantes” y después *analizar* la línea para encontrar alguna “parte interesante” en la línea. ¿Qué tal si quisiéramos imprimir el día de la semana de las líneas que comienzan con “From”?

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

El método split es muy efectivo cuando nos encontramos este tipo de problemas. Podemos escribir un pequeño programa que busca líneas donde la línea comienza con “From”, split (dividir) esas líneas, y finalmente imprimir la tercer palabra de la línea:

```
man_a = open('mbox-short.txt')
for linea in man_a:
    linea = linea.rstrip()
    if not linea.startswith('From '): continue
    palabras = linea.split()
    print(palabras[2])

# Código: https://es.py4e.com/code3/search5.py
```

El programa produce lo siguiente:

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

Más tarde, aprenderemos técnicas muy sofisticadas para obtener las líneas que queremos para trajar sobre ellas y cómo sacar el fragmento exacto de información que estamos buscando.

8.11 Objetos y valores

Si ejecutamos las siguientes sentencias de asignación:

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

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

Figure 8.1: Variables y objetos

Por un lado, `a` y `b` se refieren a dos objetos diferentes que tienen el mismo valor. Por otro lado, apuntan al mismo objeto.

Para revisar si dos variables apuntan al mismo objeto, puedes utilizar el operador `is`.

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

En este ejemplo, Python solamente creó un objeto de cadena, y ambos `a` y `b` apuntan a él.

Pero cuando creas dos listas, obtienes dos objetos diferentes:

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

En este caso podríamos decir 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, son también equivalentes, pero si son equivalentes, no son necesariamente idénticos.

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

8.12 Alias

Si `a` se refiere a un objecto y tu asignas`b = a`, entonces ambas variables se refieren al mismo objeto:

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

La asociación de una variable a un objeto es llamada una *referencia*. En este ejemplo, hay dos referencias al mismo objeto.

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

Si el alias del objeto es mutable, los cambios hechos a 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 usar alias cuando estás trabajando con objetos mutables.

Para objetos inmutables como cadenas, los alias no son un problema realmente. En este ejemplo:

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

casi nunca hay diferencia si `a` y `b` apuntan a la misma cadena o no.

8.13 Listas como argumentos

Cuando pasas una lista a una función, la función obtiene un apuntador a la lista. Si la función modifica un parámetro de la lista, el código que ha llamado la función también verá el cambio. Por ejemplo, `remover_primero` elimina el primer elemento de una lista:

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

Aquí está el ejemplo de cómo se usa:

```
>>> letras = ['a', 'b', 'c']
>>> remover_primero(letras)
>>> print(letras)
['b', 'c']
```

El parámetro `t` y la variable `letras` 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]
>>> t2 is t3
False
```

Esta diferencia es importante cuando escribes funciones que no están destinadas a modificar listas. Por ejemplo, esta función *no* elimina el primer elemento de una lista:

```
def mal_eliminar_primero(t):
    t = t[1:]              # ¡EQUIVOCADO!
```

El operador de rebanado crea una nueva lista y el asignamiento hace que `t` apunte a la lista, pero nada de esto tiene efecto en la lista que fue pasada como argumento.

Una alternativa es escribir una función que cree y regrese una nueva lista. Por ejemplo, `cola` regresa todo excepto el primer elemento de una lista:

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

Esta función deja la lista original sin modificar. Aquí está como es que se usa:

```
>>> letras = ['a', 'b', 'c']
>>> resto = cola(letras)
>>> print(resto)
['b', 'c']
```

**Ejercicio 1: Escribe una función llamada `recortar` que toma una lista y la modifica, removiendo el primer y último elemento, y regresa `None`. Después escribe una función llamada `medio` que toma una lista y regresa una nueva lista que contiene todo excepto el primero y último elementos.

8.14 Depuración

El uso descuidado de listas (y otros objetos mutables) puede llevar a largas horas de depuración. Aquí están algumos de los errores más comunes y las formas de evitarlos:

1. No olvides que la mayoría de métodos de listas modifican el argumento y regresan `None`. Esto es lo opuesto a los métodos de cadenas, que regresan una nueva cadena y dejan la original sin modificar.

   Si estás acostumbrado a escribir código de cadenas como este:

   ```
   palabra = palabra.strip()
   ```

   Estás propenso a escribir código de listas como este:

   ```
   t = t.sort()           # ¡EQUIVOCADO!
   ```

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

   Antes de usar métodos y operadores de listas, deberías leer la documentación cuidadosamente y después probarlos en modo interactivo. Los métodos y operadores que las listas comparten con otras secuencias (como cadenas) están documentados en:

   docs.python.org/library/stdtypes.html#common-sequence-operations

   Los métodos y operadores que solamente aplican a secuencias mutables están documentados en:

   docs.python.org/library/stdtypes.html#mutable-sequence-types

2. Elige un estilo y apégate a él.

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

   Para agregar un elemento, puedes utilizar el método `append` o el operador `+`. Pero no olvides que esos también son correctos:

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

   Y esos son incorrectos:

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

   Prueba cada uno de esos ejemplos en modo interactivo para asegurarte que entiendes lo que hacen. Nota que solamente la última provoca un error en tiempo de ejecución (runtime error); los otros tres son válidos, pero hacen la función equivocada.

3. Hacer copias para evitar alias.

   Si quieres utilizar un método como `sort` que modifica el argumento, pero necesitas mantener la lista original también, puedes hacer una copia.

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

   En este ejemplo podrías tambien usar la función interna `sorted`, la cual regresa una lista nueva y ordenada, y deja la original sin modificar. ¡Pero en ese caso deberías evitar usar `sorted` como un nombre de variable!

4. Listas, `split`, y archivos

   Cuando leemos y analizamos archivos, hay muchas oportunidades de encontrar entradas que pueden hacer fallar a nuestro programa, así que es una buena idea revisar el patrón *guardián* cuando escribimos programas que leen a través de un archivo y buscan una “aguja en un pajar”.

   Vamos a revisar nuestro programa que busca por el día de la semana en las líneas que contienen “from” en el archivo“:

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

   Puesto que estamos dividiendo esta línea en palabras, podríamos apañarnos con el uso de `startswith` y simplemente buscar la primer palabra de la línea para determinar si estamos interesados en esa línea o no. Podemos usar `continue` para saltarnos líneas que no tienen “From” como la primer palabra, tal como sigue:

   ```
   manejador = open('mbox-short.txt')
   for linea in manejador:
       palabras = linea.split()
       if palabras[0] != 'From' : continue
       print(palabras[2])
   ```

   Esto se ve mucho más simple y ni siquiera necesitamos hacer `rstrip` para remover el salto de 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 palabras[0] != 'From' : continue
   IndexError: list index out of range
   ```

   De alguna manera funciona y vemos el día de la primer línea (Sat), pero luego el programa falla con un error. ¿Qué fue lo que falló? ¿Qué datos estropearon e hicieron fallar a nuestro elegante, inteligente, y muy Pythónico programa?

   Puedes mirar el código por un largo tiempo y tratar de resolverlo o preguntar a alguien más, pero el método más rápido e inteligente es agregar una sentencia `print`. El mejor lugar para agregar la sentencia “print” es justo

antes de la línea donde el programa falló, e imprimir los datos que parece que causan la falla.

Ahora bien, este método podría generar muchas líneas de salida, pero al menos tendrás inmediatamente alguna pista de cuál es el problema. Así que agregamos un print a la variable `palabras` justo antes de la línea cinco. Incluso podemos agregar un prefijo "Depuración:" a la línea de modo que mantenemos nuestra salida regular separada de la salida de mensajes de depuración.

```
for linea in manejador:
    palabras = line.split()
    print('Depuración:', palabras)
    if palabras[0] != 'From' : continue
    print(palabras[2])
```

Cuando ejecutamos el programa, se generan muchos mensajes de salida en la pantalla, pero al final, vemos nuestra salida de depuración y el mensaje de error, de modo que sabemos qué sucedió justo antes del error.

```
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 palabras[0] != 'From' : continue
IndexError: list index out of range
```

Cada línea de depuración imprime la lista de palabras que obtuvimos cuando la función `split` dividió 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 lo siguiente:

```
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 vacía! Por supuesto, hay "cero palabras" en una lista vacía. ¿Por qué no pensamos en eso cuando estábamos escribiendo el código? Cuando el código busca la primera palabra (`palabras[0]`) para revisar si coincide con "From", obtenemos un error "index out of range" (índice fuera de rango).

Este es, por supuesto, el lugar perfecto para agregar algo de *código guardián* para evitar revisar la primer palabra si la primer palabra no existe. Hay muchas maneras de proteger este código; vamos a optar por revisar el número de palabras que tenemos antes de mirar a la primer palabra:

```
manejador = open('mbox-short.txt')
contador = 0
```

```
for linea in manejador:
    palabras = linea.split()
    # print 'Depuración:', palabras
    if len(palabras) == 0 : continue
    if palabras[0] != 'From' : continue
    print(palabras[2])
```

Primero comentamos la sentencia de depuración en vez de removerla, en caso de que nuestra modificación falle y tengamos que depurar de nuevo. Luego, agregamos una sentencia guardián que revisa si tenemos cero palabras, y si así fuera, utilizamos `continue` para saltarnos a la siguiente línea en el archivo.

Podemos pensar en las dos sentencias `continue` para ayudarnos a redefinir el juego de líneas que son "interesantes" para nosotros y cuáles queremos procesar más. Una línea que no tenga palabras "no es interesante" para nosotros así que saltamos a la siguiente línea. Una línea que no tenga "From" como su primera palabra tampoco nos interesa así que la saltamos.

El programa modificado ejecuta exitosamente, así que quizás es correcto. Nuestra sentencia guardián se asegura de que `palabras[0]` nunca falle, pero quizá no sea suficiente. Cuando estamos programando, siempre debemos pensar, "¿qué podría salir mal?"

Ejercicio 2: Encontrar que línea del programa de arriba no está protegida (método guardián) propiamente. Trata de construir un archivo de texto que cause que el programa falle y después modifica el programa de modo que la línea es propiamente protegida y pruébalo para asegurarte que el programa es capaz de manejar tu nuevo archivo de texto.

Ejercicio 3: Reescribe el código guardián en el ejemplo de arriba sin las dos sentencias `if`. En vez de eso, utiliza una expresión lógica compuesta utilizando el operador lógico `or` con una sola sentencia `if`.

8.15 Glosario

alias Una circunstancia donde dos o más variables apuntan al mismo objeto.

delimitador Un caracter o cadena utilizado para indicar dónde una cadena debe ser dividida.

elemento Uno de los valores en una lista (u otra secuencia); también llamados ítems.

equivalente Que tiene el mismo valor.

idéntico Ser el mismo objeto (lo cual implica equivalencia).

índice Un valor entero que indica un elemento en una lista.

lista Una secuencia de valores.

lista anidada Una lista que es uno de los elementos de otra lista.

objeto Algo a lo que una variable puede referirse. Un objeto tiene un tipo y un valor.

recorrido de lista Acceso secuencial a cada elemento de una lista.

referencia La asociación entre una variable y su valor.

8.16 Ejercicios

Ejercicio 4: Descargar una copia de un archivo www.py4e.com/code3/romeo.txt. Escribir un programa para abrir el archivo *romeo.txt* y leerlo línea por línea. Para cada línea, dividir la línea en una lista de palabras utilizando la función `split`. Para cada palabra, revisar si la palabra ya se encuentra previamente en la lista. Si la palabra no está en la lista, agregarla a la lista. Cuando el programa termine, ordenar e imprimir las palabras resultantes en orden alfabético.

```
Ingresar nombre de archivo: 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: Escribir un programa para leer a través de datos de una bandeja de entrada de correo y cuando encuentres una línea que comience con "From", dividir la línea en palabras utilizando la función `split`. Estamos interesados en quién envió el mensaje, lo cual es la segunda palabra en las líneas que comienzan con From.

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

Tendrás que analizar la línea From e imprimir la segunda palabra de cada línea From, después tendrás que contar el número de líneas From (no incluir From:) e imprimir el total al final. Este es un buen ejemplo de salida con algunas líneas de salida removidas:

```
python fromcuenta.py
Ingresa un nombre de archivo: mbox-short.txt
stephen.marquard@uct.ac.za
louis@media.berkeley.edu
zqian@umich.edu

[...líneas de salida removidas...]

ray@media.berkeley.edu
cwen@iupui.edu
cwen@iupui.edu
cwen@iupui.edu
Hay 27 lineas en el archivo con la palabra From al inicio
```

Ejercicio 6: Reescribe el programa que pide al usuario una lista de números e imprime el máximo y el mínimo de los números al final cuando el usuario ingresa "hecho". Escribe el programa para almacenar los números que el usuario ingrese en una lista, y utiliza las funciones `max()` y `min()` para calcular el máximo y el mínimo después de que el bucle termine.

```
Ingresa un número: 6
Ingresa un número: 2
Ingresa un número: 9
Ingresa un número: 3
Ingresa un número: 5
Ingresa un número: hecho
Máximo: 9.0
Minimo: 2.0
```