Python para todos

Explorando la información con Python 3

Charles R. Severance

Créditos

Soporte Editorial: Elliott Hauser, Sue Blumenberg
Diseño de portada: Aimee Andrion

Historial de impresión

- 05-Jul-2016 Primera versión completa de Python 3.0
- 20-Dic-2015 Borrador inicial de conversión a Python 3.0

Detalles de Copyright

Prólogo

Remezclando un Libro Libre

Se suele decir de los académicos deben "publicar o perecer" continuamente, de modo que es bastante normal que siempre quieran empezar algo desde cero, para que sea su propia y flamante creación. Este libro es un experimento, ya que no parte desde cero, sino que en vez de eso "remezcla" el libro titulado *Think Python: How to Think Like a Computer Scientist* (Piensa en Python: Cómo pensar como un científico de la computación), escrito por Allen B. Bowney, Jeff Elkner, y otros.

En Diciembre de 2009, yo me estaba preparando para enseñar *SI502 - Programación en Red* en la Universidad de Michigan por quinto semestre consecutivo, y decidí que ya era hora de escribir un libro de texto sobre Python que se centrase en la exploración de datos en lugar de en explicar algoritmos y abstracciones. Mi objetivo en SI502 es enseñar a la gente habilidades permanentes para el manejo de datos usando Python. Pocos de mis estudiantes pretenden llegar a ser programadores de computadoras profesionales. En vez de eso, quieren ser bibliotecarios, gerentes, abogados, biólogos, economistas, etc., que tal vez quieran aplicar el uso de la tecnología en sus respectivos campos.

Parecía que no podría encontrar el libro perfecto para mi curso, que estuviera orientado al manejo de datos en Python, de modo que decidí empezar a escribirlo por mi mismo. Por suerte, en una reunión de profesores tres semanas antes de las vacaciones, que era la fecha en que tenía planeado empezar a escribir mi libro desde cero, el Dr. Atul Prakash me mostró el libro *Think Python* (Piensa en Python), que él había utilizado para impartir su curso de Python ese semestre. Se trata de un texto de Ciencias de la Computación bien escrito, con un enfoque breve, explicaciones directas y fácil de aprender.

La estructura principal del libro se ha cambiado, para empezar a realizar problemas de análisis de datos lo antes posible, y para tener una serie de ejemplos funcionales y de ejercicios sobre el análisis de datos desde el principio.

Los capítulos 2-10 son similares a los del libro *Think Python*, pero ha habido cambios importantes. Los ejemplos orientados a números y los ejercicios se han reemplazado por otros orientados a datos. Los temas se presentan en el orden necesario para ir creando soluciones de análisis de datos cuya complejidad aumente progresivamente. Algunos temas como `try` y `except` (manejo de excepciones) se han adelantado, y se presentan como parte del capítulo de los condicionales. Las funciones se tratan muy por encima hasta que son necesarias para manejar programas complejos, en lugar de introducirlas como abstracción en las primeras lecciones. Casi todas las funciones definidas por el usuario se han eliminado del código de los ejemplos y de los ejercicios excepto en el capítulo 4. La palabra "recursión"[1] no aparece en todo el libro.

Todo el contenido del capítulo 1 y del 11 al 16 es nuevo, centrado en aplicaciones para el mundo real y en ejemplos simples del uso de Python para el análisis de datos, incluyendo expresiones regulares para búsqueda y análisis, automatización de tareas en la computadora, descarga de datos a través de la red, escaneo de páginas web para recuperar datos, programación orientada a objetos, uso de servicios

[1] Excepto, por supuesto, en esa línea.

web, análisis de datos en formato XML y JSON, creación y uso de bases de datos usando el Lenguaje de Consultas Estructurado (SQL), y la visualización de datos.

El objetivo final de todos estos cambios es variar la orientación, desde una dirigida a las Ciencias de la Computación hacia otra puramente informática, que trate sólo temas adecuados para una clase de tecnología para principiantes, que puedan resultarles útiles incluso si eligen no ser programadores profesionales.

Los estudiantes que encuentren este libro interesante y quieran ir más allá, deberían echar un vistazo al libro *Think Python* de Allen B. Downey's. Como ambos libros comparten un montón de materia, los estudiantes adquirirán rápidamente habilidades en las áreas adicionales de la programación técnica y pensamiento algorítmico que se tratan en *Think Python*. Y dado que ambos libros comparten un estilo de escritura similar, deberían ser capaces de avanzar rápidamente a través del contenido de *Think Python* con un esfuerzo mínimo.

Como propietario del copyright de *Think Python*, Allen me ha dado permiso para cambiar la licencia del contenido de su libro que se utiliza en éste, y que originalmente poseía una *GNU Free Documentation License* a otra más actual, Creative Commons Attribution — Share Alike license. Así se sigue una tendencia general en las licencias de documentación abierta, que están pasando desde la GFDL a la CC-BY-SA (por ejemplo, Wikipedia). El uso de la licencia CC-BY-SA mantiene la arraigada tradición *copyleft* del libro, a la vez que hacen más sencillo para los autores nuevos la reutilización de ese material a su conveniencia.

Personalmente creo que este libro sirve como ejemplo de por qué los contenidos libres son tan importantes para el futuro de la educación, y quiero agradecer a Allen B. Downey y a la *Cambridge University Press* por su amplitud de miras a la hora de distribuir el libro bajo un copyright abierto. Espero que se sientan satisfechos con el resultado de mis esfuerzos y deseo que tú como lector también te sientas satisfecho de *nuestros* esfuerzos colectivos.

Quiero agradecer a Allen B. Downey y Lauren Cowles su ayuda, paciencia y orientación a la hora de tratar y resolver los problemas de copyright referentes a este libro.

Charles Severance
www.dr-chuck.com
Ann Arbor, MI, USA
9 de Septiembre, 2013

Charles Severance es Profesor Clínico Adjunto en la Escuela de Información (*School of Information*) de la Universidad de Michigan.

Contents

Chapter 1

¿Por qué deberías aprender a escribir programas?

Escribir programas (o programar) es una actividad muy creativa y gratificante. Puedes escribir programas por muchas razones, que pueden ir desde mantenerte activo resolviendo un problema de análisis de datos complejo hasta hacerlo por pura diversión ayudando a otros a resolver un enigma. Este libro asume que *todo el mundo* necesita saber programar, y que una vez que aprendas a programar ya encontrarás qué quieres hacer con esas habilidades recién adquiridas.

En nuestra vida diaria estamos rodeados de computadoras, desde equipos portátiles (laptops) hasta teléfonos móviles (celulares). Podemos pensar en esas computadoras como nuestros "asistentes personales", que pueden ocuparse de muchas tareas por nosotros. El hardware en los equipos que usamos cada día está diseñado esencialmente para hacernos la misma pregunta de forma constante, "¿Qué quieres que haga ahora?"

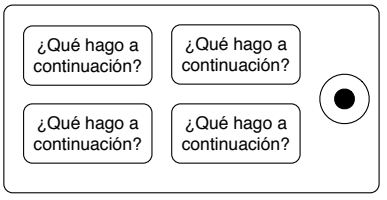

Figure 1.1: Personal Digital Assistant

Los programadores suelen añadir un sistema operativo y un conjunto de aplicaciones al hardware y así nos proporcionan un Asistente Digital Personal que es bastante útil y capaz de ayudarnos a realizar una gran variedad de tareas.

Nuestros equipos son rápidos y tienen grandes cantidades de memoria. Podrían resultarnos muy útiles si tan solo supiéramos qué idioma utilizar para explicarle a la computadora qué es lo que queremos que "haga ahora". Si conociéramos ese idioma, podríamos pedirle al aparato que realizase en nuestro lugar, por ejemplo, tareas repetitivas. Precisamente el tipo de cosas que las computadoras saben hacer mejor suelen ser el tipo de cosas que las personas encontramos pesadas y aburridas.

Por ejemplo, mira los primeros tres párrafos de este capítulos y dime cuál es la palabra que más se repite, y cuántas veces se ha utilizado. Aunque seas capaz de leer y comprender las palabras en pocos segundos, contarlas te resultará casi doloroso, porque la mente humana no fue diseñada para resolver ese tipo de problemas. Para una computadora es justo al revés, leer y comprender texto de un trozo de papel le sería difícil, pero contar las palabras y decirte cuántas veces se ha repetido la más utilizada le resulta muy sencillo:

```
python words.py
Enter file:words.txt
to 16
```

Nuestro "asistente de análisis de información personal" nos dirá enseguida que la palabra "que" se usó nueve veces en los primeros tres párrafos de este capítulo.

El hecho de que los computadores sean buenos en aquellas cosas en las que los humanos no lo son es el motivo por el que necesitas aprender a hablar el "idioma de las computadoras". Una vez que aprendas este nuevo lenguaje, podrás delegar tareas mundanas a tu compañero (la computadora), lo que te dejará más tiempo para ocuparte de las cosas para las que sólo tú estás capacitado. Tú pondrás la creatividad, intuición y el ingenio en esa alianza.

1.1 Creatividad y motivación

A pesar de que este libro no va dirigido a los programadores profesionales, la programación a nivel profesional puede ser un trabajo muy gratificante, tanto a nivel financiero como personal. Crear programas útiles, elegantes e inteligentes para que los usen otros, es una actividad muy creativa. Tu computadora o Asistente Digital Personal (PDA[1]), normalmente contiene muchos programas diferentes pertenecientes a distintos grupos de programadores, cada uno de ellos compitiendo por tu atención e interés. Todos ellos hacen su mejor esfuerzo por adaptarse a tus necesidades y proporcionarte una experiencia de usuario satisfactoria. En ocasiones, cuando elijes un software determinado, sus programadores son directamente recompensados gracias a tu elección.

Si pensamos en los programas como el producto de la creatividad de los programadores, tal vez la figura siguiente sea una versión más acertada de nuestra PDA:

Figure 1.2: Programadores Dirigiéndose a Ti

Por ahora, nuestra principal motivación no es conseguir dinero ni complacer a los usuarios finales, sino simplemente conseguir ser más productivos a nivel personal

[1] Personal Digital Assistant en inglés (N. del T.).

en el manejo de datos e información que encontremos en nuestras vidas. Cuando se empieza por primera vez, uno es a la vez programador y usuario final de sus propios programas. A medida que se gana habilidad como programador, y la programación se hace más creativa para uno mismo, se puede empezar a pensar en desarrollar programas para los demás.

1.2 Arquitectura hardware de las computadoras

Antes de que empecemos a aprender el lenguaje que deberemos hablar para darle instrucciones a las computadoras para desarrollar software, tendremos que aprender un poco acerca de cómo están construidos esas máquinas. Si desmontaras tu computadora o *smartphone* y mirases dentro con atención, encontrarías los siguientes componentes:

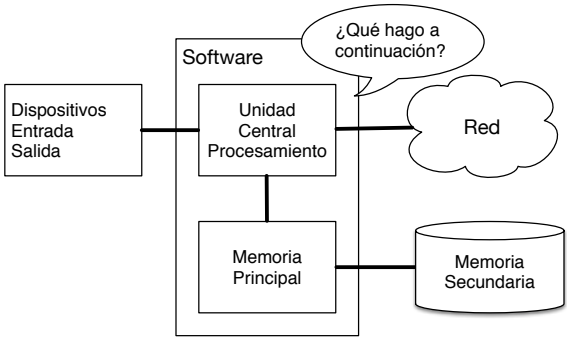

Figure 1.3: Arquitectura Hardware

Las definiciones de alto nivel de esos componentes son las siguientes:

- La *Unidad Central de Procesamiento* (o CPU[2]) es el componente de la computadora diseñado para estar obsesionado con el "¿qué hago ahora?". Si tu equipo está dentro de la clasificación de 3.0 Gigahercios, significa que la CPU preguntará "¿Qué hago ahora?" tres mil millones de veces por segundo. Vas a tener que aprender a hablar muy rápido para mantener el ritmo de la CPU.

- La *Memoria Principal* se usa para almacenar la información que la CPU necesita de forma inmediata. La memoria principal es casi tan rápida como la CPU. Pero la información almacenada en la memoria principal desaparece cuando se apaga el equipo.

- La *Memoria Secundaria* también se utiliza para almacenar información, pero es mucho más lenta que la memoria principal. La ventaja de la memoria secundaria es que puede almacenar la información incluso cuando el equipo está apagado. Algunos ejemplos de memoria secundaria serían las unidades de disco o las memorias flash (que suelen encontrarse en los *pendrives* USB y en los reproductores de música portátiles).

[2]Central Processing Unit en inglés (N. del T.).

- Los *Dispositivos de Entrada y Salida* son simplemente la pantalla, teclado, ratón, micrófono, altavoz, *touchpad*, etc. Incluyen cualquier modo de interactuar con una computadora.
- Actualmente, casi todos los equipos tienen una *Conexión de Red* para recibir información dentro de una red. Podemos pensar en una red como en un lugar donde almacenar y recuperar datos de forma muy lenta, que puede no estar siempre "activo". Así que, en cierto sentido, la red no es más que un tipo de *Memoria Secundaria* más lenta y a veces poco fiable.

Aunque la mayoría de los detalles acerca de cómo funcionan estos componentes es mejor dejársela a los constructores de equipos, resulta útil disponer de cierta terminología para poder referirnos a ellos a la hora de escribir nuestros programas.

Como programador, tu trabajo es usar y orquestar cada uno de esos recursos para resolver el problema del que tengas que ocuparte y analizar los datos de los que dispongas para encontrar la solución. Como programador estarás casi siempre "hablando" con la CPU y diciéndole qué es lo siguiente que debe hacer. A veces le tendrás que pedir a la CPU que use la memoria principal, la secundaria, la red, o los dispositivos de entrada/salida.

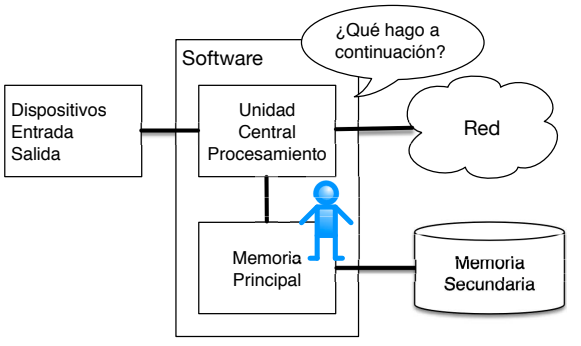

Figure 1.4: ¿Dónde estás?

Tú deberás ser la persona que responda a la pregunta "¿Qué hago ahora?" de la CPU. Pero sería muy incómodo encogerse uno mismo hasta los 5 mm. de altura e introducirse dentro de la computadora sólo para poder dar una orden tres mil millones de veces por segundo. Así que en vez de eso, tendrás que escribir las instrucciones por adelantado. Esas instrucciones almacenadas reciben el nombre de *programa* y el acto de escribirlas y encontrar cuáles son las instrucciones adecuadas, *programar*.

1.3 Comprendiendo la programación

En el resto de este libro, intentaremos convertirte en una persona experta en el arte de programar. Al terminar, te habrás convertido en un *programador* - tal vez no uno profesional, pero al menos tendrás la capacidad de encarar un problema de análisis de datos/información y desarrollar un programa para resolverlo.

En cierto modo, necesitas dos capacidades para ser programador:

- Primero, necesitas saber un lenguaje de programación (Python) - debes conocer su vocabulario y su gramática. Debes ser capaz de deletrear correctamente las palabras en ese nuevo lenguaje y saber construir "frases" bien formadas.
- Segundo, debes "contar una historia". Al escribir un relato, combinas palabras y frases para comunicar una idea al lector. Hay una cierta técnica y arte en la construcción de un relato, y la habilidad para escribir relatos mejora escribiendo y recibiendo cierta respuesta. En programación, nuestro programa es el "relato" y el problema que estás tratando de resolver es la "idea".

Una vez que aprendas un lenguaje de programación como Python, encontrarás mucho más fácil aprender un segundo lenguaje como JavaScript o C++. Cada nuevo lenguaje tiene un vocabulario y gramática muy diferentes, pero la técnica de resolución de problemas será la misma en todos ellos.

Aprenderás el "vocabulario" y "frases" de Python bastante rápido. Te llevará más tiempo el ser capaz de escribir un programa coherente para resolver un problema totalmente nuevo. Se enseña programación de forma muy similar a como se enseña a escribir. Se empieza leyendo y explicando programas, luego se escriben programas sencillos, y a continuación se van escribiendo programas progresivamente más complejos con el tiempo. En algún momento "encuentras tu musa", empiezas a descubrir los patrones por ti mismo y empiezas a ver casi de forma instintiva cómo abordar un problema y escribir un programa para resolverlo. Y una vez alcanzado ese punto, la programación se convierte en un proceso muy placentero y creativo.

Comenzaremos con el vocabulario y la estructura de los programas en Python. Ten paciencia si la simplicidad de los ejemplos te recuerda a cuando aprendiste a leer.

1.4 Palabras y frases

A diferencia de los lenguajes humanos, el vocabulario de Python es en realidad bastante reducido. Llamamos a este "vocabulario" las "palabras reservadas". Se trata de palabras que tienen un significado muy especial para Python. Cuando Python se encuentra estas palabras en un programa, sabe que sólo tienen un único significado para él. Más adelante, cuando escribas programas, podrás usar tus propias palabras con significado, que reciben el nombre de *variables*. Tendrás gran libertad a la hora de elegir los nombres para tus variables, pero no podrás utilizar ninguna de las palabras reservadas de Python como nombre de una variable.

Cuando se entrena a un perro, se utilizan palabras especiales como "siéntate", "quieto" y "tráelo". Cuando te diriges a un perro y no usas ninguna de las palabras reservadas, lo único que consigues es que se te quede mirando con cara extrañada, hasta que le dices una de las palabras que reconoce. Por ejemplo, si dices, "Me gustaría que más gente saliera a caminar para mejorar su salud general", lo que la mayoría de los perros oirían es: "bla bla bla *caminar* bla bla bla bla.". Eso se debe a que "caminar" es una palabra reservada en el lenguaje del perro. Seguramente habrá quien apunte que el lenguaje entre humanos y gatos no dispone de palabras reservadas[3].

[3] http://xkcd.com/231/

Las palabras reservadas en el lenguaje que utilizan los humanos para hablar con Python son, entre otras, las siguientes:

```
and       del       global      not       with
as        elif      if          or        yield
assert    else      import      pass
break     except    in          raise
class     finally   is          return
continue  for       lambda      try
def       from      nonlocal    while
```

Es decir, a diferencia de un perro, Python ya está completamente entrenado. Cada vez le digas "inténtalo", Python lo intentará una vez tras otra sin desfallecer[4].

Aprenderemos cuáles son las palabras reservadas y cómo utilizarlas en su momento, pero por ahora nos centraremos en el equivalente en Python de "habla" (en el lenguaje humano-perro). Lo bueno de pedirle a Python que hable es que podemos incluso indicarle lo que debe decir, pasándole un mensaje entre comillas:

```
print('¡Hola, mundo!')
```

Y ya acabamos de escribir nuestra primera oración sintácticamente correcta en Python. La frase comienza con la función *print* seguida de la cadena de texto que hayamos elegido dentro de comillas simples. Las comillas simples y dobles cumplen la misma función; la mayoría de las personas usan las comillas simples, excepto cuando la cadena de texto contiene también una comilla simple (que puede ser un apóstrofo).

1.5 Conversando con Python

Ahora que ya conocemos una palabra y sabemos cómo crear una frase sencilla en Python, necesitamos aprender a iniciar una conversación con él para comprobar nuestras nuevas capacidades con el lenguaje.

Antes de que puedas conversar con Python, deberás instalar el software necesario en tu computadora y aprender a iniciar Python en ella. En este capítulo no entraremos en detalles sobre cómo hacerlo, pero te sugiero que consultes https://es.py4e.com/ , donde encontrarás instrucciones detalladas y capturas sobre cómo configurar e iniciar Python en sistemas Macintosh y Windows. Si sigues los pasos, llegará un momento en que te encuentres ante una ventana de comandos o terminal. Si escribes entonces *python*, el intérprete de Python empezará a ejecutarse en modo interactivo, y aparecerá algo como esto:

```
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  6 2015, 01:54:25)
[MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

[4] En inglés "inténtalo" es "try", que es también una palabra reservada dentro del lenguaje Python (N. del T.).

El indicador >>> es el modo que tiene el intérprete de Python de preguntarte, "¿Qué quieres que haga ahora?". Python está ya preparado para mantener una conversación contigo. Todo lo que tienes que saber es cómo hablar en su idioma.

Supongamos por ejemplo que aún no conoces ni las palabras ni frases más sencillas de Python. Puede que quieras utilizar el método clásico de los astronautas cuando aterrizan en un planeta lejano e intentan hablar con los habitantes de ese mundo:

```
>>> Vengo en son de paz, por favor llévame ante tu líder
  File "<stdin>", line 1
    Vengo en son de paz, por favor llévame ante tu líder
          ^
SyntaxError: invalid syntax
>>>
```

Esto no se ve bien. A menos que pienses en algo rápidamente, los habitantes del planeta sacarán sus lanzas, te ensartarán, te asarán sobre el fuego y al final les servirás de cena.

Por suerte compraste una copia de este libro durante tus viajes, así que lo abres precisamente por esta página y pruebas de nuevo:

```
>>> print('¡Hola, mundo!')
¡Hola, mundo!
```

Esto tiene mejor aspecto, de modo que intentas comunicarte un poco más:

```
>>> print('Usted debe ser el dios legendario que viene del cielo')
Usted debe ser el dios legendario que viene del cielo
>>> print('Hemos estado esperándole durante mucho tiempo')
Hemos estado esperándole durante mucho tiempo
>>> print('La leyenda dice que debe estar usted muy rico con mostaza')
La leyenda dice que debe estar usted muy rico con mostaza
>>> print 'Tendremos un festín esta noche a menos que diga
  File "<stdin>", line 1
    print 'Tendremos un festín esta noche a menos que diga
                                                         ^
SyntaxError: Missing parentheses in call to 'print'
>>>
```

La conversación fue bien durante un rato, pero en cuanto cometiste el más mínimo fallo al utilizar el lenguaje Python, Python volvió a sacar las lanzas.

En este momento, te habrás dado cuenta que a pesar de que Python es tremendamente complejo y poderoso, y muy estricto en cuanto a la sintaxis que debes usar para comunicarte con él, Python *no* es inteligente. En realidad estás solamente manteniendo una conversación contigo mismo; eso sí, usando una sintaxis adecuada.

En cierto modo, cuando utilizas un programa escrito por otra persona, la conversación se mantiene entre tú y el programador, con Python actuando meramente de

intermediario. Python es una herramienta que permite a los creadores de programas expresar el modo en que la conversación supuestamente debe fluir. Y dentro de unos pocos capítulos más, serás uno de esos programadores que utilizan Python para hablar con los usuarios de tu programa.

Antes de que abandonemos nuestra primera conversación con el intérprete de Python, deberías aprender cual es el modo correcto de decir "adiós" al interactuar con los habitantes del Planeta Python:

```
>>> adiós
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'adiós' is not defined
>>> if you don't mind, I need to leave\footnote{si no te importa, tengo que marcha
  File "<stdin>", line 1
    if you don't mind, I need to leave
              ^
SyntaxError: invalid syntax
>>> quit()
```

Te habrás fijado en que el error es diferente en cada uno de los dos primeros intentos. El segundo error es diferente porque *if* es una palabra reservada, y cuando Python la ve, cree que estamos intentando decirle algo, pero encuentra la sintaxis de la frase incorrecta.

La forma correcta de decirle "adiós" a Python es introducir *quit()* en el símbolo indicador del sistema >>>. Seguramente te hubiera llevado un buen rato adivinarlo, así que tener este libro a mano probablemente te haya resultado útil.

1.6 Terminología: intérprete y compilador

Python es un lenguaje *de alto nivel*, pensado para ser relativamente sencillo de leer y escribir para las personas, y fácil de leer y procesar para las máquinas. Otros lenguajes de alto nivel son Java, C++, PHP, Ruby, Basic, Perl, JavaScript, y muchos más. El hardware real que está dentro de la Unidad Central de Procesamiento (CPU), no entiende ninguno de esos lenguajes de alto nivel.

La CPU entiende únicamente un lenguaje llamado *lenguaje de máquina* o *código máquina*. El código máquina es muy simple y francamente muy pesado de escribir, ya que está representado en su totalidad por solamente ceros y unos:

```
00101000111010010010101010000001111
11100110000011101010010101101101
...
```

El código máquina parece bastante sencillo a simple vista, dado que sólo contiene ceros y unos, pero su sintaxis es incluso más compleja y mucho más enrevesada que la de Python, razón por la cual muy pocos programadores escriben en código máquina. En vez de eso, se han creado varios programas traductores para permitir a los programadores escribir en lenguajes de alto nivel como Python o Javascript,

y son esos traductores quienes convierten los programas a código máquina, que es lo que ejecuta en realidad la CPU.

Dado que el código máquina está ligado al hardware de la máquina que lo ejecuta, ese código no es *portable* (trasladable) entre equipos de diferente tipo. Los programas escritos en lenguajes de alto nivel pueden ser trasladados entre distintas máquinas usando un intérprete diferente en cada una de ellas, o recompilando el código para crear una versión diferente del código máquina del programa para cada uno de los tipos de equipo.

Esos traductores de lenguajes de programación forman dos categorías generales: (1) intérpretes y (2) compiladores.

Un *intérprete* lee el código fuente de los programas tal y como ha sido escrito por el programador, lo analiza, e interpreta sus instrucciones sobre la marcha. Python es un intérprete y cuando lo estamos ejecutando de forma interactiva, podemos escribir una línea de Python (una frase), y este la procesa de forma inmediata, quedando listo para que podamos escribir otra línea.

Algunas de esas líneas le indican a Python que tú quieres que recuerde cierto valor para utilizarlo más tarde. Tenemos que escoger un nombre para que ese valor sea recordado y usaremos ese nombre simbólico para recuperar el valor más tarde. Utilizamos el término *variable* para denominar las etiquetas que usamos para referirnos a esos datos almacenados.

```
>>> x = 6
>>> print(x)
6
>>> y = x * 7
>>> print(y)
42
>>>
```

En este ejemplo, le pedimos a Python que recuerde el valor seis y use la etiqueta x para que podamos recuperar el valor más tarde. Comprobamos que Python ha guardado de verdad el valor usando *print*. Luego le pedimos a Python que recupere x, lo multiplique por siete y guarde el valor calculado en y. Finalmente, le pedimos a Python que escriba el valor actual de y.

A pesar de que estamos escribiendo estos comandos en Python línea a línea, Python los está tratando como una secuencia ordenada de sentencias, en la cual las últimas frases son capaces de obtener datos creados en las anteriores. Estamos, por tanto, escribiendo nuestro primer párrafo sencillo con cuatro frases en un orden lógico y útil.

La esencia de un *intérprete* consiste en ser capaz de mantener una conversación interactiva como la mostrada más arriba. Un *compilador* necesita que le entreguen el programa completo en un fichero, y luego ejecuta un proceso para traducir el código fuente de alto nivel a código máquina, tras lo cual coloca ese código máquina resultante dentro de otro fichero para su ejecución posterior.

En sistemas Windows, a menudo esos ejecutables en código máquina tienen un sufijo o extensión como ".exe" o ".dll", que significan "ejecutable" y "librería de

enlace dinámico" (*dynamic link library*) respectivamente. En Linux y Macintosh, no existe un sufijo que identifique de manera exclusiva a un fichero como ejecutable.

Si abrieras un fichero ejecutable con un editor de texto, verías algo complemente disparatado e ilegible:

```
^?ELF^A^A^A^@^@^@^@^@^@^@^@^B^@^C^@^A^@^@^@\xa0\x82
^D^H4^@^@^@\x90^]^@^@^@^@^@^@4^@ ^@^G^@(^@$^@!^@^F^@
^@^@4^@^@^@4\x80^D^H4\x80^D^H\xe0^@^@^@\xe0^@^@^@^E
^@^@^@^D^@^@^@^C^@^@^@^T^A^@^@^T\x81^D^H^T\x81^D^H^S
^@^@^@^S^@^@^@^D^@^@^@^A^@^@^@^A\^D^HQVhT\x83^D^H\xe8
....
```

No resulta fácil leer o escribir código máquina, pero afortunadamente disponemos de *intérpretes* y *compiladores* que nos permiten escribir en lenguajes de alto nivel, como Python o C.

Llegados a este punto en la explicación acerca de los compiladores e intérpretes, seguramente te estarás preguntando algunas cosas acerca del propio intérprete de Python. ¿En qué lenguaje está escrito? ¿Está escrito en un lenguaje compilado? Cuando escribimos "python", ¿qué ocurre exactamente?

El intérprete de Python está escrito en un lenguaje de alto nivel llamado "C". En realidad, puedes ver el código fuente del intérprete de Python acudiendo a www.python.org e incluso modificarlo a tu gusto. Quedamos, pues, en que Python es en sí mismo un programa y que está compilado en código máquina. Cuando instalaste Python en tu computadora (o el vendedor lo instaló), colocaste una copia del código máquina del programa Python traducido para tu sistema. En Windows, el código máquina ejecutable para el intérprete de Python es probablemente un fichero con un nombre como:

```
C:\Python35\python.exe
```

En realidad, eso ya es más de lo que necesitas saber para ser un programador en Python, pero algunas veces vale la pena contestar estas pequeñas dudas recurrentes justo al principio.

1.7 Escribiendo un programa

Escribir comandos en el intérprete de Python es un buen modo de experimentar con las capacidades de Python, pero no es lo más recomendado a la hora de resolver problemas más complejos.

Cuando queremos escribir un programa, usamos un editor de texto para escribir las instrucciones de Python en un fichero, que recibe el nombre de *script*. Por convención, los scripts de Python tienen nombres que terminan en `.py`.

Para ejecutar el script, hay que indicarle al intérprete de Python el nombre del fichero. En una ventana de comandos de Unix o Windows, escribirías `python hola.py`, de este modo:

```
csev$ cat hola.py
print('¡Hola, mundo!')
csev$ python hola.py
¡Hola, mundo!
csev$
```

"csev$" es el indicador (*prompt*) del sistema operativo, y el comando "cat hola.py" nos muestra que el archivo "hola.py" contiene un programa con una única línea de código que imprime en pantalla una cadena de texto.

Llamamos al intérprete de Python y le pedimos que lea el código fuente desde el archivo "hola.py", en vez de esperar a que vayamos introduciendo líneas de código Python de forma interactiva.

Habrás notado que cuando trabajamos con un fichero no necesitamos incluir el comando *quit()* al final del programa Python. Cuando Python va leyendo tu código fuente desde un archivo, sabe que debe parar cuando llega al final del fichero.

1.8 ¿Qué es un programa?

Podemos definir un *programa*, en su forma más básica, como una secuencia de declaraciones o sentencias que han sido diseñadas para hacer algo. Incluso nuestro sencillo script "hola.py" es un programa. Es un programa de una sola línea y no resulta particularmente útil, pero si nos ajustamos estrictamente a la definición, se trata de un programa en Python.

Tal vez resulte más fácil comprender qué es un programa pensando en un problema que pudiera ser resuelto a través de un programa, y luego estudiando cómo sería el programa que solucionaría ese problema.

Supongamos que estás haciendo una investigación de computación o informática social en mensajes de Facebook, y te interesa conocer cual es la palabra más utilizada en un conjunto de mensajes. Podrías imprimir el flujo de mensajes de Facebook y revisar con atención el texto, buscando la palabra más común, pero sería un proceso largo y muy propenso a errores. Sería más inteligente escribir un programa en Python para encargarse de la tarea con rapidez y precisión, y así poder emplear el fin de semana en hacer otras cosas más divertidas.

Por ejemplo, fíjate en el siguiente texto, que trata de un payaso y un coche. Estúdialo y trata de averiguar cual es la palabra más común y cuántas veces se repite.

```
el payaso corrió tras el coche y el coche se metió dentro de la tienda
y la tienda cayó sobre el payaso y el coche
```

Ahora imagina que haces lo mismo pero buscando a través de millones de líneas de texto. Francamente, tardarías menos aprendiendo Python y escribiendo un programa en ese lenguaje para contar las palabras que si tuvieras que ir revisando todas ellas una a una.

Pero hay una noticia aún mejor, y es que se me ha ocurrido un programa sencillo para encontrar cuál es la palabra más común dentro de un fichero de texto. Ya lo escribí, lo probé, y ahora te lo regalo para que lo puedas utilizar y ahorrarte mucho tiempo.

```
name = input('Enter file:')
handle = open(name, 'r')
counts = dict()

for line in handle:
    words = line.split()
    for word in words:
        counts[word] = counts.get(word, 0) + 1

bigcount = None
bigword = None
for word, count in list(counts.items()):
    if bigcount is None or count > bigcount:
        bigword = word
        bigcount = count

print(bigword, bigcount)

# Code: http://www.py4e.com/code3/words.py
```

No necesitas ni siquiera saber Python para usar este programa. Tendrás que llegar hasta el capítulo 10 de este libro para entender por completo las impresionantes técnicas de Python que se han utilizado para crearlo. Ahora eres el usuario final, sólo tienes que usar el programa y sorprenderte de sus habilidades y de cómo te permite ahorrar un montón de esfuerzo. Tan sólo tienes que escribir el código dentro de un fichero llamado *words.py* y ejecutarlo, o puedes descargar el código fuente directamente desde https://es.py4e.com/code3/ y ejecutarlo.

Este es un buen ejemplo de cómo Python y el lenguaje Python actúan como un intermediario entre tú (el usuario final) y yo (el programador). Python es un medio para que intercambiemos secuencias de instrucciones útiles (es decir, programas) en un lenguaje común que puede ser usado por cualquiera que instale Python en su computadora. Así que ninguno de nosotros está hablando *con Python*, sino que estamos comunicándonos uno con el otro *a través de* Python.

1.9 Los bloques de construcción de los programas

En los próximos capítulos, aprenderemos más sobre el vocabulario, la estructura de las frases y de los párrafos y la estructura de los relatos en Python. Aprenderemos cuáles son las poderosas capacidades de Python y cómo combinar esas capacidades entre sí para crear programas útiles.

Hay ciertos patrones conceptuales de bajo nivel que se usan para estructurar los programas. Esas estructuras no son exclusivas de Python, sino que forman parte de cualquier lenguaje de programación, desde el código máquina hasta los lenguajes de alto nivel.

entrada Obtener datos del "mundo exterior". Puede consistir en leer datos desde un fichero, o incluso desde algún tipo de sensor, como un micrófono o un GPS.

En nuestros primeros programas, las entradas van a provenir del usuario, que introducirá los datos a través del teclado.

salida Mostrar los resultados del programa en una pantalla, almacenarlos en un fichero o incluso es posible enviarlos a un dispositivo como un altavoz para reproducir música o leer un texto.

ejecución secuencial Ejecutar una sentencia tras otra en el mismo orden en que se van encontrando en el *script*.

ejecución condicional Comprobar ciertas condiciones y luego ejecutar u omitir una secuencia de sentencias.

ejecución repetida Ejecutar un conjunto de sentencias varias veces, normalmente con algún tipo de variación.

reutilización Escribir un conjunto de instrucciones una vez, darles un nombre y así poder reutilizarlas luego cuando se necesiten en cualquier punto de tu programa.

Parece demasiado simple para ser cierto, y por supuesto nunca es tan sencillo. Es como si dijéramos que andar es simplemente "poner un pie delante del otro". El "arte" de escribir un programa es componer y entrelazar juntos esos elementos básicos muchas veces hasta conseguir al final algo que resulte útil para sus usuarios.

El programa para contar palabras que vimos antes utiliza al mismo tiempo todos esos patrones excepto uno.

1.10 ¿Qué es posible que vaya mal?

Como vimos en nuestra anterior conversación con Python, debemos comunicarnos con mucha precisión cuando escribimos código Python. El menor error provocará que Python se niegue a hacer funcionar tu programa.

Los programadores novatos a menudo se toman el hecho de que Python no permita cometer errores como la prueba definitiva de que es perverso, odioso y cruel. A pesar de que a Python parece gustarle todos los demás, es capaz de identificar a los novatos en concreto, y les guarda un gran rencor. Debido a ello, toma sus programas perfectamente escritos, y los rechaza, considerándolos como "inservibles", sólo para atormentarlos.

```
>>> primt '¡Hola, mundo!'
  File "<stdin>", line 1
    primt '¡Hola, mundo!'
                ^
SyntaxError: invalid syntax
>>> primt ('Hola, mundo')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'primt' is not defined

>>> ¡Te odio, Python!
  File "<stdin>", line 1
    ¡Te odio, Python!
     ^
```

```
SyntaxError: invalid syntax
>>> si sales fuera, te daré una lección
  File "<stdin>", line 1
    si sales fuera, te daré una lección
               ^
SyntaxError: invalid syntax
>>>
```

No es mucho lo que se gana discutiendo con Python. Solo es una herramienta. No tiene emociones, es feliz y está preparado para servirte en el momento que lo necesites. Sus mensajes de error parecen crueles, pero simplemente se trata de una petición de ayuda del propio Python. Ha examinado lo que has tecleado, y sencillamente no es capaz de entender lo que has escrito.

Python se parece mucho a un perro, te quiere incondicionalmente, entiende algunas pocas palabras clave, te mira con una mirada dulce en su cara(>>>), y espera que le digas algo que él pueda comprender. Cuando Python dice "SyntaxError: invalid syntax" (Error de sintaxis: sintaxis inválida), tan solo está agitando su cola y diciendo: "Creo que has dicho algo, pero no te entiendo; de todos modos, por favor, sigue hablando conmigo (>>>)."

A medida que tus programas vayan aumentando su complejidad, te encontrarás con tres tipos de errores generales:

Errores de sintaxis (Syntax errors) Estos son los primeros errores que cometerás y también los más fáciles de solucionar. Un error de sintaxis significa que has violado las reglas "gramaticales" de Python. Python hace todo lo que puede para señalar el punto exacto, la línea y el carácter donde ha detectado el fallo. Lo único complicado de los errores de sintaxis es que a veces el error que debe corregirse está en realidad en una línea anterior a la cual Python *detectó* ese fallo. De modo que la línea y el carácter que Python indica en un error de sintaxis pueden ser tan sólo un punto de partida para tu investigación.

Errores lógicos Se produce un error lógico cuando un programa tiene una sintaxis correcta, pero existe un error en el orden de las sentencias o en la forma en que están relacionadas unas con otras. Un buen ejemplo de un error lógico sería: "toma un trago de tu botella de agua, ponla en tu mochila, camina hasta la biblioteca y luego vuelve a enroscar la tapa en la botella."

Errores semánticos Un error semántico ocurre cuando la descripción que has brindado de los pasos a seguir es sintácticamente perfecta y está en el orden correcto, pero sencillamente hay un error en el programa. El programa es correcto, pero no hace lo que tú *pretendías* que hiciera. Un ejemplo podría ser cuando le das indicaciones a alguien sobre cómo llegar a un restaurante, y le dices "...cuando llegues a la intersección con la gasolinera, gira a la izquierda, continúa durante otro kilómetro y el restaurante es el edificio rojo que encontrarás a tu izquierda.". Tu amigo se retrasa y te llama para decirte que está en una granja dando vueltas alrededor de un granero, sin rastro alguno de un restaurante. Entonces le preguntas "¿giraste a la izquierda o la derecha?", y te responde "Seguí tus indicaciones al pie de la letra, dijiste que girara a la izquierda y continuar un kilómetro desde la gasolinera.", entonces le respondes "Lo siento mucho, porque a pesar de que mis indica-

ciones fueron sintácticamente correctas, tristemente contenían un pequeño pero indetectado error semántico.".

Insisto en que, ante cualquiera de estos tres tipos de errores, Python únicamente hace lo que está a su alcance por seguir al pie de la letra lo que tú le has pedido que haga.

1.11 Depurando los programas

Cuando Python muestra un error, u obtiene un resultado diferente al que esperabas, empieza una intensa búsqueda de la causa del error. Depurar es el proceso de encontrar la causa o el origen de ese error en tu código. Cuando depuras un programa, y especialmente cuando tratas con un *bug* algo difícil de solucionar, existen cuatro cosas por hacer:

leer Revisar tu código, leerlo de nuevo, y asegurarte de que ahí está expresado de forma correcta lo que quieres decir.

ejecutar Prueba haciendo cambios y ejecutando diferentes versiones. Con frecuencia, si muestras en tu programa lo correcto en el lugar indicado, el problema se vuelve obvio, pero en ocasiones debes invertir algo de tiempo hasta conseguirlo.

pensar detenidamente ¡Toma tu tiempo para pensar!, ¿A qué tipo de error corresponde: sintaxis, en tiempo de ejecución, semántico?, ¿Qué información puedes obtener de los mensajes de error, o de la salida del programa?, ¿Qué tipo de errores podría generar el problema que estás abordando?, ¿Cuál fue el último cambio que hiciste, antes de que se presentara el problema?

retroceder En algún momento, lo mejor que podrás hacer es dar marcha atrás, deshacer los cambios recientes hasta obtener de nuevo un programa que funcione y puedas entender. Llegado a ese punto, podrás continuar con tu trabajo.

Algunas veces, los programadores novatos se quedan estancados en una de estas actividades y olvidan las otras. Encontrar un *bug* requiere leer, ejecutar, pensar detenidamente y algunas veces retroceder. Si te bloqueas en alguna de estas actividades, prueba las otras. Cada actividad tiene su procedimiento de análisis.

Por ejemplo, leer tu código podría ayudar si el problema es un error tipográfico, pero no si es uno conceptual. Si no comprendes lo que hace tu programa, puedes leerlo 100 veces y no encontrarás el error, puesto que dicho error está en tu mente.

Experimentar puede ayudar, especialmente si ejecutas pequeñas pruebas. Pero si experimentas sin pensar o leer tu código, podrías caer en un patrón que llamo "programación de paseo aleatorio", que es el proceso de realizar cambios al azar hasta que el programa logre hacer lo que debería. No hace falta mencionar que este tipo de programación puede tomar mucho tiempo.

Debes tomar el tiempo suficiente para pensar. Depurar es como una ciencia experimental. Debes plantear al menos una hipótesis sobre qué podría ser el problema. Si hay dos o más posibilidades, piensa en alguna prueba que pueda ayudarte a descartar una de ellas.

Descansar y conversar ayuda a estimular el pensamiento. Si le explicas el problema a alguien más (o incluso a tí mismo), a veces encontrarás la respuesta antes de terminar la pregunta.

Pero incluso las mejores técnicas de depuración fallarán si hay demasiados errores, o si el código que intentas mejorar es demasiado extenso y complicado. Algunas veces la mejor opción es retroceder, y simplificar el programa hasta que obtengas algo que funcione y puedas entender.

Por lo general, los programadores novatos son reacios a retroceder porque no soportan tener que borrar una línea de código (incluso si está mal). Si te hace sentir mejor, prueba a copiar tu programa en otro archivo antes de empezar a modificarlo. De esa manera, puedes recuperar poco a poco pequeñas piezas de código que necesites.

1.12 El camino del aprendizaje

Según vayas avanzando por el resto del libro, no te asustes si los conceptos no parecen encajar bien unos con otros al principio. Cuando estabas aprendiendo a hablar, no supuso un problema que durante los primeros años solo pudieras emitir lindos balbuceos. Y también fue normal que te llevara seis meses pasar de un vocabulario simple a frases simples, y que te llevara 5-6 años más pasar de frases a párrafos, y que todavía tuvieran que transcurrir unos cuantos años más hasta que fuiste capaz de escribir tu propia historia corta interesante.

Pretendemos que aprendas Python rápidamente, por lo que te enseñaremos todo al mismo tiempo durante los próximos capítulos. Aún así, ten en cuenta que el proceso es similar a aprender un idioma nuevo, que lleva un tiempo absorber y comprender antes de que te resulte familiar. Eso produce cierta confusión, puesto que revisaremos en distintas ocasiones determinados temas, y trataremos que de esa manera puedas visualizar los pequeños fragmentos que componen esta obra completa. A pesar de que el libro está escrito de forma lineal, no dudes en ser no lineal en la forma en que abordes las materias. Avanza y retrocede, y lee a veces por encima. Al ojear material más avanzado sin comprender del todo los detalles, tendrás una mejor comprensión del "¿por qué?" de la programación. Al revisar el material anterior e incluso al realizar nuevamente los ejercicios previos, te darás cuenta que ya has aprendido un montón de cosas, incluso si el tema que estás examinando en ese momento parece un poco difícil de abordar.

Normalmente, cuando uno aprende su primer lenguaje de programación, hay unos pocos momentos "¡A-já!" estupendos, en los cuales puedes levantar la vista de la roca que estás machacando con martillo y cincel, separarte unos pasos y comprobar que lo que estás intentando construir es una maravillosa escultura.

Si algo parece particularmente difícil, generalmente no vale la pena quedarse mirándolo toda la noche. Respira, toma una siesta, come algo, explícale a alguien (quizás a tu perro) con qué estás teniendo problemas, y después vuelve a observarlo con un perspectiva diferente. Te aseguro que una vez que aprendas los conceptos de la programación en el libro, volverás atrás y verás que en realidad todo era fácil, elegante y que simplemente te ha llevado un poco de tiempo llegar a absorberlo.

1.13 Glosario

bug Un error en un programa.

código fuente Un programa en un lenguaje de alto nivel.

código máquina El lenguaje de más bajo nivel para el software, es decir, el lenguaje que es directamente ejecutado por la unidad central de procesamiento (CPU).

compilar Traducir un programa completo escrito en un lenguaje de alto nivel a un lenguaje de bajo nivel, para dejarlo listo para una ejecución posterior.

error semántico Un error dentro de un programa que provoca que haga algo diferente de lo que pretendía el programador.

función print Una instrucción que hace que el intérprete de Python muestre un valor en la pantalla.

indicador de línea de comandos (*prompt*) Cuando un programa muestra un mensaje y se detiene para que el usuario teclee algún tipo de dato.

interpretar Ejecutar un programa escrito en un lenguaje de alto nivel traduciéndolo línea a línea.

lenguaje de alto nivel Un lenguaje de programación como Python, que ha sido diseñado para que sea fácil de leer y escribir por las personas.

lenguaje de bajo nivel Un lenguaje de programación que está diseñado para ser fácil de ejecutar para una computadora; también recibe el nombre de "código máquina", "lenguaje máquina" o "lenguaje ensamblador".

memoria principal Almacena los programas y datos. La memoria principal pierde su información cuando se desconecta la alimentación.

memoria secundaria Almacena los programas y datos y mantiene su información incluso cuando se interrumpe la alimentación. Es generalmente más lenta que la memoria principal. Algunos ejemplos de memoria secundaria son las unidades de disco y memorias flash que se encuentran dentro de los dispositivos USB.

modo interactivo Un modo de usar el intérprete de Python, escribiendo comandos y expresiones directamente en el indicador de la línea de comandos.

parsear Examinar un programa y analizar la estructura sintáctica.

portabilidad Es la propiedad que poseen los programas que pueden funcionar en más de un tipo de computadora.

programa Un conjunto de instrucciones que indican cómo realizar algún tipo de cálculo.

resolución de un problema El proceso de formular un problema, encontrar una solución, y mostrar esa solución.

semántica El significado de un programa.

unidad central de procesamiento El corazón de cualquier computadora. Es lo que ejecuta el software que escribimos. También recibe el nombre de "CPU" por sus siglas en inglés (*Central Processsing Unit*), o simplemente, "el procesador".

1.14 Ejercicios

Ejercicio 1: ¿Cuál es la función de la memoria secundaria en una computadora?

a) Ejecutar todos los cálculos y la lógica del programa

b) Descargar páginas web de Internet
c) Almacenar información durante mucho tiempo, incluso después de ciclos de apagado y encendido
d) Recolectar la entrada del usuario

Ejercicio 2: ¿Qué es un programa?

Ejercicio 3: ¿Cuál es la diferencia entre un compilador y un intérprete?

Ejercicio 4: ¿Cuál de los siguientes contiene "código máquina"?

a) El intérprete de Python
b) El teclado
c) El código fuente de Python
d) Un documento de un procesador de texto

Ejercicio 5: ¿Qué está mal en el siguiente código?:

```
>>> primt '¡Hola, mundo!'
  File "<stdin>", line 1
    primt '¡Hola, mundo!'
                ^
SyntaxError: invalid syntax
>>>
```

Ejercicio 6: ¿En qué lugar del computador queda almacenada una variable, como en este caso "X", después de ejecutar la siguiente línea de Python?:

```
x = 123
```

a) Unidad central de procesamiento
b) Memoria Principal
c) Memoria Secundaria
d) Dispositivos de Entrada
e) Dispositivos de Salida

Ejercicio 7: ¿Qué mostrará en pantalla el siguiente programa?:

```
x = 43
x = x + 1
print(x)
```

a) 43
b) 44
c) x + 1
d) Error, porque x = x + 1 no es posible matemáticamente.

Ejercicio 8: Explica cada uno de los siguientes conceptos usando un ejemplo de una capacidad humana: (1) Unidad central de procesamiento, (2) Memoria principal, (3) Memoria secundaria, (4) Dispositivos de entrada, y (5) Dispositivos de salida. Por ejemplo, "¿Cuál sería el equivalente humano de la Unidad central de procesamiento?".

Ejercicio 9: ¿Cómo puedes corregir un "Error de sintaxis"?.

Chapter 2

Variables, expresiones y sentencias

2.1 Valores y tipos

Un *valor* es una de las cosas básicas que utiliza un programa, como una letra o un número. Los valores que hemos visto hasta ahora han sido `1`, `2`, y "¡Hola, mundo!"

Esos valores pertenecen a *tipos* diferentes: `2` es un entero (int), y "¡Hola, mundo!" es una *cadena* (string), que recibe ese nombre porque contiene una "cadena" de letras. Tú (y el intérprete) podéis identificar las cadenas porque van encerradas entre comillas.

La sentencia `print` también funciona con enteros. Vamos a usar el comando `python` para iniciar el intérprete.

```
python
>>> print(4)
4
```

Si no estás seguro de qué tipo de valor estás manejando, el intérprete te lo puede decir.

```
>>> type(''¡Hola, mundo!)
<class 'str'>
>>> type(17)
<class 'int'>
```

Not surprisingly, strings belong to the type `str` and integers belong to the type `int`. Less obviously, numbers with a decimal point belong to a type called `float`, because these numbers are represented in a format called *floating point*.

```
>>> type(3.2)
<class 'float'>
```

¿Qué ocurre con valores como "17" y "3.2"? Parecen números, pero van entre comillas como las cadenas.

```
>>> type('17')
<class 'str'>
>>> type('3.2')
<class 'str'>
```

Son cadenas.

Cuando escribes un entero grande, puede que te sientas tentado a usar comas o puntos para separarlo en grupos de tres dígitos, como en `1,000,000` [1]. Eso no es un entero válido en Python, pero en cambio sí que resulta válido algo como:

```
>>> print(1,000,000)
1 0 0
```

Bien, ha funcionado. ¡Pero eso no era lo que esperábamos!. Python interpreta `1,000,000` como una secuencia de enteros separados por comas, así que lo imprime con espacios en medio.

Éste es el primer ejemplo que hemos visto de un error semántico: el código funciona sin producir ningún mensaje de error, pero no hace su trabajo "correctamente".

2.2 Variables

Una de las características más potentes de un lenguaje de programación es la capacidad de manipular *variables*. Una variable es un nombre que se refiere a un valor.

Una *sentencia de asignación* crea variables nuevas y las da valores:

```
>>> mensaje = 'Y ahora algo completamente diferente'
>>> n = 17
>>> pi = 3.1415926535897931
```

Este ejemplo hace tres asignaciones. La primera asigna una cadena a una variable nueva llamada `mensaje`; la segunda asigna el entero `17` a `n`; la tercera asigna el valor (aproximado) de π a `pi`.

Para mostrar el valor de una variable, se puede usar la sentencia print:

```
>>> print(n)
17
>>> print(pi)
3.141592653589793
```

El tipo de una variable es el tipo del valor al que se refiere.

[1] En el mundo anglosajón el "separador de millares" es la coma, y no el punto (Nota del trad.)

```
>>> type(mensaje)
<class 'str'>
>>> type(n)
<class 'int'>
>>> type(pi)
<class 'float'>
```

2.3 Nombres de variables y palabras claves

Los programadores generalmente eligen nombres para sus variables que tengan sentido y documenten para qué se usa esa variable.

Los nombres de las variables pueden ser arbitrariamente largos. Pueden contener tanto letras como números, pero no pueden comenzar con un número. Se pueden usar letras mayúsculas, pero es buena idea comenzar los nombres de las variables con una letras minúscula (veremos por qué más adelante).

El carácter guión-bajo (_) puede utilizarse en un nombre. A menudo se utiliza en nombres con múltiples palabras, como en `mi_nombre` o `velocidad_de_golondrina_sin_carg` Los nombres de las variables pueden comenzar con un carácter guión-bajo, pero generalmente se evita usarlo así a menos que se esté escribiendo código para librerías que luego utilizarán otros.

Si se le da a una variable un nombre no permitido, se obtiene un error de sintaxis:

```
>>> 76trombones = 'gran desfile'
SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Teorema avanzado de Zymurgy'
SyntaxError: invalid syntax
```

`76trombones` es incorrecto porque comienza por un número. `more@` es incorrecto porque contiene un carácter no premitido, `@`. Pero, ¿qué es lo que está mal en `class`?

Pues resulta que `class` es una de las *palabras clave* de Python. El intérprete usa palabras clave para reconocer la estructura del programa, y esas palabras no pueden ser utilizadas como nombres de variables.

Python reserva 33 palabras claves para su propio uso:

```
and       del       from      None      True
as        elif      global    nonlocal  try
assert    else      if        not       while
break     except    import    or        with
class     False     in        pass      yield
continue  finally   is        raise
def       for       lambda    return
```

Puede que quieras tener esta lista a mano. Si el intérprete se queja por el nombre de una de tus variables y no sabes por qué, comprueba si ese nombre está en esta lista.

2.4 Sentencias

Una *sentencia* es una unidad de código que el intérprete de Python puede ejecutar. Hemos visto hasta ahora dos tipos de sentencia: print y las asignaciones.

Cuando escribes una sentencia en modo interactivo, el intérprete la ejecuta y muestra el resultado, si es que lo hay.

Un script normalmente contiene una secuencia de sentencias. Si hay más de una sentencia, los resultados aparecen de uno en uno según se van ejecutando las sentencias.

Por ejemplo, el script

```
print(1)
x = 2
print(x)
```

produce la salida

```
1
2
```

La sentencia de asignación no produce ninguna salida.

2.5 Operadores y operandos

Los operadores son símbolos especiales que representan cálculos, como la suma o la multiplicación. Los valores a los cuales se aplican esos operadores reciben el nombre de *operandos*.

Los operadores `+`, `-`, , `/`, y `\*` realizan sumas, restas, multiplicaciones, divisiones y exponenciación (elevar un número a una potencia), como se muestra en los ejemplos siguientes:

```
20+32
hour-1
hour*60+minute
minute/60
5**2
(5+9)*(15-7)
```

Ha habido un cambio en el operador de división entre Python 2.x y Python 3.x. En Python 3.x, el resultado de esta división es un resultado de punto flotante:

```
>>> minute = 59
>>> minute/60
0.9833333333333333
```

El operador de división en Python 2.0 dividiría dos enteros y truncar el resultado a un entero:

```
>>> minute = 59
>>> minute/60
0
```

Para obtener la misma respuesta en Python 3.0 use división dividida (// integer).

```
>>> minute = 59
>>> minute//60
0
```

En Python 3, la división de enteros funciona mucho más como cabría esperar. Si ingresaste la expresión en una calculadora.

2.6 Expresiones

Una *expresión* es una combinación de valores, variables y operadores. Un valor por si mismo se considera una expresión, y también lo es una variable, así que las siguientes expresiones son todas válidas (asumiendo que la variable `x` tenga un valor asignado):

```
17
x
x + 17
```

Si escribes una expresión en modo interactivo, el intérprete la *evalúa* y muestra el resultado:

```
>>> 1 + 1
2
```

Sin embargo, en un script, ¡una expresión por si misma no hace nada! Esto a menudo puede producir confusión entre los principiantes.

Ejercicio 1: Escribe las siguientes sentencias en el intérprete de Python para comprobar qué hacen:

```
5
x = 5
x + 1
```

2.7 Orden de las operaciones

Cuando en una expresión aparece más de un operador, el orden de evaluación depende de las *reglas de precedencia*. Para los operadores matemáticos, Python sigue las convenciones matemáticas. El acrónimo *PEMDSR* resulta útil para recordar esas reglas:

- Los *P*aréntesis tienen el nivel superior de precedencia, y pueden usarse para forzar a que una expresión sea evaluada en el orden que se quiera. Dado que las expresiones entre paréntesis son evaluadas primero, `2 * (3-1)` es 4, y `(1+1)**(5-2)` es 8. Se pueden usar también paréntesis para hacer una expresión más sencilla de leer, incluso si el resultado de la misma no varía por ello, como en `(minuto * 100) / 60`.
- La *E*xponenciación (elevar un número a una potencia) tiene el siguiente nivel más alto de precedencia, de modo que `2**1+1` es 3, no 4, y `3*1**3` es 3, no 27.
- La *M*ultiplicación y la *D*ivisión tienen la misma precedencia, que es superior a la de la *S*uma y la *R*esta, que también tienen entre si el mismo nivel de precedencia. Así que `2*3-1` es 5, no 4, y `6+4/2` es 8, no 5.
- Los operadores con igual precedencia son evaluados de izquierda a derecha. Así que la expresión `5-3-1` es 1 y no 3, ya que `5-3` se evalúa antes, y después se resta `1` de `2`.

En caso de duda, añade siempre paréntesis a tus expresiones para asegurarte de que las operaciones se realizan en el orden que tú quieres.

2.8 Operador módulo

El *operador módulo* trabaja con enteros y obtiene el resto de la operación consistente en dividir el primer operando por el segundo. En Python, el operador módulo es un signo de porcentaje (`%`). La sintaxis es la misma que se usa para los demás operadores:

```
>>> quotient = 7 // 3
>>> print(quotient)
2
>>> remainder = 7 % 3
>>> print(remainder)
1
```

Así que 7 dividido por 3 es 2 y nos sobra 1.

El operador módulo resulta ser sorprendentemente útil. Por ejemplo, puedes comprobar si un número es divisible por otro—si `x % y` es cero, entonces `x` es divisible por `y`.

También se puede extraer el dígito más a la derecha de los que componen un número. Por ejemplo, `x % 10` obtiene el dígito que está más a la derecha de `x` (en base 10). De forma similar, `x % 100` obtiene los dos últimos dígitos.

2.9 Operaciones con cadenas

El operador + funciona con las cadenas, pero no realiza una suma en el sentido matemático. En vez de eso, realiza una *concatenación*, que quiere decir que une ambas cadenas, enlazando el final de la primera con el principio de la segunda. Por ejemplo:

```
>>> primero = 10
>>> segundo = 15
>>> print(primero+segundo)
25
>>> primero = '100'
>>> segundo = '150'
>>> print(primero + segundo)
100150
```

La salida de este programa es `100150`.

El operador * también trabaja con cadenas multiplicando el contenido de una cadena por un entero. Por ejemplo:

```
>>> primero = 'Test '
>>> second = 3
>>> print(primero * second)
Test Test Test
```

2.10 Petición de información al usuario

A veces necesitaremos que sea el usuario quien nos proporcione el valor para una variable, a través del teclado. Python proporciona una función interna llamada `input` que recibe la entrada desde el teclado. Cuando se llama a esa función, el programa se detiene y espera a que el usuario escriba algo. Cuando el usuario pulsa `Retorno` o `Intro`, el programa continúa y `input` devuelve como una cadena aquello que el usuario escribió.

```
>>> entrada = input()
Cualquier cosa ridícula
>>> print(entrada)
Cualquier cosa ridícula
```

Antes de recibir cualquier dato desde el usuario, es buena idea escribir un mensaje explicándole qué debe introducir. Se puede pasar una cadena a `input`, que será mostrada al usuario antes de que el programa se detenga para recibir su entrada:

```
>>> nombre = input('¿Cómo te llamas?\n')
¿Cómo te llamas?
Chuck
>>> print(nombre)
Chuck
```

La secuencia `\n` al final del mensaje representa un *newline*, que es un carácter especial que provoca un salto de línea. Por eso la entrada del usuario aparece debajo de nuestro mensaje.

Si esperas que el usuario escriba un entero, puedes intentar convertir el valor de retorno a `int` usando la función `int()`:

```
>>> prompt = '¿Cual.... es la velocidad de vuelo de una golondrina sin carga?\n'
>>> velocidad = input(prompt)
¿Cual.... es la velocidad de vuelo de una golondrina sin carga?
17
>>> int(velocidad)
17
>>> int(velocidad) + 5
22
```

Pero si el usuario escribe algo que no sea una cadena de dígitos, obtendrás un error:

```
>>> velocidad = input(prompt)
¿Cual.... es la velocidad de vuelo de una golondrina sin carga?
¿Te refieres a una golondrina africana o a una europea?
>>> int(velocidad)
ValueError: invalid literal for int()
```

Veremos cómo controlar este tipo de errores más adelante.

2.11 Comentarios

A medida que los programas se van volviendo más grandes y complicados, se vuelven más difíciles de leer. Los lenguajes formales son densos, y a menudo es complicado mirar un trozo de código e imaginarse qué es lo que hace, o por qué.

Por eso es buena idea añadir notas a tus programas, para explicar en un lenguaje normal qué es lo que el programa está haciendo. Estas notas reciben el nombre de *comentarios*, y en Python comienzan con el símbolo `#`:

```
# calcula el porcentaje de hora transcurrido
porcentaje = (minuto * 100) / 60
```

En este caso, el comentario aparece como una línea completa. Pero también puedes poner comentarios al final de una línea

```
porcentaje = (minuto * 100) / 60      # porcentaje de una hora
```

Todo lo que va desde `#` hasta el final de la línea es ignorado—no afecta para nada al programa.

Las comentarios son más útiles cuando documentan características del código que no resultan obvias. Es razonable asumir que el lector puede descifrar *qué* es lo que el código hace; es mucho más útil explicarle *por qué*.

Este comentario es redundante con el código e inútil:

```
v = 5      # asigna 5 a v
```

Este comentario contiene información útil que no está en el código:

```
v = 5      # velocidad en metros/segundo.
```

Elegir nombres adecuados para las variables puede reducir la necesidad de comentarios, pero los nombres largos también pueden ocasionar que las expresiones complejas sean difíciles de leer, así que hay que conseguir una solución de compromiso.

2.12 Elección de nombres de variables mnemónicos

Mientras sigas las sencillas reglas de nombrado de variables y evites las palabras reservadas, dispondrás de una gran variedad de opciones para poner nombres a tus variables. Al principio, esa diversidad puede llegar a resultarte confusa, tanto al leer un programa como al escribir el tuyo propio. Por ejemplo, los tres programas siguientes son idénticos en cuanto a la función que realizan, pero muy diferentes cuando los lees e intentas entenderlos.

```
a = 35.0
b = 12.50
c = a * b
print(c)
```

```
horas = 35.0
tarifa = 12.50
salario = horas * tarifa
print(salario)
```

```
x1q3z9ahd = 35.0
x1q3z9afd = 12.50
x1q3p9afd = x1q3z9ahd * x1q3z9afd
print(x1q3p9afd)
```

El intérprete de Python ve los tres programas como *exactamente idénticos*, pero los humanos ven y asimilan estos programas de forma bastante diferente. Los humanos entenderán más rápidamente el *objetivo* del segundo programa, ya que el programador ha elegido nombres de variables que reflejan lo que pretendía de acuerdo al contenido que iba almacenar en cada variable.

Esa sabia elección de nombres de variables se denomina utilizar "nombres de variables mnemónicos". La palabra *mnemónico*[2] significa "que ayuda a memorizar".

[2]Consulta https://es.wikipedia.org/wiki/Mnemonico para obtener una descripción detallada de la palabra "mnemónico".

Elegimos nombres de variables mnemónicos para ayudarnos a recordar por qué creamos las variables al principio.

A pesar de que todo esto parezca estupendo, y de que sea una idea muy buena usar nombres de variables mnemónicos, ese tipo de nombres pueden interponerse en el camino de los programadores novatos a la hora de analizar y comprender el código. Esto se debe a que los programadores principiantes no han memorizado aún las palabras reservadas (sólo hay 33), y a veces variables con nombres que son demasiado descriptivos pueden llegar a parecerles parte del lenguaje y no simplemente nombres de variable bien elegidos[3].

Echa un vistazo rápido al siguiente código de ejemplo en Python, que se mueve en bucle a través de un conjunto de datos. Trataremos los bucles pronto, pero por ahora tan sólo trata de entender su significado:

```
for word in words:
    print(word)
```

¿Qué ocurre aquí? ¿Cuáles de las piezas (for, word, in, etc.) son palabras reservadas y cuáles son simplemente nombres de variables? ¿Acaso Python comprende de un modo básico la noción de palabras (`words`)? Los programadores novatos tienen problemas separando qué parte del código *debe* mantenerse tal como está en este ejemplo y qué partes son simplemente elección del programador.

El código siguiente es equivalente al de arriba:

```
for slice in pizza:
    print(slice)
```

Para los principiantes es más fácil estudiar este código y saber qué partes son palabras reservadas definidas por Python y qué partes son simplemente nombres de variables elegidas por el programador. Está bastante claro que Python no entiende nada de pizza ni de porciones, ni del hecho de que una pizza consiste en un conjunto de una o más porciones.

Pero si nuestro programa lo que realmente va a hacer es leer datos y buscar palabras en ellos, `pizza` y `porción` son nombres muy poco mnemónicos. Elegirlos como nombres de variables distrae del propósito real del programa.

Dentro de muy poco tiempo, conocerás las palabras reservadas más comunes, y empezarás a ver cómo esas palabras reservadas resaltan sobre las demás:

Las partes del código que están definidas por Python (`for`, `in`, `print`, y `:`) están en negrita, mientras que las variables elegidas por el programador (`word` y `words`) no lo están. Muchos editores de texto son conscientes de la sintaxis de Python y colorearán las palabras reservadas de forma diferente para darte pistas que te permitan mantener tus variables y las palabras reservadas separados. Dentro de poco empezarás a leer Python y podrás determinar rápidamente qué es una variable y qué es una palabra reservada.

[3] El párrafo anterior se refiere más bien a quienes eligen nombres de variables en inglés, ya que todas las palabras reservadas de Python coinciden con palabras propias de ese idioma (Nota del trad.)

2.13 Depuración

En este punto, el error de sintaxis que es más probable que cometas será intentar utilizar nombres de variables no válidos, como `class` y `yield`, que son palabras clave, o `odd~job` y `US$`, que contienen caracteres no válidos.

Si pones un espacio en un nombre de variable, Python cree que se trata de dos operandos sin ningún operador:

```
>>> bad name = 5
SyntaxError: invalid syntax

>>> month = 09
  File "<stdin>", line 1
    month = 09
            ^
SyntaxError: invalid token
```

Para la mayoría de errores de sintaxis, los mensajes de error no ayudan mucho. Los mensajes más comunes son `SyntaxError: invalid syntax` y `SyntaxError: invalid token`, ninguno de los cuales resulta muy informativo.

El runtime error (error en tiempo de ejecución) que es más probable que obtengas es un "use before def" (uso antes de definir); que significa que estás intentando usar una variable antes de que le hayas asignado un valor. Eso puede ocurrir si escribes mal el nombre de la variable:

```
>>> principal = 327.68
>>> interest = principle * rate
NameError: name 'principle' is not defined
```

Los nombres de las variables son sensibles a mayúsculas, así que `LaTeX` no es lo mismo que `latex`.

En este punto, la causa más probable de un error semántico es el orden de las operaciones. Por ejemplo, para evaluar $\frac{1}{2\pi}$, puedes sentirte tentado a escribir

```
>>> 1.0 / 2.0 * pi
```

Pero la división se evalúa antes, ¡así que obtendrás $\pi/2$, que no es lo mismo! No hay forma de que Python sepa qué es lo que querías escribir exactamente, así que en este caso no obtienes un mensaje de error; simplemente obtienes una respuesta incorrecta.

2.14 Glosario

asignación Una sentencia que asigna un valor a una variable.

cadena Un tipo que representa secuencias de caracteres.

concatenar Unir dos operandos, uno a continuación del otro.

comentario Información en un programa que se pone para otros programadores (o para cualquiera que lea el código fuente), y no tiene efecto alguno en la ejecución del programa.

división entera La operación que divide dos números y trunca la parte fraccionaria.

entero Un tipo que representa números enteros.

evaluar Simplificar una expresión realizando las operaciones en orden para obtener un único valor.

expresión Una combinación de variables, operadores y valores que representan un único valor resultante.

mnemónico Una ayuda para memorizar. A menudo damos nombres mnemónicos a las variables para ayudarnos a recordar qué está almacenado en ellas.

palabra clave Una palabra reservada que es usada por el compilador para analizar un programa; no se pueden usar palabres clave como `if`, `def`, y `while` como nombres de variables.

punto flotante Un tipo que representa números con parte decimal.

operador Un símbolo especial que representa un cálculo simple, como suma, multiplicación o concatenación de cadenas.

operador módulo Un operador, representado por un signo de porcentaje (`%`), que funciona con enteros y obtiene el resto cuando un número es dividido por otro.

operando Uno de los valores con los cuales un operador opera.

reglas de precedencia El conjunto de reglas que gobierna el orden en el cual son evaluadas las expresiones que involucran a múltiples operadores.

sentencia Una sección del código que representa un comando o acción. Hasta ahora, las únicas sentencias que hemos visto son asignaciones y sentencias print.

tipo Una categoría de valores. Los tipos que hemos visto hasta ahora son enteros (tipo `int`), números en punto flotante (tipo `float`), y cadenas (tipo `str`).

valor Una de las unidades básicas de datos, como un número o una cadena, que un programa manipula.

variable Un nombre que hace referencia a un valor.

2.15 Ejercicios

Ejercicio 2: Escribe un programa que use `input` para pedirle al usuario su nombre y luego darle la bienvenida.

```
Introduzca tu nombre: Chuck
Hola, Chuck
```

Ejercicio 3: Escribe un programa para pedirle al usuario el número de horas y la tarifa por hora para calcular el salario bruto.

```
Introduzca Horas: 35
Introduzca Tarifa: 2.75
Salario: 96.25
```

Por ahora no es necesario preocuparse de que nuestro salario tenga exactamente dos dígitos después del punto decimal. Si quieres, puedes probar la función interna de Python `round` para redondear de forma adecuada el salario resultante a dos dígitos decimales.

Ejercicio 4: Asume que ejecutamos las siguientes sentencias de asignación:

```
ancho = 17
alto = 12.0
```

Para cada una de las expresiones siguientes, escribe el valor de la expresión y el tipo (del valor de la expresión).

1. `ancho/2`
2. `ancho/2.0`
3. `alto/3`
4. `1 + 2 * 5`

Usa el intérprete de Python para comprobar tus respuestas.

Ejercicio 5: Escribe un programa que le pida al usuario una temperatura en grados Celsius, la convierta a grados Fahrenheit e imprima por pantalla la temperatura convertida.

Chapter 3

Ejecución condicional

3.1 Expresiones booleanas

Una *expresión booleana* es aquella que puede ser verdadera (`True`) o falsa (`False`). Los ejemplos siguientes usan el operador `==`, que compara dos operandos y devuelve `True` si son iguales y `False` en caso contrario:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

`True` y `False` son valores especiales que pertenecen al tipo `bool (booleano)`; no son cadenas:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

El operador `==` es uno de los *operadores de comparación*; los demás son:

```
x != y               # x es distinto de y
x > y                # x es mayor que y
x < y                # x es menor que y
x >= y               # x es mayor o igual que y
x <= y               # x es menor o igual que y
x is y               # x es lo mismo que y
x is not y           # x no es lo mismo que y
```

A pesar de que estas operaciones probablemente te resulten familiares, los símbolos en Python son diferentes de los símbolos matemáticos que se usan para realizar las mismas operaciones. Un error muy común es usar sólo un símbolo igual (`=`) en vez del símbolo de doble igualdad (`==`). Recuerda que `=` es un operador de asignación, y `==` es un operador de comparación. No existe algo como `=<` o `=>`.

3.2 Operadores lógicos

Existen tres *operadores lógicos*: `and (y)`, `or (o)`, y `not (no)`. El significado semántico de estas operaciones es similar a su significado en inglés. Por ejemplo,

`x > 0 and x < 10`

es verdadero sólo cuando `x` es mayor que 0 *y* menor que 10.

`n%2 == 0 or n%3 == 0` es verdadero si *cualquiera* de las condiciones es verdadera, es decir, si el número es divisible por 2 *o* por 3.

Finalmente, el operador `not` niega una expresión booleana, de modo que `not (x > y)` es verdadero si `x > y` es falso; es decir, si `x` es menor o igual que `y`.

Estrictamente hablando, los operandos de los operadores lógicos deberían ser expresiones booleanas, pero Python no es muy estricto. Cualquier número distinto de cero se interpreta como "verdadero."

```
>>> 17 and True
True
```

Esta flexibilidad puede ser útil, pero existen ciertas sutilezas en ese tipo de uso que pueden resultar confusas. Es posible que prefieras evitar usarlo de este modo hasta que estés bien seguro de lo que estás haciendo.

3.3 Ejecución condicional

Para poder escribir programas útiles, casi siempre vamos a necesitar la capacidad de comprobar condiciones y cambiar el comportamiento del programa de acuerdo a ellas. Las `sentencias condicionales` nos proporcionan esa capacidad. La forma más sencilla es la sentencia `if`:

```
if x > 0 :
    print('x es positivo')
```

La expresión booleana después de la sentencia `if` recibe el nombre de *condición*. La sentencia `if` se finaliza con un carácter de dos-puntos (:) y la(s) línea(s) que van detrás de la sentencia if van indentadas[1] (es decir, llevan una tabulación o varios espacios en blanco al principio).

Si la condición lógica es verdadera, la sentencia indentada será ejecutada. Si la condición es falsa, la sentencia indentada será omitida.

La sentencia `if` tiene la misma estructura que la definición de funciones o los bucles `for`[2]. La sentencia consiste en una línea de encabezado que termina con el carácter dos-puntos (:) seguido por un bloque indentado. Las sentencias de este tipo reciben el nombre de *sentencias compuestas*, porque se extienden a lo largo de varias líneas.

[1] el término correcto en español sería "sangradas", pero en el mundillo de la programación se suele decir que las líneas van "indentadas" (Nota del trad.)

[2] Estudiaremos las funciones en el capítulo 4 y los bucles en el capítulo 5.

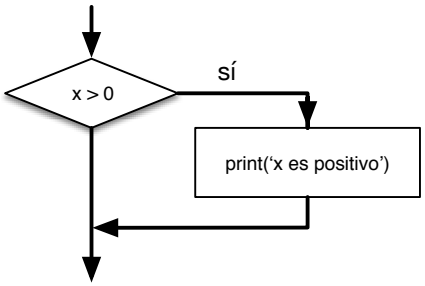

Figure 3.1: If Logic

No hay límite en el número de sentencias que pueden aparecer en el cuerpo, pero debe haber al menos una. Ocasionalmente, puede resultar útil tener un cuerpo sin sentencias (normalmente como emplazamiento reservado para código que no se ha escrito aún). En ese caso, se puede usar la sentencia `pass`, que no hace nada.

```
if x < 0 :
    pass          # ¡necesito gestionar los valores negativos!
```

Si introduces una sentencia `if` en el intérprete de Python, el prompt cambiará su aspecto habitual por puntos suspensivos, para indicar que estás en medio de un bloque de sentencias, como se muestra a continuación:

```
>>> x = 3
>>> if x < 10:
...     print('Pequeño')
...
Pequeño
>>>
```

Al usar el intérprete de Python, debe dejar una línea en blanco al final de un bloque, de lo contrario Python devolverá un error:

```
>>> x = 3
>>> if x < 10:
...     print('Pequeño')
... print('Hecho')
  File "<stdin>", line 3
    print('Hecho')
        ^
SyntaxError: invalid syntax
```

No es necesaria una línea en blanco al final de un bloque de instrucciones al escribir y ejecutar un script, pero puede mejorar la legibilidad de su código.

3.4 Ejecución alternativa

La segunda forma de la sentencia `if` es la *ejecución alternativa*, en la cual existen dos posibilidades y la condición determina cual de ellas será ejecutada. La sintaxis es similar a ésta:

```
if x%2 == 0 :
    print('x es par')
else :
    print('x es impar')
```

Si al dividir `x` por 2 obtenemos como resto 0, entonces sabemos que `x` es par, y el programa muestra un mensaje a tal efecto. Si esa condición es falsa, se ejecuta el segundo conjunto de sentencias.

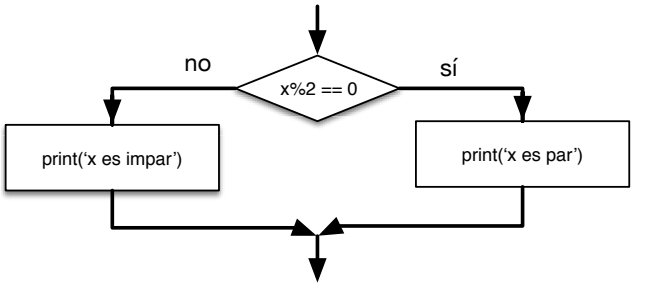

Figure 3.2: If-Then-Else Logic

Dado que la condición debe ser obligatoriamente verdadera o falsa, solamente una de las alternativas será ejecutada. Las alternativas reciben el nombre de *ramas*, dado que se trata de ramificaciones en el flujo de la ejecución.

3.5 Condicionales encadenados

Algunas veces hay más de dos posibilidades, de modo que necesitamos más de dos ramas. Una forma de expresar una operación como ésa es usar un *condicional encadenado*:

```
if x < y:
    print('x es menor que y')
elif x > y:
    print('x es mayor que y')
else:
    print('x e y son iguales')
```

`elif` es una abreviatura para "else if". En este caso también será ejecutada únicamente una de las ramas.

No hay un límite para el número de sentencias `elif`. Si hay una clausula `else`, debe ir al final, pero tampoco es obligatorio que ésta exista.

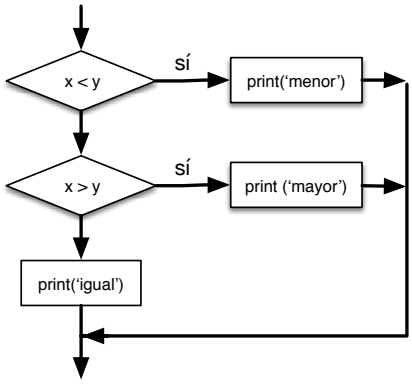

Figure 3.3: If-Then-ElseIf Logic

```
if choice == 'a':
    print('Respuesta incorrecta')
elif choice == 'b':
    print('Respuesta correcta')
elif choice == 'c':
    print('Casi, pero no es correcto')
```

Cada condición es comprobada en orden. Si la primera es falsa, se comprueba la siguiente y así con las demás. Si una de ellas es verdadera, se ejecuta la rama correspondiente, y la sentencia termina. Incluso si hay más de una condición que sea verdadera, sólo se ejecuta la primera que se encuentra.

3.6 Condicionales anidados

Un condicional puede también estar anidado dentro de otro. Podríamos haber escrito el ejemplo anterior de las tres ramas de este modo:

```
if x == y:
    print('x e y son iguales')
else:
    if x < y:
        print('x es menor que y')
    else:
        print('x es mayor que y')
```

El condicional exterior contiene dos ramas. La primera rama ejecuta una sentencia simple. La segunda contiene otra sentencia `if`, que tiene a su vez sus propias dos ramas. Esas dos ramas son ambas sentencias simples, pero podrían haber sido sentencias condicionales también.

A pesar de que el indentado de las sentencias hace que la estructura esté clara, los *condicionales anidados* pueden volverse difíciles de leer rápidamente. En general, es buena idea evitarlos si se puede.

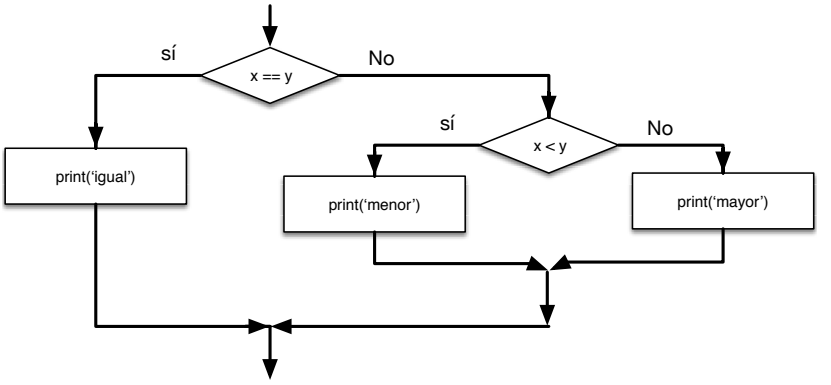

Figure 3.4: Nested If Statements

Los operadores lógicos a menudo proporcionan un modo de simplificar las sentencias condicionales anidadas. Por ejemplo, el código siguiente puede ser reescrito usando un único condicional:

```
if 0 < x:
    if x < 10:
        print('x es un número positivo con un sólo dígito.')
```

La sentencia `print` se ejecuta solamente si se cumplen las dos condiciones anteriores, así que en realidad podemos conseguir el mismo efecto con el operador `and`:

```
if 0 < x and x < 10:
    print('x es un número positivo con un sólo dígito.')
```

3.7 Captura de excepciones usando try y except

Anteriormente vimos un fragmento de código donde usábamos las funciones `input` e `int` para leer y analizar un número entero introducido por el usuario. También vimos lo poco seguro que podía llegar a resultar hacer algo así:

```
>>> velocidad = input(prompt)
¿Cual.... es la velocidad de vuelo de una golondrina sin carga?
¿Te refieres a una golondrina africana o a una europea?
>>> int(velocidad)
ValueError: invalid literal for int() with base 10:
>>>
```

Cuando estamos trabajando con el intérprete de Python, tras el error simplemente nos aparece de nuevo el prompt, así que pensamos "¡epa, me he equivocado!", y continuamos con la siguiente sentencia.

Sin embargo, si se escribe ese código en un script de Python y se produce el error, el script se detendrá inmediatamente, y mostrará un "traceback". No ejecutará la siguiente sentencia.

He aquí un programa de ejemplo para convertir una temperatura desde grados Fahrenheit a grados Celsius:

```
ent = input('Introduzca la Temperatura Fahrenheit:')
fahr = float(ent)
cel = (fahr - 32.0) * 5.0 / 9.0
print(cel)

# Code: http://www.py4e.com/code3/fahren.py
```

Si ejecutamos este código y le damos una entrada no válida, simplemente fallará con un mensaje de error bastante antipático:

```
python fahren.py
Introduzca la Temperatura Fahrenheit:72
22.2222222222


python fahren.py
Introduzca la Temperatura Fahrenheit:fred
Traceback (most recent call last):
  File "fahren.py", line 2, in <module>
    fahr = float(ent)
ValueError: invalid literal for float(): fred
```

Existen estructuras de ejecución condicional dentro de Python para manejar este tipo de errores esperados e inesperados, llamadas "try / except". La idea de `try` y `except` es que si se sabe que cierta secuencia de instrucciones puede generar un problema, sea posible añadir ciertas sentencias para que sean ejecutadas en caso de error. Estas sentencias extras (el bloque except) serán ignoradas si no se produce ningún error.

Puedes pensar en la característica `try` y `except` de Python como una "póliza de seguros" en una secuencia de sentencias.

Se puede reescribir nuestro conversor de temperaturas de esta forma:

```
ent = input('Introduzca la Temperatura Fahrenheit:')
try:
    fahr = float(ent)
    cel = (fahr - 32.0) * 5.0 / 9.0
    print(cel)
except:
    print('Por favor, introduzca un número')

# Code: http://www.py4e.com/code3/fahren2.py
```

Python comienza ejecutando la secuencia de sentencias del bloque `try`. Si todo va bien, se saltará todo el bloque `except` y terminará. Si ocurre una excepción dentro del bloque `try`, Python saltará fuera de ese bloque y ejecutará la secuencia de sentencias del bloque `except`.

```
python fahren2.py
Introduzca la Temperatura Fahrenheit:72
22.2222222222


python fahren2.py
Introduzca la Temperatura Fahrenheit:fred
Por favor, introduzca un número
```

Gestionar una excepción con una sentencia `try` recibe el nombre de *capturar* una excepción. En este ejemplo, la clausula `except` muestra un mensaje de error. En general, capturar una excepción te da la oportunidad de corregir el problema, volverlo a intentar o, al menos, terminar el programa con elegancia.

3.8 Evaluación en cortocircuito de expresiones lógicas

Cuando Python está procesando una expresión lógica, como `x >= 2 and (x/y) > 2`, evalúa la expresión de izquierda a derecha. Debido a la definición de `and`, si `x` es menor de 2, la expresión `x >= 2` resulta ser `falsa`, de modo que la expresión completa ya va a resultar `falsa`, independientemente de si `(x/y) > 2` se evalúa como `verdadera` o `falsa`.

Cuando Python detecta que no se gana nada evaluando el resto de una expresión lógica, detiene su evaluación y no realiza el cálculo del resto de la expresión. Cuando la evaluación de una expresión lógica se detiene debido a que ya se conoce el valor final, eso es conocido como *cortocircuitar* la evaluación.

A pesar de que esto pueda parecer hilar demasiado fino, el funcionamiento en cortocircuito nos descubre una ingeniosa técnica conocida como *patrón guardián*. Examina la siguiente secuencia de código en el intérprete de Python:

```
>>> x = 6
>>> y = 2
>>> x >= 2 and (x/y) > 2
True
>>> x = 1
>>> y = 0
>>> x >= 2 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and (x/y) > 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

La tercera operación ha fallado porque Python intentó evaluar `(x/y)` e `y` era cero, lo cual provoca un runtime error (error en tiempo de ejecución). Pero el segundo

ejemplo *no* falló, porque la primera parte de la expresión `x >= 2` fue evaluada como `falsa`, así que `(x/y)` no llegó a ejecutarse debido a la regla del *cortocircuito*, y no se produjo ningún error.

Es posible construir las expresiones lógicas colocando estratégicamente una evaluación como *guardián* justo antes de la evaluación que podría causar un error, como se muestra a continuación:

```
>>> x = 1
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x >= 2 and (x/y) > 2 and y != 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

En la primera expresión lógica, `x >= 2` es `falsa`, así que la evaluación se detiene en el `and`. En la segunda expresión lógica, `x >= 2` es `verdadera`, pero `y != 0` es `falsa`, de modo que nunca se alcanza `(x/y)`.

En la tercera expresión lógica, el `y != 0` va *después* del cálculo de `(x/y)`, de modo que la expresión falla con un error.

En la segunda expresión, se dice que `y != 0` actúa como *guardián* para garantizar que sólo se ejecute `(x/y)` en el caso de que `y` no sea cero.

3.9 Depuración

Los "traceback" que Python muestra cuando se produce un error contienen un montón de información, pero pueden resultar abrumadores. Las partes más útiles normalmente son:

- Qué tipo de error se ha producido, y
- Dónde ha ocurrido.

Los errores de sintaxis (syntax errors), normalmente son fáciles de localizar, pero a veces tienen trampa. Los errores debido a espacios en blanco pueden ser complicados, ya que los espacios y las tabulaciones son invisibles, y solemos ignorarlos.

```
>>> x = 5
>>>  y = 6
  File "<stdin>", line 1
    y = 6
    ^
IndentationError: unexpected indent
```

En este ejemplo, el problema es que la segunda línea está indentada por un espacio. Pero el mensaje de error apunta a `y`, lo cual resulta engañoso. En general, los mensajes de error indican dónde se ha descubierto el problema, pero el error real podría estar en el código previo, a veces en alguna línea anterior.

Ocurre lo mismo con los errores en tiempo de ejecución (runtime errors). Supón que estás tratando de calcular una relación señal-ruido en decibelios. La fórmula es $SNR_{db} = 10 \log_{10}(P_{senal}/P_{ruido})$. En Python, podrías escribir algo como esto:

```
import math
int_senal = 9
int_ruido = 10
relacion = int_senal / int_ruido
decibelios = 10 * math.log10(relacion)
print(decibelios)

# Code: http://www.py4e.com/code3/snr.py
```

Pero cuando lo haces funcionar, obtienes un mensaje de error[3]:

```
Traceback (most recent call last):
  File "snr.py", line 5, in ?
    decibelios = 10 * math.log10(relacion)
OverflowError: math range error
```

El mensaje de error apunta a la línea 5, pero no hay nada incorrecto en ese línea. Para encontrar el error real, puede resultar útil mostrar en pantalla el valor de `relacion`, que resulta ser 0. El problema está en la línea 4, ya que al dividir dos enteros se realiza una división entera. La solución es representar la intensidad de la señal y la intensidad del ruido con valores en punto flotante.

En general, los mensajes de error te dicen dónde se ha descubierto el problema, pero a menudo no es ahí exactamente donde se ha producido.

3.10 Glosario

condición La expresión booleana en una sentencia condicional que determina qué rama será ejecutada.

condicional anidado Una sentencia condicional que aparece en una de las ramas de otra sentencia condicional.

condicional encadenado Una sentencia condicional con una serie de ramas alternativas.

[3] En Python 3.0, ya no se produce el mensaje de error; el operador de división realiza división en punto flotante incluso con operandos enteros.

cortocircuito Cuando Python va evaluando una expresión lógica por tramos y detiene el proceso de evaluación debido a que ya conoce el valor final que va a tener el resultado sin necesidad de evaluar el resto de la expresión.

cuerpo La secuencia de sentencias en el interior de una sentencia compuesta.

expresión booleana Un expresión cuyo valor puede ser o bien `Verdadero` o bien `Falso`.

operadores de comparación Uno de los operadores que se utiliza para comparar dos operandos: `==`, `!=`, `>`, `<`, `>=`, y `<=`.

operador lógico Uno de los operadores que se combinan en las expresiones booleanas: `and`, `or`, y `not`.

patrón guardián Cuando construimos una expresión lógica con comparaciones adicionales para aprovecharnos del funcionamiento en cortocircuito.

rama Una de las secuencias alternativas de sentencias en una sentencia condicional.

sentencia compuesta Una sentencia que consiste en un encabezado y un cuerpo. El encabezado termina con dos-puntos (:). El cuerpo está indentado con relación al encabezado.

sentencia condicional Una sentencia que controla el flujo de ejecución, dependiendo de cierta condición.

traceback Una lista de las funciones que se están ejecutando, que se muestra en pantalla cuando se produce una excepción.

3.11 Ejercicios

Ejercicio 1: Reescribe el programa del cálculo del salario para darle al empleado 1.5 veces la tarifa horaria para todas las horas trabajadas que excedan de 40.

```
Introduzca las Horas: 45
Introduzca la Tarifa por hora: 10
Salario: 475.0
```

Ejercicio 2: Reescribe el programa del salario usando `try` y `except`, de modo que el programa sea capaz de gestionar entradas no numéricas con elegancia, mostrando un mensaje y saliendo del programa. A continuación se muestran dos ejecuciones del programa:

```
Introduzca las Horas: 20
Introduzca la Tarifa por hora: nueve
Error, por favor introduzca un número

Introduzca las Horas: cuarenta
Error, por favor introduzca un número
```

Ejercicio 3: Escribe un programa que solicite una puntuación entre 0.0 y 1.0. Si la puntuación está fuera de ese rango, muestra un mensaje de error. Si la puntuación está entre 0.0 y 1.0, muestra la calificación usando la tabla siguiente:

```
Puntuación Calificación
>= 0.9     Sobresaliente
>= 0.8     Notable
>= 0.7     Bien
>= 0.6     Suficiente
< 0.6      Insuficiente

Introduzca puntuación: 0.95
Sobresaliente

Introduzca puntuación: perfecto
Puntuación incorrecta

Introduzca puntuación: 10.0
Puntuación incorrecta

Introduzca puntuación: 0.75
Bien

Introduzca puntuación: 0.5
Insuficiente
```

Ejecuta el programa repetidamente, como se muestra arriba, para probar con varios valores de entrada diferentes.

Chapter 4

Funciones

4.1 Llamadas a funciones

En el contexto de la programación, una *función* es una secuencia de sentencias que realizan una operación y que reciben un nombre. Cuando se define una función, se especifica el nombre y la secuencia de sentencias. Más adelante, se puede "llamar" a la función por ese nombre. Ya hemos visto un ejemplo de una *llamada a una función*:

```
>>> type(32)
<class 'int'>
```

El nombre de la función es `type`. La expresión entre paréntesis recibe el nombre de *argumento* de la función. El argumento es un valor o variable que se pasa a la función como parámetro de entrada. El resultado de la función `type` es el tipo del argumento.

Es habitual decir que una función "toma" (o recibe) un argumento y "retorna" (o devuelve) un resultado. El resultado se llama *valor de retorno*.

4.2 Funciones internas

Python proporciona un número importante de funciones internas, que pueden ser usadas sin necesidad de tener que definirlas previamente. Los creadores de Python han escrito un conjunto de funciones para resolver problemas comunes y las han incluido en Python para que las podamos utilizar.

Las funciones `max` y `min` nos darán respectivamente el valor mayor y menor de una lista:

```
>>> max('¡Hola, mundo!')
'u'
>>> min('¡Hola, mundo!')
' '
>>>
```

La función `max` nos dice cuál es el "carácter más grande" de la cadena (que resulta ser la letra "u"), mientras que la función `min` nos muestra el carácter más pequeño (que en ese caso es un espacio).

Otra función interna muy común es `len`, que nos dice cuántos elementos hay en su argumento. Si el argumento de `len` es una cadena, nos devuelve el número de caracteres que hay en la cadena.

```
>>> len('Hola, mundo')
11
>>>
```

Estas funciones no se limitan a buscar en cadenas. Pueden operar con cualquier conjunto de valores, como veremos en los siguientes capítulos.

Se deben tratar los nombres de las funciones internas como si fueran palabras reservadas (es decir, evita usar "max" como nombre para una variable).

4.3 Funciones de conversión de tipos

Python también proporciona funciones internas que convierten valores de un tipo a otro. La función `int` toma cualquier valor y lo convierte en un entero, si puede, o se queja si no puede:

```
>>> int('32')
32
>>> int('Hola')
ValueError: invalid literal for int() with base 10: 'Hola'
```

`int` puede convertir valores en punto flotante a enteros, pero no los redondea; simplemente corta y descarta la parte decimal:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

`float` convierte enteros y cadenas en números de punto flotante:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

Finalmente, `str` convierte su argumento en una cadena:

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

4.4 Funciones matemáticas

Python tiene un módulo matemático (`math`), que proporciona la mayoría de las funciones matemáticas habituales. Antes de que podamos utilizar el módulo, deberemos importarlo:

```
>>> import math
```

Esta sentencia crea un *objeto módulo* llamado math. Si se imprime el objeto módulo, se obtiene cierta información sobre él:

```
>>> print(math)
<module 'math' (built-in)>
```

El objeto módulo contiene las funciones y variables definidas en el módulo. Para acceder a una de esas funciones, es necesario especificar el nombre del módulo y el nombre de la función, separados por un punto (también conocido en inglés como *peŕiod*). Este formato recibe el nombre de *notación punto*.

```
>>> relacion = int_senal / int_ruido
>>> decibelios = 10 * math.log10(relacion)

>>> radianes = 0.7
>>> altura = math.sin(radianes)
```

El primer ejemplo calcula el logaritmo en base 10 de la relación señal-ruido. El módulo math también proporciona una función llamada `log` que calcula logaritmos en base `e`.

El segundo ejemplo calcula el seno de la variable `radianes`. El nombre de la variable es una pista de que `sin` y las otras funciones trigonométricas (`cos`, `tan`, etc.) toman argumentos en radianes. Para convertir de grados a radianes, hay que dividir por 360 y multiplicar por 2π:

```
>>> grados = 45
>>> radianes = grados / 360.0 * 2 * math.pi
>>> math.sin(radianes)
0.7071067811865476
```

La expresión `math.pi` toma la variable `pi` del módulo math. El valor de esa variable es una aproximación de π, con una precisión de unos 15 dígitos.

Si sabes de trigonometría, puedes comprobar el resultado anterior, comparándolo con la raíz cuadrada de dos dividida por dos:

```
>>> math.sqrt(2) / 2.0
0.7071067811865476
```

4.5 Números aleatorios

A partir de las mismas entradas, la mayoría de los programas generarán las mismas salidas cada vez, que es lo que llamamos comportamiento *determinista*. El determinismo normalmente es algo bueno, ya que esperamos que la misma operación nos proporcione siempre el mismo resultado. Para ciertas aplicaciones, sin embargo, querremos que el resultado sea impredecible. Los juegos son el ejemplo obvio, pero hay más.

Conseguir que un programa sea realmente no-determinista no resulta tan fácil, pero hay modos de hacer que al menos lo parezca. Una de ellos es usar *algoritmos* que generen números *pseudoaleatorios*. Los números pseudoaleatorios no son verdaderamente aleatorios, ya que son generados por una operación determinista, pero si sólo nos fijamos en los números resulta casi imposible distinguirlos de los aleatorios de verdad.

El módulo `random` proporciona funciones que generan números pseudoaleatorios (a los que simplemente llamaremos "aleatorios" de ahora en adelante).

La función `random` devuelve un número flotante aleatorio entre 0.0 y 1.0 (incluyendo 0.0, pero no 1.0). Cada vez que se llama a `random`, se obtiene el número siguiente de una larga serie. Para ver un ejemplo, ejecuta este bucle:

```
import random

for i in range(10):
    x = random.random()
    print(x)
```

Este programa produce la siguiente lista de 10 números aleatorios entre 0.0 y hasta (pero no incluyendo) 1.0.

```
0.11132867921152356
0.5950949227890241
0.04820265884996877
0.841003109276478
0.997914947094958
0.04842330803368111
0.7416295948208405
0.510535245390327
0.27447040171978143
0.028511805472785867
```

Ejercicio 1: Ejecuta el programa en tu sistema y observa qué números obtienes.

La función `random` es solamente una de las muchas que trabajan con números aleatorios. La función `randint` toma los parámetros `inferior` y `superior`, y devuelve un entero entre `inferior` y `superior` (incluyendo ambos extremos).

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

Para elegir un elemento de una secuencia aleatoriamente, se puede usar `choice`:

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

El módulo `random` también proporciona funciones para generar valores aleatorios de varias distribuciones continuas, incluyendo gaussiana, exponencial, gamma, y unas cuantas más.

4.6 Añadiendo funciones nuevas

Hasta ahora, sólo hemos estado usando las funciones que vienen incorporadas en Python, pero es posible añadir también funciones nuevas. Una *definición de función* especifica el nombre de una función nueva y la secuencia de sentencias que se ejecutan cuando esa función es llamada. Una vez definida una función, se puede reutilizar una y otra vez a lo largo de todo el programa.

He aquí un ejemplo:

```
def muestra_estribillo():
    print('Soy un leñador, qué alegría.')
    print('Duermo toda la noche y trabajo todo el día.')
```

`def` es una palabra clave que indica que se trata de una definición de función. El nombre de la función es `muestra_estribillo`. Las reglas para los nombres de las funciones son los mismos que para las variables: se pueden usar letras, números y algunos signos de puntuación, pero el primer carácter no puede ser un número. No se puede usar una palabra clave como nombre de una función, y se debería evitar también tener una variable y una función con el mismo nombre.

Los paréntesis vacíos después del nombre indican que esta función no toma ningún argumento. Más tarde construiremos funciones que reciban argumentos de entrada.

La primera línea de la definición de la función es llamada la *cabecera*; el resto se llama el *cuerpo*. La cabecera debe terminar con dos-puntos (:), y el cuerpo debe ir indentado. Por convención, el indentado es siempre de cuatro espacios. El cuerpo puede contener cualquier número de sentencias.

Las cadenas en la sentencia print están encerradas entre comillas. Da igual utilizar comillas simples que dobles; la mayoría de la gente prefiere comillas simples, excepto en aquellos casos en los que una comilla simple (que también se usa como apostrofe) aparece en medio de la cadena.

Si escribes una definición de función en modo interactivo, el intérprete mostrará puntos suspensivos (. . .) para informarte de que la definición no está completa:

```
>>> def muestra_estribillo():
...     print 'Soy un leñador, qué alegría.'
...     print 'Duermo toda la noche y trabajo todo el día.'
...
```

Para finalizar la función, debes introducir una línea vacía (esto no es necesario en un script).

Al definir una función se crea una variable con el mismo nombre.

```
>>> print(muestra_estribillo)
<function muestra_estribillo at 0xb7e99e9c>
>>> print(type(muestra_estribillo))
<type 'function'>
```

El valor de `muestra_estribillo` es *function object* (objeto función), que tiene como tipo "function".

La sintaxis para llamar a nuestra nueva función es la misma que usamos para las funciones internas:

```
>>> muestra_estribillo()
Soy un leñador, qué alegría.
Duermo toda la noche y trabajo todo el día.
```

Una vez que se ha definido una función, puede usarse dentro de otra. Por ejemplo, para repetir el estribillo anterior, podríamos escribir una función llamada `repite_estribillo`:

```
def repite_estribillo():
    muestra_estribillo()
    muestra_estribillo()
```

Y después llamar a `repite_estribillo`:

```
>>> repite_estribillo()
Soy un leñador, qué alegría.
Duermo toda la noche y trabajo todo el día.
Soy un leñador, qué alegría.
Duermo toda la noche y trabajo todo el día.
```

Pero en realidad la canción no es así.

4.7 Definición y usos

Reuniendo los fragmentos de código de las secciones anteriores, el programa completo sería algo como esto:

```
def muestra_estribillo():
    print('Soy un leñador, que alegría.')
    print('Duermo toda la noche y trabajo todo el día.')

def repite_estribillo():
    muestra_estribillo()
    muestra_estribillo()

repite_estribillo()

# Code: http://www.py4e.com/code3/lyrics.py
```

Este programa contiene dos definiciones de funciones: `muestra_estribillo` y `repite_estribillo`. Las definiciones de funciones son ejecutadas exactamente igual que cualquier otra sentencia, pero su resultado consiste en crear objetos del tipo función. Las sentencias dentro de cada función son ejecutadas solamente cuando se llama a esa función, y la definición de una función no genera ninguna salida.

Como ya te imaginarás, es necesario crear una función antes de que se pueda ejecutar. En otras palabras, la definición de la función debe ser ejecutada antes de que la función se llame por primera vez.

Ejercicio 2: Desplaza la última línea del programa anterior hacia arriba, de modo que la llamada a la función aparezca antes que las definiciones. Ejecuta el programa y observa qué mensaje de error obtienes.

Ejercicio 3: Desplaza la llamada de la función de nuevo hacia el final, y coloca la definición de `muestra_estribillo` después de la definición de `repite_estribillo`. ¿Qué ocurre cuando haces funcionar ese programa?

4.8 Flujo de ejecución

Para asegurarnos de que una función está definida antes de usarla por primera vez, es necesario saber el orden en que las sentencias son ejecutadas, que es lo que llamamos el *flujo de ejecución*.

La ejecución siempre comienza en la primera sentencia del programa. Las sentencias son ejecutadas una por una, en orden de arriba hacia abajo.

Las *definiciones* de funciones no alteran el flujo de la ejecución del programa, pero recuerda que las sentencias dentro de una función no son ejecutadas hasta que se llama a esa función.

Una llamada a una función es como un desvío en el flujo de la ejecución. En vez de pasar a la siguiente sentencia, el flujo salta al cuerpo de la función, ejecuta todas las sentencias que hay allí, y después vuelve al punto donde lo dejó.

Todo esto parece bastante sencillo, hasta que uno recuerda que una función puede llamar a otra. Cuando está en mitad de una función, el programa puede tener que ejecutar las sentencias de otra función. Pero cuando está ejecutando esa nueva función, ¡tal vez haya que ejecutar todavía más funciones!

Afortunadamente, Python es capaz de llevar el seguimiento de dónde se encuentra en cada momento, de modo que cada vez que completa la ejecución de una función, el programa vuelve al punto donde lo dejó en la función que había llamado a esa. Cuando esto le lleva hasta el final del programa, simplemente termina.

¿Cuál es la moraleja de esta extraña historia? Cuando leas un programa, no siempre te convendrá hacerlo de arriba a abajo. A veces tiene más sentido seguir el flujo de la ejecución.

4.9 Parámetros y argumentos

Algunas de las funciones internas que hemos visto necesitan argumentos. Por ejemplo, cuando se llama a `math.sin`, se le pasa un número como argumento. Algunas funciones necesitan más de un argumento: `math.pow` toma dos, la base y el exponente.

Dentro de las funciones, los argumentos son asignados a variables llamadas *parámetros*. A continuación mostramos un ejemplo de una función definida por el usuario que recibe un argumento:

```
def muestra_dos_veces(bruce):
    print(bruce)
    print(bruce)
```

Esta función asigna el argumento a un parámetro llamado `bruce`. Cuando la función es llamada, imprime el valor del parámetro (sea éste lo que sea) dos veces.

Esta función funciona con cualquier valor que pueda ser mostrado en pantalla.

```
>>> muestra_dos_veces('Spam')
Spam
Spam
>>> muestra_dos_veces(17)
17
17
>>> muestra_dos_veces(math.pi)
3.14159265359
3.14159265359
```

Las mismas reglas de composición que se aplican a las funciones internas, también se aplican a las funciones definidas por el usuario, de modo que podemos usar cualquier tipo de expresión como argumento para `muestra_dos_veces`:

```
>>> muestra_dos_veces('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> muestra_dos_veces(math.cos(math.pi))
-1.0
-1.0
```

El argumento es evaluado antes de que la función sea llamada, así que en los ejemplos, la expresión `Spam *4` y `math.cos(math.pi)` son evaluadas sólo una vez.

También se puede usar una variable como argumento:

```
>>> michael = 'Eric, la medio-abeja.'
>>> muestra_dos_veces(michael)
Eric, la medio-abeja.
Eric, la medio-abeja.
```

El nombre de la variable que pasamos como argumento, (`michael`) no tiene nada que ver con el nombre del parámetro (`bruce`). No importa cómo se haya llamado al valor en origen (en la llamada); dentro de `muestra_dos_veces`, siempre se llamará `bruce`.

4.10 Funciones productivas y funciones estériles

Algunas de las funciones que estamos usando, como las matemáticas, producen resultados; a falta de un nombre mejor, las llamaremos *funciones productivas* (fruitful functions). Otras funciones, como `muestra_dos_veces`, realizan una acción, pero no devuelven un valor. A esas las llamaremos *funciones estériles* (void functions).

Cuando llamas a una función productiva, casi siempre querrás hacer luego algo con el resultado; por ejemplo, puede que quieras asignarlo a una variable o usarlo como parte de una expresión:

```
x = math.cos(radians)
aurea = (math.sqrt(5) + 1) / 2
```

Cuando llamas a una función en modo interactivo, Python muestra el resultado:

```
>>> math.sqrt(5)
2.23606797749979
```

Pero en un script, si llamas a una función productiva y no almacenas el resultado de la misma en una variable, ¡el valor de retorno se desvanece en la niebla!

```
math.sqrt(5)
```

Este script calcula la raíz cuadrada de 5, pero dado que no almacena el resultado en una variable ni lo muestra, no resulta en realidad muy útil.

Las funciones estériles pueden mostrar algo en la pantalla o tener cualquier otro efecto, pero no devuelven un valor. Si intentas asignar el resultado a una variable, obtendrás un valor especial llamado `None` (nada).

```
>>> resultado = muestra_dos_veces('Bing')
Bing
Bing
>>> print(resultado)
None
```

El valor `None` no es el mismo que la cadena "None". Es un valor especial que tiene su propio tipo:

```
>>> print(type(None))
<class 'NoneType'>
```

Para devolver un resultado desde una función, usamos la sentencia `return` dentro de ella. Por ejemplo, podemos crear una función muy simple llamada `sumados`, que suma dos números y devuelve el resultado.

```
def sumados(a, b):
    suma = a + b
    return suma

x = sumados(3, 5)
print(x)

# Code: http://www.py4e.com/code3/addtwo.py
```

Cuando se ejecuta este script, la sentencia `print` mostrará "8", ya que la función `sumados` ha sido llamada con 3 y 5 como argumentos. Dentro de la función, los parámetros `a` y `b` equivaldrán a 3 y a 5 respectivamente. La función calculó la suma de ambos número y la guardó en una variable local a la función llamada `suma`. Después usó la sentencia `return` para enviar el valor calculado de vuelta al código de llamada como resultado de la función, que fue asignado a la variable `x` y mostrado en pantalla.

4.11 ¿Por qué funciones?

Puede no estar muy claro por qué merece la pena molestarse en dividir un programa en funciones. Existen varias razones:

- El crear una función nueva te da la oportunidad de dar nombre a un grupo de sentencias, lo cual hace tu programa más fácil de leer, entender y depurar.
- Las funciones pueden hacer un programa más pequeño, al eliminar código repetido. Además, si quieres realizar cualquier cambio en el futuro, sólo tendrás que hacerlo en un único lugar.
- Dividir un programa largo en funciones te permite depurar las partes de una en una y luego ensamblarlas juntas en una sola pieza.
- Las funciones bien diseñadas a menudo resultan útiles para otros muchos programas. Una vez que has escrito y depurado una, puedes reutilizarla.

A lo largo del resto del libro, a menudo usaremos una definición de función para explicar un concepto. Parte de la habilidad de crear y usar funciones consiste en llegar a tener una función que represente correctamente una idea, como "encontrar el valor más pequeño en una lista de valores". Más adelante te mostraremos el código para encontrar el valor más pequeño de una lista de valores y te lo presentaremos como una función llamada `min`, que toma una lista de valores como argumento y devuelve el menor valor de esa lista.

4.12 Depuración

Si estás usando un editor de texto para escribir tus propios scripts, puede que tengas problemas con los espacios y tabulaciones. El mejor modo de evitar esos problemas es usar espacios exclusivamente (no tabulaciones). La mayoría de los editores de texto que reconocen Python lo hacen así por defecto, aunque hay algunos que no.

Las tabulaciones y los espacios normalmente son invisibles, lo cual hace que sea difícil depurar los errores que se pueden producir, así que mejor busca un editor que gestione el indentado por ti.

Tampoco te olvides de guardar tu programa antes de hacerlo funcionar. Algunos entornos de desarrollo lo hacen automáticamente, pero otros no. En ese caso, el programa que estás viendo en el editor de texto puede no ser el mismo que estás ejecutando en realidad.

¡La depuración puede llevar mucho tiempo si estás haciendo funcionar el mismo programa con errores una y otra vez!

Asegúrate de que el código que estás examinando es el mismo que estás ejecutando. Si no estás seguro, pon algo como `print("hola")` al principio del programa y hazlo funcionar de nuevo. Si no ves `hola` en la pantalla, ¡es que no estás ejecutando el programa correcto!

4.13 Glosario

algoritmo Un proceso general para resolver una categoría de problemas.

argumento Un valor proporcionado a una función cuando ésta es llamada. Ese valor se asigna al parámetro correspondiente en la función.

cabecera La primera línea de una definición de función.

cuerpo La secuencia de sentencias dentro de la definición de una función.

composición Uso de una expresión o sentencia como parte de otra más larga,

definición de función Una sentencia que crea una función nueva, especificando su nombre, parámetros, y las sentencias que ejecuta.

determinístico Perteneciente a un programa que hace lo mismo cada vez que se ejecuta, a partir de las mismas entradas.

función Una secuencia de sentencias con un nombre que realizan alguna operación útil. Las funciones pueden tomar argumentos o no, y pueden producir un resultado o no.

función productiva (fruitful function) Una función que devuelve un valor.

función estéril (void function) Una función que no devuelve ningún valor.

flujo de ejecución El orden en el cual se ejecutan las sentencias durante el funcionamiento de un programa.

llamada a función Una sentencia que ejecuta una función. Consiste en el nombre de la función seguido por una lista de argumentos.

notación punto La sintaxis para llamar a una función de otro módulo, especificando el nombre del módulo seguido por un punto y el nombre de la función.

objeto función Un valor creado por una definición de función. El nombre de la función es una variable que se refiere al objeto función.

objeto módulo Un valor creado por una sentencia `import`, que proporciona acceso a los datos y código definidos en un módulo.

parámetro Un nombre usado dentro de una función para referirse al valor pasado como argumento.

pseudoaleatorio Perteneciente a una secuencia de números que parecen ser aleatorios, pero son generados por un programa determinista.

sentencia import Una sentencia que lee un archivo módulo y crea un objeto módulo.

valor de retorno El resultado de una función. Si una llamada a una función es usada como una expresión, el valor de retorno es el valor de la expresión.

4.14 Ejercicios

Ejercicio 4: ¿Cuál es la utilidad de la palabra clave "def" en Python?

a) Es una jerga que significa "este código es realmente estupendo"
b) Indica el comienzo de una función
c) Indica que la siguiente sección de código indentado debe ser almacenada para usarla más tarde
d) b y c son correctas ambas
e) Ninguna de las anteriores

Ejercicio 5: ¿Qué mostrará en pantalla el siguiente programa Python?

```
def fred():
   print("Zap")

def jane():
   print("ABC")

jane()
fred()
jane()
```

a) Zap ABC jane fred jane
b) Zap ABC Zap
c) ABC Zap jane
d) ABC Zap ABC
e) Zap Zap Zap

Ejercicio 6: Reescribe el programa de cálculo del salario, con tarifa-y-media para las horas extras, y crea una función llamada `calculo_salario` que reciba dos parámetros (`horas` y `tarifa`).

```
Introduzca Horas: 45
Introduzca Tarifa: 10
Salario: 475.0
```

Ejercicio 7: Reescribe el programa de calificaciones del capítulo anterior usando una función llamada `calcula_calificacion`, que reciba una puntuación como parámetro y devuelva una calificación como cadena.

```
Puntuación Calificación
> 0.9      Sobresaliente
> 0.8      Notable
> 0.7      Bien
> 0.6      Suficiente
<= 0.6     Insuficiente


Introduzca puntuación: 0.95
Sobresaliente
```

```
Introduzca puntuación: perfecto
Puntuación incorrecta


Introduzca puntuación: 10.0
Puntuación incorrecta


Introduzca puntuación: 0.75
Bien


Introduzca puntuación: 0.5
Insuficiente
```

Ejecuta el programa repetidamente para probar con varios valores de entrada diferentes.

Chapter 5

Iteración

5.1 Actualización de variables

Uno de los usos habituales de las sentencias de asignación consiste en realizar una actualización sobre una variable – en la cual el valor nuevo de esa variable depende del antiguo.

```
x = x + 1
```

Esto quiere decir "'toma el valor actual de `x`, añádele 1, y luego actualiza `x` con el nuevo valor".

Si intentas actualizar una variable que no existe, obtendrás un error, ya que Python evalúa el lado derecho antes de asignar el valor a `x`:

```
>>> x = x + 1
NameError: name 'x' is not defined
```

Antes de que puedas actualizar una variable, debes *inicializarla*, normalmente mediante una simple asignación:

```
>>> x = 0
>>> x = x + 1
```

Actualizar una variable añadiéndole 1 se denomina *incrementar*; restarle 1 recibe el nombre de *decrementar* (o disminuir).

5.2 La sentencia `while`

Los PCs se suelen utilizar a menudo para automatizar tareas repetitivas. Repetir tareas idénticas o muy similares sin cometer errores es algo que a las máquinas se les da bien y en cambio a las personas no. Como las iteraciones resultan tan

habituales, Python proporciona varias características en su lenguaje para hacerlas más sencillas.

Una forma de iteración en Python es la sentencia `while`. He aquí un programa sencillo que cuenta hacia atrás desde cinco y luego dice "¡Despegue!".

```
n = 5
while n > 0:
    print(n)
    n = n - 1
print('¡Despegue!')
```

Casi se puede leer la sentencia `while` como si estuviera escrita en inglés. Significa, "Mientras `n` sea mayor que 0, muestra el valor de `n` y luego reduce el valor de `n` en 1 unidad. Cuando llegues a 0, sal de la sentencia `while` y muestra la palabra `¡Despegue!`"

Éste es el flujo de ejecución de la sentencia `while`, explicado de un modo más formal:

1. Se evalúa la condición, obteniendo `Verdadero` or `Falso`.
2. Si la condición es falsa, se sale de la sentencia `while` y se continúa la ejecución en la siguiente sentencia.
3. Si la condición es verdadera, se ejecuta el cuerpo del `while` y luego se vuelve al paso 1.

Este tipo de flujo recibe el nombre de *bucle*, ya que el tercer paso enlaza de nuevo con el primero. Cada vez que se ejecuta el cuerpo del bucle se dice que realizamos una *iteración*. Para el bucle anterior, podríamos decir que "ha tenido cinco iteraciones", lo que significa que el cuerpo del bucle se ha ejecutado cinco veces.

El cuerpo del bucle debe cambiar el valor de una o más variables, de modo que la condición pueda en algún momento evaluarse como falsa y el bucle termine. La variable que cambia cada vez que el bucle se ejecuta y controla cuándo termina éste, recibe el nombre de *variable de iteración*. Si no hay variable de iteración, el bucle se repetirá para siempre, resultando así un *bucle infinito*.

5.3 Bucles infinitos

Una fuente de diversión sin fin para los programadores es la constatación de que las instrucciones del champú: "Enjabone, aclare, repita", son un bucle infinito, ya que no hay una *variable de iteración* que diga cuántas veces debe ejecutarse el proceso.

En el caso de una `cuenta atrás`, podemos verificar que el bucle termina, ya que sabemos que el valor de `n` es finito, y podemos ver que ese valor se va haciendo más pequeño cada vez que se repite el bucle, de modo que en algún momento llegará a 0. Otras veces un bucle es obviamente infinito, porque no tiene ninguna variable de iteración.

5.4 "Bucles infinitos" y break

A veces no se sabe si hay que terminar un bucle hasta que se ha recorrido la mitad del cuerpo del mismo. En ese caso se puede crear un bucle infinito a propósito y usar la sentencia `break` para salir fuera de él cuando se desee.

El bucle siguiente es, obviamente, un *bucle infinito*, porque la expresión lógica de la sentencia `while` es simplemente la constante lógica `True (verdadero)`;

```
n = 10
while True:
    print(n, end=' ')
    n = n - 1
print('¡Terminado!')
```

Si cometes el error de ejecutar este código, aprenderás rápidamente cómo detener un proceso de Python bloqueado en el sistema, o tendrás que localizar dónde se encuentra el botón de apagado de tu equipo. Este programa funcionará para siempre, o hasta que la batería del equipo se termine, ya que la expresión lógica al principio del bucle es siempre cierta, en virtud del hecho de que esa expresión es precisamente el valor constante `True`.

A pesar de que en este caso se trata de un bucle infinito inútil, se puede usar ese diseño para construir bucles útiles, siempre que se tenga la precaución de añadir código en el cuerpo del bucle para salir explícitamente, usando `break` cuando se haya alcanzado la condición de salida.

Por ejemplo, supón que quieres recoger entradas de texto del usuario hasta que éste escriba `fin`. Podrías escribir:

```
while True:
    linea = input('> ')
    if linea == 'fin':
        break
    print(linea)
print('¡Terminado!')

# Code: http://www.py4e.com/code3/copytildone1.py
```

La condición del bucle es `True`, lo cual es verdadero siempre, así que el bucle se repetirá hasta que se ejecute la sentencia break.

Cada vez que se entre en el bucle, se pedirá una entrada al usuario. Si el usuario escribe `fin`, la sentencia `break` hará que se salga del bucle. En cualquier otro caso, el programa repetirá cualquier cosa que el usuario escriba y volverá al principio del bucle. Éste es un ejemplo de su funcionamiento:

```
> hola a todos
hola a todos
> he terminado
he terminado
> fin
¡Terminado!
```

Este modo de escribir bucles `while` es habitual, ya que así se puede comprobar la condición en cualquier punto del bucle (no sólo al principio), y se puede expresar la condición de parada afirmativamente ("detente cuando ocurra..."), en vez de tener que hacerlo con lógica negativa ("sigue haciéndolo hasta que ocurra...").

5.5 Finalizar iteraciones con `continue`

Algunas veces, estando dentro de un bucle se necesita terminar con la iteración actual y saltar a la siguiente de forma inmediata. En ese caso se puede utilizar la sentencia `continue` para pasar a la siguiente iteración sin terminar la ejecución del cuerpo del bucle para la actual.

A continuación se muestra un ejemplo de un bucle que repite lo que recibe como entrada hasta que el usuario escribe "fin", pero trata las líneas que empiezan por el carácter almohadilla como líneas que no deben mostrarse en pantalla (algo parecido a lo que hace Python con los comentarios).

```
while True:
    linea = input('> ')
    if linea[0] == '#' :
        continue
    if linea == 'fin':
        break
    print(linea)
print('¡Terminado!')

# Code: http://www.py4e.com/code3/copytildone2.py
```

He aquí una ejecución de ejemplo de ese nuevo programa con la sentencia `continue` añadida.

```
> hola a todos
hola a todos
> # no imprimas esto
> ¡imprime esto!
¡imprime esto!
> fin
¡Terminado!
```

Todas las líneas se imprimen en pantalla, excepto la que comienza con el símbolo de almohadilla, ya que en ese caso se ejecuta `continue`, finaliza la iteración actual y salta de vuelta a la sentencia `while` para comenzar la siguiente iteración, de modo que que se omite la sentencia `print`.

5.6 Bucles definidos usando `for`

A veces se desea repetir un bucle a través de un *conjunto* de cosas, como una lista de palabras, las líneas de un archivo, o una lista de números. Cuando se

tiene una lista de cosas para recorrer, se puede construir un bucle *definido* usando una sentencia `for`. A la sentencia `while` se la llama un bucle *indefinido*, porque simplemente se repite hasta que cierta condición se hace `Falsa`, mientras que el bucle `for` se repite a través de un conjunto conocido de elementos, de modo que ejecuta tantas iteraciones como elementos hay en el conjunto.

La sintaxis de un bucle `for` es similar a la del bucle `while`, en ella hay una sentencia `for` y un cuerpo que se repite:

```
amigos = ['Joseph', 'Glenn', 'Sally']
for amigo in amigos:
    print('Feliz año nuevo:', amigo)
print('¡Terminado!')
```

En términos de Python, la variable `amigos` es una lista[1] de tres cadenas y el bucle `for` se mueve recorriendo la lista y ejecuta su cuerpo una vez para cada una de las tres cadenas en la lista, produciendo esta salida:

```
Feliz año nuevo: Joseph
Feliz año nuevo: Glenn
Feliz año nuevo: Sally
¡Terminado!
```

La traducción de este bucle `for` al español no es tan directa como en el caso del `while`, pero si piensas en los amigos como un *conjunto*, sería algo así como: "Ejecuta las sentencias en el cuerpo del bucle una vez *para (for)* cada amigo que esté *en (in)* el conjunto llamado amigos."

Revisando el bucle `for`, *for* e *in* son palabras reservadas de Python, mientras que `amigo` y `amigos` son variables.

```
for amigo in amigos:
    print('Feliz año nuevo::', amigo)
```

En concreto, `amigo` es la *variable de iteración* para el bucle for. La variable `amigo` cambia para cada iteración del bucle y controla cuándo se termina el bucle `for`. La *variable de iteracion* se desplaza sucesivamente a través de las tres cadenas almacenadas en la variable `amigos`.

5.7 Diseños de bucles

A menudo se usa un bucle `for` o `while` para movernos a través de una lista de elementos o el contenido de un archivo y se busca algo, como el valor más grande o el más pequeño de los datos que estamos revisando.

Los bucles generalmente se construyen así:

- Se inicializan una o más variables antes de que el bucle comience

[1] Examinaremos las listas con más detalle en un capítulo posterior.

- Se realiza alguna operación con cada elemento en el cuerpo del bucle, posiblemente cambiando las variables dentro de ese cuerpo.
- Se revisan las variables resultantes cuando el bucle se completa

Usaremos ahora una lista de números para demostrar los conceptos y construcción de estos diseños de bucles.

5.7.1 Bucles de recuento y suma

Por ejemplo, para contar el número de elementos en una lista, podemos escribir el siguiente bucle `for`:

```
contador = 0
for valor in [3, 41, 12, 9, 74, 15]:
    contador = contador + 1
print('Num. elementos: ', contador)
```

Ajustamos la variable `contador` a cero antes de que el bucle comience, después escribimos un bucle `for` para movernos a través de la lista de números. Nuestra variable de *iteración* se llama `valor`, y dado que no usamos `valor` dentro del bucle, lo único que hace es controlar el bucle y hacer que el cuerpo del mismo sea ejecutado una vez para cada uno de los valores de la lista.

En el cuerpo del bucle, añadimos 1 al valor actual de `contador` para cada uno de los valores de la lista. Mientras el bucle se está ejecutando, el valor de `contador` es la cantidad de valores que se hayan visto "hasta ese momento".

Una vez el bucle se completa, el valor de `contador` es el número total de elementos. El número total "cae en nuestro poder" al final del bucle. Se construye el bucle de modo que obtengamos lo que queremos cuando éste termina.

Otro bucle similar, que calcula el total de un conjunto de números, se muestra a continuación:

```
total = 0
for valor in [3, 41, 12, 9, 74, 15]:
    total = total + valor
print('Total: ', total)
```

En este bucle, *sí* utilizamos la *variable de iteración*. En vez de añadir simplemente uno a `contador` como en el bucle previo, ahora durante cada iteración del bucle añadimos el número actual (3, 41, 12, etc.) al total en ese momento. Si piensas en la variable `total`, ésta contiene la "suma parcial de valores hasta ese momento". Así que antes de que el bucle comience, `total` es cero, porque aún no se ha examinado ningún valor. Durante el bucle, `total` es la suma parcial, y al final del bucle, `total` es la suma total definitiva de todos los valores de la lista.

Cuando el bucle se ejecuta, `total` acumula la suma de los elementos; una variable que se usa de este modo recibe a veces el nombre de *acumulador*.

Ni el bucle que cuenta los elementos ni el que los suma resultan particularmente útiles en la práctica, dado que existen las funciones internas `len()` y `sum()` que cuentan el número de elementos de una lista y el total de elementos en la misma respectivamente.

5.7.2 Bucles de máximos y mínimos

Para encontrar el valor mayor de una lista o secuencia, construimos el bucle siguiente:

```
mayor = None
print('Antes:', mayor)
for valor in [3, 41, 12, 9, 74, 15]:
    if mayor is None or valor > mayor :
        mayor = valor
    print('Bucle:', valor, mayor)
print('Mayor:', mayor)
```

Cuando se ejecuta el programa, se obtiene la siguiente salida:

```
Antes: None
Bucle: 3 3
Bucle: 41 41
Bucle: 12 41
Bucle: 9 41
Bucle: 74 74
Bucle: 15 74
Mayor: 74
```

Debemos pensar en la variable `mayor` como el "mayor valor visto hasta ese momento". Antes del bucle, asignamos a `mayor` el valor `None`. `None` es un valor constante especial que se puede almacenar en una variable para indicar que la variable está "vacía".

Antes de que el bucle comience, el mayor valor visto hasta entonces es `None`, dado que no se ha visto aún ningún valor. Durante la ejecución del bucle, si `mayor` es `None`, entonces tomamos el primer valor que tenemos como el mayor hasta entonces. Se puede ver en la primera iteración, cuando el valor de `valor` es 3, mientras que `mayor` es `None`, inmediatamente hacemos que `mayor` pase a ser 3.

Tras la primera iteración, `mayor` ya no es `None`, así que la segunda parte de la expresión lógica compuesta que comprueba si `valor > mayor` se activará sólo cuando encontremos un valor que sea mayor que el "mayor hasta ese momento". Cuando encontramos un nuevo valor "mayor aún", tomamos ese nuevo valor para `mayor`. Se puede ver en la salida del programa que `mayor` pasa desde 3 a 41 y luego a 74.

Al final del bucle, se habrán revisado todos los valores y la variable `mayor` contendrá entonces el mayor valor de la lista.

Para calcular el número más pequeño, el código es muy similar con un pequeño cambio:

```
print('Antes:', menor)
for valor in [3, 41, 12, 9, 74, 15]:
    if menor is None or valor < menor:
        menor = valor
    print('Bucle:', valor, menor)
print('Menor:', menor)
```

De nuevo, `menor` es el "menor hasta ese momento" antes, durante y después de que el bucle se ejecute. Cuando el bucle se ha completado, `menor` contendrá el valor mínimo de la lista

También como en el caso del número de elementos y de la suma, las funciones internas `max()` y `min()` convierten la escritura de este tipo de bucles en innecesaria.

Lo siguiente es una versión simple de la función interna de Python `min()`:

```
def min(valores):
    menor = None
    for valor in valores:
        if menor is None or valor < menor:
            menor = valor
    return menor
```

En esta versión de la función para calcular el mínimo, hemos eliminado las sentencias `print`, de modo que sea equivalente a la función `min`, que ya está incorporada dentro de Python.

5.8 Depuración

A medida que vayas escribiendo programas más grandes, puede que notes que vas necesitando emplear cada vez más tiempo en depurarlos. Más código significa más oportunidades de cometer un error y más lugares donde los bugs pueden esconderse.

Un método para acortar el tiempo de depuración es "depurar por bisección". Por ejemplo, si hay 100 líneas en tu programa y las compruebas de una en una, te llevará 100 pasos.

En lugar de eso, intenta partir el problema por la mitad. Busca en medio del programa, o cerca de ahí, un valor intermedio que puedas comprobar. Añade una sentencia `print` (o alguna otra cosa que tenga un efecto verificable), y haz funcionar el programa.

Si en el punto medio la verificación es incorrecta, el problema debería estar en la primera mitad del programa. Si ésta es correcta, el problema estará en la segunda mitad.

Cada vez que realices una comprobación como esta, reduces a la mitad el número de líneas en las que buscar. Después de seis pasos (que son muchos menos de 100), lo habrás reducido a una o dos líneas de código, al menos en teoría.

En la práctica no siempre está claro qué es "en medio del programa", y no siempre es posible colocar ahí una verificación. No tiene sentido contar las líneas y encontrar

el punto medio exacto. En lugar de eso, piensa en lugares del programa en los cuales pueda haber errores y en lugares donde resulte fácil colocar una comprobación. Luego elige un sitio donde estimes que las oportunidades de que el bug esté por delante y las de que esté por detrás de esa comprobación son más o menos las mismas.

5.9 Glosario

acumulador Una variable usada en un bucle para sumar o acumular un resultado.

bucle infinito Un bucle en el cual la condición de terminación no se satisface nunca o para el cual no existe dicha condición de terminación.

contador Una variable usada en un bucle para contar el número de veces que algo sucede. Inicializamos el contador a cero y luego lo vamos incrementando cada vez que queramos que "cuente" algo.

decremento Una actualización que disminuye el valor de una variable.

inicializar Una asignación que da un valor inicial a una variable que va a ser después actualizada.

incremento Una actualización que aumenta el valor de una variable (a menudo en una unidad).

iteración Ejecución repetida de una serie de sentencias usando bien una función que se llama a si misma o bien un bucle.

5.10 Ejercicios

Ejercicio 1: Escribe un programa que lea repetidamente números hasta que el usuario introduzca "fin". Una vez se haya introducido "fin", muestra por pantalla el total, la cantidad de números y la media de esos números. Si el usuario introduce cualquier otra cosa que no sea un número, detecta su fallo usando `try` y `except`, muestra un mensaje de error y pasa al número siguiente.

```
Introduzca un número: 4
Introduzca un número: 5
Introduzca un número: dato erróneo
Entrada inválida
Introduzca un número: 7
Introduzca un número: fin
16 3 5.33333333333
```

Ejercicio 2: Escribe otro programa que pida una lista de números como la anterior y al final muestre por pantalla el máximo y mínimo de los números, en vez de la media.

Chapter 6

Cadenas

6.1 Una cadena es una secuencia

Una cadena es una *secuencia* de caracteres. Puedes acceder a los caracteres de uno en uno con el operador corchete:

```
>>> fruta = 'banana'
>>> letra = fruta[1]
```

La segunda sentencia extrae el carácter en la posición del indice 1 de la variable `fruta` y la asigna a la variable `letra`.

La expresión en los corchetes es llamada *índice*. El índice indica qué carácter de la secuencia quieres (de ahí el nombre).

Pero podrías no obtener lo que esperas:

```
>>> print(letra)
a
```

Para la mayoría de las personas, la primer letra de "banana" es "b", no "a". Pero en Python, el índice es un desfase desde el inicio de la cadena, y el desfase de la primera letra es cero.

```
>>> letra = fruta[0]
>>> print(letra)
b
```

Así que "b" es la letra 0 ("cero") de "banana", "a" es la letra con índice 1, y "n" es la que tiene índice 2, etc.

Puedes usar cualquier expresión, incluyendo variables y operadores, como un índice, pero el valor del índice tiene que ser un entero. De otro modo obtendrás:

```
>>> letra = fruta[1.5]
TypeError: string indices must be integers
```

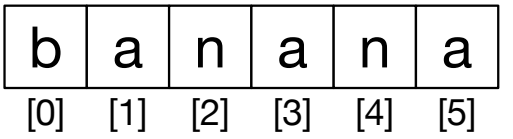

Figure 6.1: Indices de Cadenas

6.2 Obtener el tamaño de una cadena usando `len`

`len` es una función nativa que devuelve el número de caracteres en una cadena:

```
>>> fruta = 'banana'
>>> len(fruta)
6
```

Para obtener la última letra de una cadena, podrías estar tentado a probar algo como esto:

```
>>> tamaño = len(fruta)
>>> ultima = fruta[tamaño]
IndexError: string index out of range
```

La razón de que haya un `IndexError` es que ahí no hay ninguna letra en "banana" con el índice 6. Puesto que empezamos a contar desde cero, las seis letras están enumeradas desde 0 hasta 5. Para obtener el último carácter, tienes que restar 1 a `length`:

```
>>> ultima = fruta[tamaño-1]
>>> print(ultima)
a
```

Alternativamente, puedes usar índices negativos, los cuales cuentan hacia atrás desde el final de la cadena. La expresión `fruta[-1]` devuelve la última letra, `fruta[-2]` la penúltima letra, y así sucesivamente.

6.3 Recorriendo una cadena mediante un bucle

Muchos de los cálculos requieren procesar una cadena carácter por carácter. Frecuentemente empiezan desde el inicio, seleccionando cada carácter presente, haciendo algo con él, y continuando hasta el final. Este patrón de procesamiento es llamado un *recorrido*. Una manera de escribir un recorrido es con un bucle `while`:

```
indice = 0
while indice < len(fruta):
    letra = fruta[indice]
    print(letra)
    indice = indice + 1
```

Este bucle recorre la cadena e imprime cada letra en una línea cada una. La condición del bucle es `indice < len(fruta)`, así que cuando `indice` es igual al tamaño de la cadena, la condición es falsa, y el código del bucle no se ejecuta. El último carácter accedido es el que tiene el índice `len(fruta)-1`, el cual es el último carácter en la cadena.

Ejercicio 1: Escribe un bucle `while` que comience con el último carácter en la cadena y haga un recorrido hacia atrás hasta el primer carácter en la cadena, imprimiendo cada letra en una línea independiente.

Otra forma de escribir un recorrido es con un bucle `for`:

```
for caracter in fruta:
    print(caracter)
```

Cada vez que iteramos el bucle, el siguiente carácter en la cadena es asignado a la variable `caracter`. El ciclo continúa hasta que no quedan caracteres.

6.4 Parte (slicing) de una cadena

Un segmento de una cadena es llamado *parte*. Seleccionar una parte es similar a seleccionar un carácter:

```
>>> s = 'Monty Python'
>>> print(s[0:5])
Monty
>>> print(s[6:12])
Python
```

El operador [n:m] retorna la parte de la cadena desde el "n-ésimo" carácter hasta el "m-ésimo" carácter, incluyendo el primero pero excluyendo el último.

Si omites el primer índice (antes de los dos puntos), la parte comienza desde el inicio de la cadena. Si omites el segundo índice, la parte va hasta el final de la cadena:

```
>>> fruta = 'banana'
>>> fruta[:3]
'ban'
>>> fruta[3:]
'ana'
```

Si el primer índice es mayor que o igual que el segundo, el resultado es una *cadena vacía*, representado por dos comillas:

```
>>> fruta = 'banana'
>>> fruta[3:3]
''
```

Una cadena vacía no contiene caracteres y tiene un tamaño de 0, pero fuera de esto es lo mismo que cualquier otra cadena.

Ejercicio 2: Dado que `fruta` es una cadena, ¿que significa `fruta[:]`?

6.5 Los cadenas son inmutables

Puede ser tentador utilizar el operador [] en el lado izquierdo de una asignación, con la intención de cambiar un carácter en una cadena. Por ejemplo:

```
>>> saludo = 'Hola, mundo!'
>>> saludo[0] = 'J'
TypeError: 'str' object does not support item assignment
```

El "object" en este caso es la cadena y el "ítem" es el carácter que tratamos de asignar. Por ahora, un *object* es la misma cosa que un valor, pero vamos a redefinir esa definición después. Un *ítem* es uno de los valores en una secuencia.

La razón por la cual ocurre el error es que las cadenas son *inmutables*, lo cual significa que no puedes modificar una cadena existente. Lo mejor que puedes hacer es crear una nueva cadena que sea una variación del original:

```
>>> saludo = 'Hola, mundo!'
>>> nuevo_saludo = 'J' + saludo[1:]
>>> print(nuevo_saludo)
Jola, mundo!
```

Este ejemplo concatena una nueva letra a una parte de `saludo`. Esto no tiene efecto sobre la cadena original.

6.6 Iterando y contando

El siguiente programa cuenta el número de veces que la letra "a" aparece en una cadena:

```
palabra = 'banana'
contador = 0
for letra in palabra:
    if letra == 'a':
        contador = contador + 1
print(contador)
```

Este programa demuestra otro patrón de computación llamado *contador*. La variable `contador` es inicializada a 0 y después se incrementa cada vez que una "a" es encontrada. Cuando el bucle termina, `contador` contiene el resultado: el número total de a's.

Ejercicio 3: Encapsula este código en una función llamada `cuenta`, y hazla genérica de tal modo que pueda aceptar una cadena y una letra como argumentos.

6.7 El operador in

La palabra in es un operador booleano que toma dos cadenas y regresa True si la primera cadena aparece como una subcadena de la segunda:

```
>>> 'a' in 'banana'
True
>>> 'semilla' in 'banana'
False
```

6.8 Comparación de cadenas

Los operadores de comparación funcionan en cadenas. Para ver si dos cadenas son iguales:

```
if palabra == 'banana':
    print('Muy bien, bananas.')
```

Otras operaciones de comparación son útiles para poner palabras en orden alfabético:

```
if palabra < 'banana':
    print('Tu palabra, ' + palabra + ', está antes de banana.')
elif palabra > 'banana':
    print('Tu palabra, ' + palabra + ', está después de banana.')
else:
    print('Muy bien, bananas.')
```

Python no maneja letras mayúsculas y minúsculas de la misma forma que la gente lo hace. Todas las letras mayúsculas van antes que todas las letras minúsculas, por ejemplo:

```
Tu palabra, Piña, está antes que banana.
```

Una forma común de manejar este problema es convertir cadenas a un formato estándar, como todas a minúsculas, antes de llevar a cabo la comparación. Ten en cuenta eso en caso de que tengas que defenderte contra un hombre armado con una Piña.

6.9 Métodos de cadenas

Los cadenas son un ejemplo de *objetos* en Python. Un objeto contiene tanto datos (el valor de la cadena misma) como *métodos*, los cuales son efectivamente funciones que están implementadas dentro del objeto y que están disponibles para cualquier *instancia* del objeto.

Python tiene una función llamada dir la cual lista los métodos disponibles para un objeto. La función type muestra el tipo de un objeto y la función dir muestra los métodos disponibles.

```
>>> cosa = 'Hola mundo'
>>> type(cosa)
<class 'str'>
>>> dir(cosa)
['capitalize', 'casefold', 'center', 'count', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'format_map',
'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit',
'isidentifier', 'islower', 'isnumeric', 'isprintable',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',
'lstrip', 'maketrans', 'partition', 'replace', 'rfind',
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
'split', 'splitlines', 'startswith', 'strip', 'swapcase',
'title', 'translate', 'upper', 'zfill']
>>> help(str.capitalize)
Help on method_descriptor:

capitalize(...)
    S.capitalize() -> str

    Return a capitalized version of S, i.e. make the first character
    have upper case and the rest lower case.
>>>
```

Aunque la función `dir` lista los métodos y puedes usar la función `help` para obtener una breve documentación de un método, una mejor fuente de documentación para los métodos de cadenas se puede encontrar en https://docs.python.org/library/stdtypes.html#string-methods.

Llamar a un *método* es similar a llamar una función (esta toma argumentos y devuelve un valor) pero la sintaxis es diferente. Llamamos a un método uniendo el nombre del método al de la variable, usando un punto como delimitador.

Por ejemplo, el método `upper` toma una cadena y devuelve una nueva cadena con todas las letras en mayúscula:

En vez de la sintaxis de función `upper(word)`, éste utiliza la sintaxis de método `word.upper()`.

```
>>> palabra = 'banana'
>>> nueva_palabra = palabra.upper()
>>> print(nueva_palabra)
BANANA
```

Esta forma de notación con punto especifica el nombre del método, `upper`, y el nombre de la cadena al que se le aplicará el método, `palabra`. Los paréntesis vacíos indican que el método no toma argumentos.

Una llamada a un método es conocida como una *invocación*; en este caso, diríamos que estamos invocando `upper` en `palabra`.

Por ejemplo, existe un método de cadena llamado `find` que busca la posición de una cadena dentro de otra:

```
>>> palabra = 'banana'
>>> indice = palabra.find('a')
>>> print(indice)
1
```

En este ejemplo, invocamos `find` en `palabra` y pasamos la letra que estamos buscando como un parámetro.

El método `find` puede encontrar subcadenas así como caracteres:

```
>>> palabra.find('na')
2
```

También puede tomar como un segundo argumento el índice desde donde debe empezar:

```
>>> palabra.find('na', 3)
4
```

Una tarea común es eliminar los espacios en blanco (espacios, tabs, o nuevas líneas) en el inicio y el final de una cadena usando el método `strip`:

```
>>> linea = '  Aquí vamos  '
>>> linea.strip()
'Aquí vamos'
```

Algunos métodos como *startswith* devuelven valores booleanos.

```
>>> linea = 'Que tengas un buen día'
>>> linea.startswith('Que')
True
>>> linea.startswith('q')
False
```

Puedes notar que `startswith` requiere que el formato (mayúsculas y minúsculas) coincida, de modo que a veces tendremos que tomar la línea y cambiarla completamente a minúsculas antes de hacer la verificación, utilizando el método `lower`.

```
>>> linea = 'Que tengas un buen día'
>>> linea.startswith('q')
False
>>> linea.lower()
'que tengas un buen día'
>>> linea.lower().startswith('q')
True
```

En el último ejemplo, el método `lower` es llamado y después usamos `startswith` para ver si la cadena resultante en minúsculas comienza con la letra "q". Siempre y cuando seamos cuidadosos con el orden, podemos hacer múltiples llamadas a métodos en una sola expresión.

Ejercicio 4: Hay un método de cadenas llamado `count` que es similar a la función del ejercicio previo. Lee la documentación de este método en:

https://docs.python.org/library/stdtypes.html#string-methods

Escribe una invocación que cuenta el número de veces que una letra aparece en "banana".

6.10 Analizando cadenas

Frecuentemente, queremos examinar una cadena para encontrar una subcadena. Por ejemplo, si se nos presentaran una seria de líneas con el siguiente formato:

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

y quisiéramos obtener únicamente la segunda parte de la dirección de correo (esto es, `uct.ac.za`) de cada línea, podemos hacer esto utilizando el método `find` y una parte de la cadena.

Primero tenemos que encontrar la posición de la arroba en la cadena. Después, tenemos que encontrar la posición del primer espacio *después* de la arroba. Y después partiremos la cadena para extraer la porción de la cadena que estamos buscando.

```
>>> dato = 'From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008'
>>> arrobapos = dato.find('@')
>>> print(arrobapos)
21
>>> espos = dato.find(' ',arrobapos)
>>> print(espos)
31
>>> direccion = dato[arrobapos+1:espos]
>>> print(direccion)
uct.ac.za
>>>
```

Utilizamos una versión del método `find` que nos permite especificar la posición en la cadena desde donde queremos que `find` comience a buscar. Cuando recortamos una parte de una cadena, extraemos los caracteres desde "uno después de la arroba hasta, *pero no incluyendo*, el carácter de espacio".

La documentación del método `find` está disponible en

https://docs.python.org/library/stdtypes.html#string-methods.

6.11 El operador de formato

El *operador de formato* % nos permite construir cadenas, reemplazando partes de las cadenas con datos almacenados en variables. Cuando lo aplicamos a enteros, % es el operador módulo. Pero cuando es aplicado a una cadena, % es el operador de formato.

El primer operando es la *cadena a formatear*, la cual contiene una o más *secuencias de formato* que especifican cómo el segundo operando es formateado. El resultado es una cadena.

Por ejemplo, la secuencia de formato %d significa que el segundo operando debería ser formateado como un entero ("d" significa "decimal"):

```
>>> camellos = 42
>>> '%d' % camellos
'42'
```

El resultado es la cadena '42', el cual no debe ser confundido con el valor entero 42.

Una secuencia de formato puede aparecer en cualquier lugar en la cadena, así que puedes meter un valor en una frase:

```
>>> camellos = 42
>>> 'Yo he visto %d camellos.' % camellos
'Yo he visto 42 camellos.'
```

Si hay más de una secuencia de formato en la cadena, el segundo argumento tiene que ser una tupla[1]. Cada secuencia de formato es relacionada con un elemento de la tupla, en orden.

El siguiente ejemplo usa %d para formatear un entero, %g para formatear un número de punto flotante (no preguntes por qué), y %s para formatear una cadena:

```
>>> 'En %d años yo he visto %g %s.' % (3, 0.1, 'camellos')
'En 3 años yo he visto 0.1 camellos.'
```

El número de elementos en la tupla debe coincidir con el número de secuencias de formato en la cadena. El tipo de los elementos también debe coincidir con la secuencia de formato:

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dolares'
TypeError: %d format: a number is required, not str
```

[1] Una tupla es una secuencia de valores separados por comas dentro de un par de paréntesis. Veremos tuplas en el Capítulo 10

En el primer ejemplo, no hay suficientes elementos; en el segundo, el elemento es de un tipo incorrecto.

El operador de formato es poderoso, pero puede ser difícil de usar. Puedes leer más al respecto en

https://docs.python.org/library/stdtypes.html#printf-style-string-formatting.

6.12 Depuración

Una habilidad que debes desarrollar cuando programas es siempre preguntarte a ti mismo, "¿Qué podría fallar aquí?" o alternativamente, "¿Qué cosa ilógica podría hacer un usuario para hacer fallar nuestro (aparentemente) perfecto programa?"

Por ejemplo, observa el programa que utilizamos para demostrar el bucle `while` en el capítulo de iteraciones:

```
while True:
    linea = input('> ')
    if linea[0] == '#' :
        continue
    if linea == 'fin':
        break
    print(linea)
print('¡Terminado!')

# Code: http://www.py4e.com/code3/copytildone2.py
```

Mira lo que pasa cuando el usuario introduce una línea vacía como entrada:

```
> hello there
hello there
> # don't print this
> print this!
print this!
>
Traceback (most recent call last):
  File "copytildone.py", line 3, in <module>
    if line[0] == '#':
IndexError: string index out of range
```

El código funciona bien hasta que se presenta una línea vacía. En ese momento no hay un carácter cero, por lo que obtenemos una traza de error (traceback). Existen dos soluciones a esto para convertir la línea tres en "segura", incluso si la línea está vacía.

Una posibilidad es simplemente usar el método `startswith` que devuelve `False` si la cadena está vacía.

```
if line.startswith('#'):
```

Otra forma segura es escribir una sentencia `if` utilizando el patrón *guardián* y asegurarse que la segunda expresión lógica es evaluada sólo cuando hay al menos un carácter en la cadena:

```
if len(line) > 0 and line[0] == '#':
```

6.13 Glosario

contador Una variable utilizada para contar algo, usualmente inicializada a cero y luego incrementada.

cadena vacía una cadena sin caracteres y de tamaño 0, representada por dos comillas sencillas.

operador de formato Un operador, `%`, que toma una cadena de formato y una tupla y genera una cadena que incluye los elementos de la tupla formateados como se especifica en la cadena de formato.

secuencia de formato Una secuencia de caracteres en una cadena a formatear, como `%d`, que especifica cómo un valor debe ser formateado.

cadena a formatear Una cadena, usado con el operador de formato, que contiene secuencias de formato.

flag (bandera) Una variable booleana utilizada para indicar si una condición es verdadera o falsa.

invocación Una sentencia que llama un método.

inmutable La propiedad de una secuencia cuyos elementos no pueden ser asignados.

índice Un valor entero utilizado para seleccionar un ítem en una secuencia, tal como un carácter en una cadena.

ítem Uno de los valores en una secuencia.

método Una función que está asociada a un objeto y es llamada utilizando la notación de punto.

objeto Algo a lo que una variable puede referirse. Por ahora, puedes usar "objeto" y "valor" indistintamente.

búsqueda Un patrón de recorrido que se detiene cuando encuentra lo que está buscando.

secuencia Un conjunto ordenado; esto es, un conjunto de valores donde cada valor es identificado por un índice entero.

parte (slice) Una parte de una cadena especificado por un rango de índices.

atravesar Iterar a través de los ítems de una secuencia, ejecutando una operación similar en cada uno.

6.14 Ejercicios

Ejercicio 5: Toma el siguiente código en Python que almacena una cadena:

```
str = 'X-DSPAM-Confidence:0.8475'
```

Utiliza `find` y una parte de la cadena para extraer la porción de la cadena después del carácter dos puntos y después utiliza la función `float` para convertir la cadena extraída en un número de punto flotante.

Ejercicio 6: Lee la documentación de los métodos de cadenas en https://docs.python.org/library/stdtypes.html#string-methods Quizá quieras experimentar con algunos de ellos para asegurarte de entender como funcionan. `strip` y `replace` son particularmente útiles.

La documentación usa una sintaxis que puede ser confusa. Por ejemplo, en `find(sub[, start[, end]])`, los corchetes indican argumentos opcionales. De modo que `sub` es requerido, pero `start` es opcional, y si se incluye `start`, entonces `end` es opcional.

Chapter 7

Archivos

7.1 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.

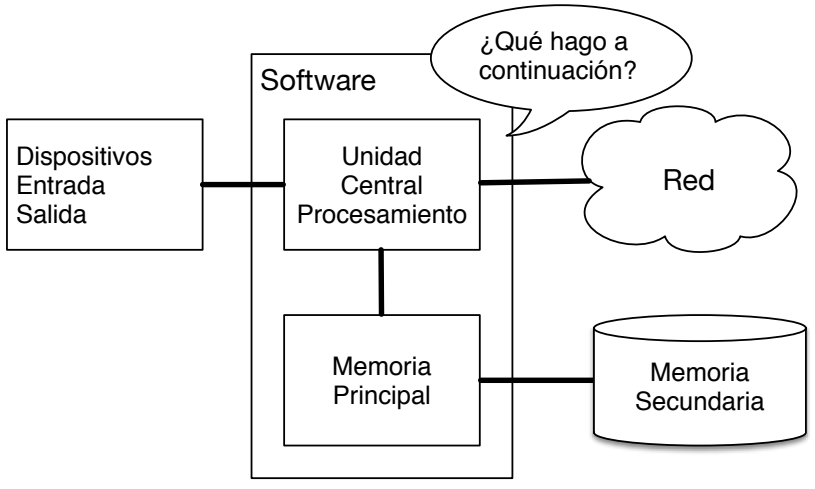

Figure 7.1: Memoria Secundaria

En este capítulo, vamos a comenzar a trabajar con *Memoria Secundaria* (o archivos). 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 archivos como los que creamos en un editor de texto. Más adelante veremos cómo trabajar con archivos de bases de datos, que son archivos binarios diseñados específicamente para ser leídos y escritos a través de software para manejo de bases de datos.

7.2 Abrir archivos

Cuando queremos abrir o escribir un archivo (digamos, en el disco duro), primero debemos *abrir* el archivo. Al abrir el archivo nos comunicamos con el sistema operativo, el cual sabe dónde están almacenados los datos de cada archivo. Cuando abres un archivo, le estás pidiendo al sistema operativo que encuentre el archivo por su nombre y se asegure de que existe. En este ejemplo, abrimos el archivo *mbox.txt*, el cual debería estar almacenado en el mismo directorio en que estás localizado cuando inicias Python. Puedes descargar este archivo desde www.py4e.com/code3/mbox.txt

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

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

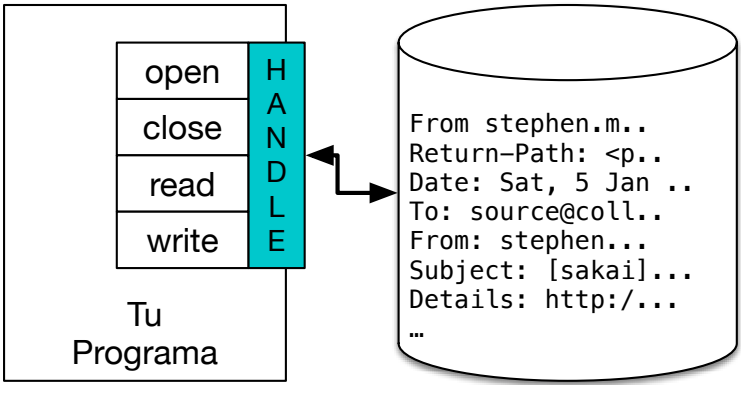

Figure 7.2: Un Manejador de Archivo

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

```
>>> manejador_archivo = 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 archivo que no existe.

7.3 Archivos de texto y líneas

Un archivo 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 archivo 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 archivo completo de interacciones por correo está disponible en

www.py4e.com/code3/mbox.txt

y una versión reducida del archivo está disponible en

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

Esos archivos están en un formato estándar para un archivo 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 archivo 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 archivo, 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 archivo en líneas.

7.4 Lectura de archivos

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

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
    count = count + 1
print('Line Count:', count)

# Code: http://www.py4e.com/code3/open.py
```

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

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

Cuando el archivo es leído usando un bucle `for` de esta manera, Python se encarga de dividir los datos del archivo 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 `line` 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 archivos muy grandes sin quedarse sin memoria principal para almacenar los datos. El programa previo puede contar las líneas de cualquier

tamaño de archivo utilizando poca memoria, puesto que cada línea es leída, contada, y después descartada.

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

```
>>> manejador_archivo = open('mbox-short.txt')
>>> inp = manejador_archivo.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* 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 archivo 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 archivo son apropiados para la memoria principal del sistema. Si el archivo es muy grande para caber en la memoria principal, deberías escribir tu programa para leer el archivo en bloques utilizando un bucle `for` o `while`.

7.5 Búsqueda a través de un archivo

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

Por ejemplo, si queremos leer un archivo 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:

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
```

```
    if line.startswith('From:'):
        print(line)

# Code: http://www.py4e.com/code3/search1.py
```

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 *line*, 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:

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

# Code: http://www.py4e.com/code3/search2.py
```

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 archivos 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:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    # Skip 'uninteresting lines'
    if not line.startswith('From:'):
        continue
    # Process our 'interesting' line
    print(line)

# Code: http://www.py4e.com/code3/search3.py
```

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):

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

# Code: http://www.py4e.com/code3/search4.py
```

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.

7.6 Permitiendo al usuario elegir el nombre de archivo

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

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

```
fname = input('Enter the file name: ')
fhand = open(fname)
count = 0
for line in fhand:
    if line.startswith('Subject:'):
        count = count + 1
print('There were', count, 'subject lines in', fname)

# Code: http://www.py4e.com/code3/search6.py
```

Leemos el nombre de archivo del usuario y lo guardamos en una variable llamada `fname` y abrimos el 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 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?"

7.7 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 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 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:

```
fname = input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print('File cannot be opened:', fname)
    exit()
count = 0
for line in fhand:
    if line.startswith('Subject:'):
        count = count + 1
print('There were', count, 'subject lines in', fname)

# Code: http://www.py4e.com/code3/search7.py
```

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 archivo incorrecto, vamos a "capturarlo" y recuperarnos de forma elegante:

```
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 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 archivo.

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.

7.8 Escritura de archivos

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

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

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

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

```
>>> linea1 = "This here's the wattle,\n"
>>> fout.write(linea1)
24
```

El manejador de archivo 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 archivo, 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 = 'the emblem of our land.\n'
>>> fout.write(linea2)
24
```

Cuando terminas de escribir, tienes que cerrar el archivo 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.

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

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

7.9 Depuración

Cuando estás leyendo y escribiendo archivos, 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 `\n`. Otros usan un carácter de retorno, representado con `\r`. Otros usan ambos. Si mueves archivos 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.

7.10 Glosario

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

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

Pythónico Una técnica que funciona de forma elegante en Python. “Utilizar try y except es la forma *Pythónica* de gestionar los archivos inexistentes”.

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.

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

7.11 Ejercicios

Ejercicio 1: Escribe un programa que lea un archivo e imprima su contenido (línea por línea), todo en mayúsculas. Al ejecutar el programa, debería parecerse a esto:

```
python shout.py
Ingresa un nombre de archivo: 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 que solicite un nombre de archivo y después lea ese archivo 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 archivo, imprime el valor medio de "spam confidence".

```
Ingresa un nombre de archivo: mbox.txt
Promedio spam confidence: 0.894128046745

Ingresa un nombre de archivo: mbox-short.txt
Promedio spam confidence: 0.750718518519
```

Prueba tu programa con los archivos *mbox.txt* y *mbox-short.txt*.

Ejercicio 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 archivo para que imprima un mensaje divertido cuando el usuario escriba "na na boo boo" como nombre de archivo. El programa debería funcionar normalmente para cualquier archivo que exista o no exista. Aquí está un ejemplo de la ejecución del programa:

```
python huevo.py
Ingresa un nombre de archivo: mbox.txt
Hay 1797 líneas subject en mbox.txt

python huevo.py
Ingresa un nombre de archivo: inexistente.tyxt
El archivo no puede ser abierto: inexistente.tyxt

python huevo.py
Ingresa un nombre de archivo: 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.

Chapter 8

Listas

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
count = 0
while (True):
    inp = input('Enter a number: ')
    if inp == 'done': break
    value = float(inp)
    total = total + value
    count = count + 1

average = total / count
print('Average:', average)

# Code: http://www.py4e.com/code3/avenum.py
```

En este programa, tenemos las variables `count` 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.

```
numlist = list()
while (True):
    inp = input('Enter a number: ')
    if inp == 'done': break
```

```
    value = float(inp)
    numlist.append(value)

average = sum(numlist) / len(numlist)
print('Average:', average)

# Code: http://www.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:

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

# Code: http://www.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 Glosariso

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.
índice Un valor entero que indica un elemento en una lista.
idéntico Ser el mismo objeto (lo cual implica equivalencia).
lista Una secuencia de valores.
recorrido de lista Acceso secuencial a cada elemento de una lista.
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.
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
```

Chapter 9

Diccionarios

Un *diccionario* es como una lista, pero más general. En una lista, los índices de posiciones tienen que ser enteros; en un diccionario, los índices pueden ser (casi) cualquier tipo.

Puedes pensar en un diccionario como una asociación entre un conjunto de índices (que son llamados *claves*) y un conjunto de valores. Cada clave apunta a un valor. La asociación de una clave y un valor es llamada *par clave-valor* o a veces *elemento*.

Como ejemplo, vamos a construir un diccionario que asocia palabras de Inglés a Español, así que todas las claves y los valores son cadenas.

La función `dict` crea un nuevo diccionario sin elementos. Debido a que `dict` es el nombre de una función interna, deberías evitar usarlo como un nombre de variable.

```
>>> eng2sp = dict()
>>> print(eng2sp)
{}
```

Las llaves, `{}`, representan un diccionario vacío. Para agregar elementos a un diccionario, puedes utilizar corchetes:

```
>>> eng2sp['one'] = 'uno'
```

Esta línea crea un elemento asociando a la clave `'one'` el valor "uno". Si imprimimos el diccionario de nuevo, vamos a ver un par clave-valor con dos puntos entre la clave y el valor:

```
>>> print(eng2sp)
{'one': 'uno'}
```

Este formato de salida es también un formato de entrada. Por ejemplo, puedes crear un nuevo diccionario con tres elementos. Pero si imprimes `eng2sp`, te vas a sorprender:

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
>>> print(eng2sp)
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

El orden de los pares clave-elemento no es el mismo. De hecho, si tu escribes este mismo ejemplo en tu computadora, podrías obtener un resultado diferente. En general, el orden de los elementos en un diccionario es impredecible.

Pero ese no es un problema porque los elementos de un diccionario nunca son indexados con índices enteros. En vez de eso, utilizas las claves para encontrar los valores correspondientes:

```
>>> print(eng2sp['two'])
'dos'
```

La clave `'two'` siempre se asocia al valor "dos", así que el orden de los elementos no importa.

Si la clave no está en el diccionario, obtendrás una excepción (exception):

```
>>> print(eng2sp['four'])
KeyError: 'four'
```

La función `len` funciona en diccionarios; ésta regresa el número de pares clave-valor:

```
>>> len(eng2sp)
3
```

El operador `in` funciona en diccionarios; éste te dice si algo aparece como una *clave* en el diccionario (aparecer como valor no es suficiente).

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

Para ver si algo aparece como valor en un diccionario, puedes usar el método `values`, el cual retorna los valores como una lista, y después puedes usar el operador `in`:

```
>>> vals = list(eng2sp.values())
>>> 'uno' in vals
True
```

El operador `in` utiliza diferentes algoritmos para listas y diccionarios. Para listas, utiliza un algoritmo de búsqueda lineal. Conforme la lista se vuelve más grande, el tiempo de búsqueda se vuelve más largo en proporción al tamaño de la lista. Para diccionarios, Python utiliza un algoritmo llamado *tabla hash* (hash table, en inglés)

que tiene una propiedad importante: el operador `in` toma la misma cantidad de tiempo sin importar cuántos elementos haya en el diccionario. No voy a explicar porqué las funciones hash son tan mágicas, pero puedes leer más al respecto en es.wikipedia.org/wiki/Tabla_hash.

Ejercicio 1: Descargar una copia del archivo www.py4e.com/code3/words.txt

Escribe un programa que lee las palabras en *words.txt* y las almacena como claves en un diccionario. No importa qué valores tenga. Luego puedes utilizar el operador `in` como una forma rápida de revisar si una cadena está en el diccionario.

9.1 Diccionario como un conjunto de contadores

Supongamos que recibes una cadena y quieres contar cuántas veces aparece cada letra. Hay varias formas en que puedes hacerlo:

1. Puedes crear 26 variables, una por cada letra del alfabeto. Luego puedes recorrer la cadena, y para cada caracter, incrementar el contador correspondiente, probablemente utilizando varios condicionales.

2. Puedes crear una lista con 26 elementos. Después podrías convertir cada caracter en un número (usando la función interna `ord`), usar el número como índice dentro de la lista, e incrementar el contador correspondiente.

3. Puedes crear un diccionario con caracteres como claves y contadores como los valores correspondientes. La primera vez que encuentres un caracter, agregarías un elemento al diccionario. Después de eso incrementarías el valor del elemento existente.

Cada una de esas opciones hace la misma operación computacional, pero cada una de ellas implementa esa operación en forma diferente.

Una *implementación* es una forma de llevar a cabo una operación computacional; algunas implementaciones son mejores que otras. Por ejemplo, una ventaja de la implementación del diccionario es que no tenemos que saber con antelación qué letras aparecen en la cadena y solamente necesitamos espacio para las letras que sí aparecen.

Aquí está un ejemplo de como se vería ese código:

```
palabra = 'brontosaurio'
d = dict()
for c in palabra:
    if c not in d:
        d[c] = 1
    else:
        d[c] = d[c] + 1
print(d)
```

Realmente estamos calculando un *histograma*, el cual es un término estadístico para un conjunto de contadores (o frecuencias).

El bucle `for` recorre la cadena. Cada vez que entramos al bucle, si el caracter `c` no está en el diccionario, creamos un nuevo elemento con la clave `c` y el valor inicial 1 (debido a que hemos visto esta letra solo una vez). Si `c` ya está previamente en el diccionario incrementamos `d[c]`.

Aquí está la salida del programa:

```
{'b': 1, 'r': 2, 'o': 3, 'n': 1, 't': 1, 's': 1, 'a': 1, 'u': 1, 'i': 1}
```

El histograma indica que las letras “a” y “b” aparecen solo una vez; “o” aparece dos, y así sucesivamente.

Los diccionarios tienen un método llamado `get` que toma una clave y un valor por defecto. Si la clave aparece en el diccionario, `get` regresa el valor correspondiente; si no, regresa el valor por defecto. Por ejemplo:

```
>>> cuentas = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
>>> print(cuentas.get('jan', 0))
100
>>> print(cuentas.get('tim', 0))
0
```

Podemos usar `get` para escribir nuestro bucle de histograma más conciso. Puesto que el método `get` automáticamente maneja el caso en que una clave no está en el diccionario, podemos reducir cuatro líneas a una y eliminar la sentencia `if`.

```
palabra = 'brontosaurio'
d = dict()
for c in palabra:
    d[c] = d.get(c,0) + 1
print(d)
```

El uso del método `get` para simplificar este bucle contador termina siendo un “idioma” muy utilizado en Python y vamos a utilizarlo muchas veces en el resto del libro. Así que deberías tomar un momento para comparar el bucle utilizando la sentencia `if` y el operador `in` con el bucle utilizando el método `get`. Ambos hacen exactamente lo mismo, pero uno es más breve.

9.2 Diccionarios y archivos

Uno de los usos más comunes de un diccionario es contar las ocurrencias de palabras en un archivo con algún texto escrito. Vamos comenzando con un archivo de palabras muy simple tomado del texto de *Romeo y Julieta*.

Para el primer conjunto de ejemplos, vamos a usar una versión más corta y más simplificada del texto sin signos de puntuación. Después trabajaremos con el texto de la escena con signos de puntuación incluidos.

```
But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

Vamos a escribir un programa de Python para leer a través de las líneas del archivo, dividiendo cada línea en una lista de palabras, y después iterando a través de cada una de las palabras en la línea y contando cada palabra utilizando un diccionario.

Verás que tenemos dos bucles `for`. El bucle externo está leyendo las líneas del archivo y el bucle interno está iterando a través de cada una de las palabras en esa línea en particular. Este es un ejemplo de un patrón llamado *bucles anidados* porque uno de los bucles es el bucle *externo* y el otro bucle es el bucle *interno*.

Como el bucle interno ejecuta todas sus iteraciones cada vez que el bucle externo hace una sola iteración, consideramos que el bucle interno itera "más rápido" y el bucle externo itera más lento.

La combinación de los dos bucles anidados asegura que contemos cada palabra en cada línea del archivo de entrada.

```
fname = input('Ingresa el nombre de archivo: ')
try:
    fhand = open(fname)
except:
    print('El archivo no se puede abrir:', fname)
    exit()

counts = dict()
for line in fhand:
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

print(counts)

# Code: http://www.py4e.com/code3/count1.py
```

En nuestra sentencia `else`, utilizamos la alternativa más compacta para incrementar una variable. `counts[word] += 1` es equivalente a `counts[word] = counts[word] + 1`. Cualquiera de los dos métodos puede usarse para cambiar el valor de una variable en cualquier cantidad. Existen alternativas similares para `-=`, `*=`, y `/=`.

Cuando ejecutamos el programa, vemos una salida sin procesar que contiene todos los contadores sin ordenar. (el archivo *romeo.txt* está disponible en es.py4e.com/code3/romeo.txt)

```
python count1.py
```

```
Ingresa el nombre de archivo: romeo.txt
{'and': 3, 'envious': 1, 'already': 1, 'fair': 1,
'is': 3, 'through': 1, 'pale': 1, 'yonder': 1,
'what': 1, 'sun': 2, 'Who': 1, 'But': 1, 'moon': 1,
'window': 1, 'sick': 1, 'east': 1, 'breaks': 1,
'grief': 1, 'with': 1, 'light': 1, 'It': 1, 'Arise': 1,
'kill': 1, 'the': 3, 'soft': 1, 'Juliet': 1}
```

Es un poco inconveniente ver a través del diccionario para encontrar las palabras más comunes y sus contadores, así que necesitamos agregar un poco más de código para mostrar una salida que nos sirva más.

9.3 Bucles y diccionarios

Si utilizas un diccionario como una secuencia para una sentencia `for`, esta recorre las claves del diccionario. Este bucle imprime cada clave y su valor correspondiente:

```
contadores = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
for clave in contadores:
    print(clave, contadores[clave])
```

Aquí está lo que muestra de salida:

```
jan 100
chuck 1
annie 42
```

De nuevo, las claves no están en ningún orden en particular.

Podemos utilizar este patrón para implementar varios idiomas de bucles que hemos descrito previamente. Por ejemplo, si queremos encontrar todas las entradas en un diccionario con valor mayor a diez, podemos escribir el siguiente código:

```
contadores = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
for clave in contadores:
    if contadores[clave] > 10 :
        print(clave, contadores[clave])
```

El bucle `for` itera a través de las *claves* del diccionario, así que debemos utilizar el operador índice para obtener el *valor* correspondiente para cada clave. Aquí está la salida del programa:

```
jan 100
annie 42
```

Vemos solamente las entradas que tienen un valor mayor a 10.

Si quieres imprimir las claves en orden alfabético, primero haces una lista de las claves en el diccionario utilizando el método `keys` disponible en los objetos de diccionario, y después ordenar esa lista e iterar a través de la lista ordenada, buscando cada clave e imprimiendo pares clave-valor ordenados, tal como se muestra a continuación:

```
contadores = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
lst = list(contadores.keys())
print(lst)
lst.sort()
for clave in lst:
    print(clave, contadores[clave])
```

Así se muestra la salida:

```
['jan', 'chuck', 'annie']
annie 42
chuck 1
jan 100
```

Primero se ve la lista de claves sin ordenar como la obtuvimos del método `keys`. Después vemos los pares clave-valor en orden desde el bucle `for`.

9.4 Análisis avanzado de texto

En el ejemplo anterior utilizando el archivo *romeo.txt*, hicimos el archivo tan simple como fue posible removiendo los signos de puntuación a mano. El text real tiene muchos signos de puntuación, como se muestra abajo.

```
But, soft! what light through yonder window breaks?
It is the east, and Juliet is the sun.
Arise, fair sun, and kill the envious moon,
Who is already sick and pale with grief,
```

Puesto que la función `split` en Python busca espacios y trata las palabras como piezas separadas por esos espacios, trataríamos a las palabras "soft!" y "soft" como *diferentes* palabras y crearíamos una entrada independiente para cada palabra en el diccionario.

Además, como el archivo tiene letras mayúsculas, trataríamos "who" y "Who" como diferentes palabras con diferentes contadores.

Podemos resolver ambos problemas utilizando los métodos de cadenas `lower`, `punctuation`, y `translate`. El método `translate` es el más sutil de los métodos. Aquí esta la documentación para `translate`:

```
line.translate(str.maketrans(fromstr, tostr, deletestr))
```

Reemplaza los caracteres en `fromstr` *con el caracter en la misma posición en* `tostr` *y elimina todos los caracteres que están en* `deletestr`*. Los parámetros* `fromstr` *y* `tostr` *pueden ser cadenas vacías y el parámetro* `deletestr` *es opcional.*

No vamos a especificar el valor de `tostr` pero vamos a utilizar el parámetro `deletestr` para eliminar todos los signos de puntuación. Incluso vamos a dejar que Python nos diga la lista de caracteres que considera como "signos de puntuación":

```
>>> import string
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

Los parámetros utilizados por `translate` eran diferentes en Python 2.0.

Hacemos las siguientes modificaciones a nuestro programa:

```
import string

fname = input('Ingresa el nombre de archivo: ')
try:
    fhand = open(fname)
except:
    print('El archivo no se puede abrir:', fname)
    exit()

counts = dict()
for line in fhand:
    line = line.rstrip()
    line = line.translate(line.maketrans('', '', string.punctuation))
    line = line.lower()
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

print(counts)

# Code: http://www.py4e.com/code3/count2.py
```

Parte de aprender el "Arte de Python" o "Pensamiento Pythónico" es entender que Python muchas veces tiene funciones internas para muchos problemas de análisis de datos comunes. A través del tiempo, verás suficientes códigos de ejemplo y leerás lo suficiente en la documentación para saber dónde buscar si alguien escribió algo que haga tu trabajo más fácil.

Lo siguiente es una versión reducida de la salida:

```
Ingresa el nombre de archivo: romeo-full.txt
{'swearst': 1, 'all': 6, 'afeard': 1, 'leave': 2, 'these': 2,
'kinsmen': 2, 'what': 11, 'thinkst': 1, 'love': 24, 'cloak': 1,
a': 24, 'orchard': 2, 'light': 5, 'lovers': 2, 'romeo': 40,
'maiden': 1, 'whiteupturned': 1, 'juliet': 32, 'gentleman': 1,
'it': 22, 'leans': 1, 'canst': 1, 'having': 1, ...}
```

Interpretar los datos a través de esta salida es aún difícil, y podemos utilizar Python para darnos exactamente lo que estamos buscando, pero para que sea así, necesitamos aprender acerca de las *tuplas* en Python. Vamos a retomar este ejemplo una vez que aprendamos sobre tuplas.

9.5 Depuración

Conforme trabajes con conjuntos de datos más grandes puede ser complicado depurar imprimiendo y revisando los datos a mano. Aquí hay algunas sugerencias para depurar grandes conjuntos de datos:

Reducir la entrada Si es posible, trata de reducir el tamaño del conjunto de datos. Por ejemplo, si el programa lee un archivo de texto, comienza solamente con las primeras 10 líneas, o con el ejemplo más pequeño que puedas encontrar. Puedes ya sea editar los archivos directamente, o (mejor) modificar el programa para que solamente lea las primeras `n` número de líneas.

Si hay un error, puedes reducir `n` al valor más pequeño que produce el error, y después incrementarlo gradualmente conforme vayas encontrando y corrigiendo errores.

Revisar extractos y tipos En lugar de imprimir y revisar el conjunto de datos completo, considera imprimir extractos de los datos: por ejemplo, el número de elementos en un diccionario o el total de una lista de números.

Una causa común de errores en tiempo de ejecución es un valor que no es el tipo correcto. Para depurar este tipo de error, generalmente es suficiente con imprimir el tipo de un valor.

Escribe auto-verificaciones Algunas veces puedes escribir código para revisar errores automáticamente. Por ejemplo, si estás calculando el promedio de una lista de números, podrías verificar que el resultado no sea más grande que el elemento más grande de la lista o que sea menor que el elemento más pequeño de la lista. Esto es llamado "prueba de sanidad" porque detecta resultados que son "completamente ilógicos".

Otro tipo de prueba compara los resultados de dos diferentes cálculos para ver si son consistentes. Esto es conocido como "prueba de consistencia".

Imprimir una salida ordenada Dar un formato a los mensajes de depuración puede facilitar encontrar un error.

De nuevo, el tiempo que inviertas haciendo una buena estructura puede reducir el tiempo que inviertas en depurar.

9.6 Glosario

diccionario Una asociación de un conjunto de claves a sus valores correspondientes.

tabla hash El algoritmo utilizado para implementar diccionarios en Python.

función hash Una función utilizada por una tabla hash para calcular la localización de una clave.

histograma Un set de contadores.

implementación Una forma de llevar a cabo un cálculo.

elemento Otro nombre para un par clave-valor.

clave Un objeto que aparece en un diccionario como la primera parte de un par clave-valor.

par clave-valor La representación de una asociación de una clave a un valor.

búsqueda Una operación de diccionario que toma una clave y encuentra su valor correspondiente.

bucles anidados Cuando hay uno o más bucles "dentro" de otro bucle. Los bucles internos terminan de ejecutar cada vez que el bucle externo ejecuta una vez.

valor Un objeto que aparece en un diccionario como la segunda parte de un par clave-valor. Esta definición es más específica que nuestro uso previo de la palabra "valor".

9.7 Ejercicios

Ejercicio 2: Escribir un programa que clasifica cada mensaje de correo dependiendo del día de la semana en que se recibió. Para hacer esto busca las líneas que comienzan con "From", después busca por la tercer palabra y mantén un contador para cada uno de los días de la semana. Al final del programa imprime los contenidos de tu diccionario (el orden no importa).

Línea de ejemplo:

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

Ejemplo de ejecución:

```
python dow.py
Ingresa un nombre de archivo: mbox-short.txt
{'Fri': 20, 'Thu': 6, 'Sat': 1}
```

Ejercicio 3: Escribe un programa para leer a través de un historial de correos, construye un histograma utilizando un diccionario para contar cuántos mensajes han llegado de cada dirección de correo electrónico, e imprime el diccionario.

```
Ingresa un nombre de archivo: mbox-short.txt
{'gopal.ramasammycook@gmail.com': 1, 'louis@media.berkeley.edu': 3,
'cwen@iupui.edu': 5, 'antranig@caret.cam.ac.uk': 1,
'rjlowe@iupui.edu': 2, 'gsilver@umich.edu': 3,
'david.horwitz@uct.ac.za': 4, 'wagnermr@iupui.edu': 1,
'zqian@umich.edu': 4, 'stephen.marquard@uct.ac.za': 2,
'ray@media.berkeley.edu': 1}
```

Ejercicio 4: Agrega código al programa anterior para determinar quién tiene la mayoría de mensajes en el archivo. Después de que todos los datos hayan sido leídos y el diccionario haya sido creado, mira a través del diccionario utilizando un bucle máximo (ve Capítulo 5: Bucles máximos y mínimos) para encontrar quién tiene la mayoría de mensajes e imprimir cuántos mensajes tiene esa persona.

```
Ingresa un nombre de archivo: mbox-short.txt
cwen@iupui.edu 5

Ingresa un nombre de archivo: mbox.txt
zqian@umich.edu 195
```

Ejercicio 5: Este programa almacena el nombre del dominio (en vez de la dirección) desde donde fue enviado el mensaje en vez de quién envió el mensaje (es decir, la dirección de correo electrónica completa). Al final del programa, imprime el contenido de tu diccionario.

```
python schoolcount.py
Ingresa un nombre de archivo: mbox-short.txt
{'media.berkeley.edu': 4, 'uct.ac.za': 6, 'umich.edu': 7,
'gmail.com': 1, 'caret.cam.ac.uk': 1, 'iupui.edu': 8}
```

Chapter 10

Tuplas

10.1 Las Tuplas son inmutables

Una tupla[1] es una secuencia de valores similar a una lista. Los valores guardados en una tupla pueden ser de cualquier tipo, y son indexados por números enteros. La principal diferencia es que las tuplas son *inmutables*. Las tuplas además son *comparables* y *dispersables* (hashables) de modo que las listas de tuplas se pueden ordenar y también usar tuplas como valores para las claves en diccionarios de Python.

Sintácticamente, una tupla es una lista de valores separados por comas:

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

Aunque no es necesario, es común encerrar las tuplas entre paréntesis para ayudarnos a identificarlas rápidamente cuando revisemos código de Python:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

Para crear una tupla con un solo elemento, es necesario incluir una coma al final:

```
>>> t1 = ('a',)
>>> type(t1)
<type 'tuple'>
```

Sin la coma, Python considera `('a')` como una expresión con una cadena entre paréntesis que es evaluada como de tipo cadena (string):

```
>>> t2 = ('a')
>>> type(t2)
<type 'str'>
```

[1] Dato curioso: La palabra "tuple" proviene de los nombres dados a secuencias de números de distintas longitudes: simple, doble, triple, cuádruple, quíntuple, séxtuple, séptuple, etc.

Otra forma de construir una tupla es utilizando la función interna `tuple`. Sin argumentos, ésta crea una tupla vacía:

```
>>> t = tuple()
>>> print(t)
()
```

Si el argumento es una secuencia (cadena, lista, o tupla), el resultado de la llamada a `tuple` es una tupla con los elementos de la secuencia:

```
>>> t = tuple('altramuces')
>>> print(t)
('a', 'l', 't', 'r', 'a', 'm', 'u', 'c', 'e', 's')
```

Dado que `tuple` es el nombre de un constructor, debería evitarse su uso como nombre de variable.

La mayoría de los operadores de listas también funcionan en tuplas. El operador corchete indexa un elemento:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print(t[0])
'a'
```

Y el operador de rebanado (slice) selecciona un rango de elementos.

```
>>> print(t[1:3])
('b', 'c')
```

Pero si se intenta modificar uno de los elementos de la tupla, se produce un error:

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

No se puede modificar los elementos de una tupla, pero sí se puede reemplazar una tupla por otra:

```
>>> t = ('A',) + t[1:]
>>> print(t)
('A', 'b', 'c', 'd', 'e')
```

10.2 Comparación de tuplas

Los operadores de comparación funcionan con tuplas y otras secuencias. Python comienza comparando el primer elemento de cada secuencia. Si ambos elementos son iguales, pasa al siguiente elemento y así sucesivamente, hasta que encuentra elementos diferentes. Los elementos subsecuentes no son considerados (aunque sean muy grandes).

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

La función `sort` funciona de la misma manera. Ordena inicialmente por el primer elemento, pero en el caso de que ambos elementos sean iguales, ordena por el segundo elemento, y así sucesivamente.

Esta característica se presta a un patrón de diseño llamado *DSU*, que

Decorate (Decora) una secuencia, construyendo una lista de tuplas con uno o más índices ordenados precediendo los elementos de la secuencia,
Sort (Ordena) la lista de tuplas utilizando la función interna `sort`, y
Undecorate (Quita la decoración) extrayendo los elementos ordenados de la secuencia.

Por ejemplo, suponiendo una lista de palabras que se quieren ordenar de la más larga a la más corta:

```
txt = 'Pero qué luz se deja ver allí'
palabras = txt.split()
t = list()
for palabra in palabras:
    t.append((len(palabra), palabra))

t.sort(reverse=True)

res = list()
for longitud, palabra in t:
    res.append(palabra)

print(res)

# Code: http://www.py4e.com/code3/soft.py
```

El primer bucle genera una lista de tuplas, donde cada tupla es una palabra precedida por su longitud.

`sort` compara el primer elemento (longitud) primero, y solamente considera el segundo elemento para desempatar. El argumento clave `reverse=True` indica a `sort` que debe ir en orden decreciente.

El segundo bucle recorre la lista de tuplas y construye una lista de palabras en orden descendente según la longitud. Las palabras de cuatro letras están ordenadas en orden alfabético *inverso*, así que "deja" aparece antes que "allí" en la siguiente lista.

La salida del programa es la siguiente:

```
['deja', 'allí', 'Pero', 'ver', 'qué', 'luz', 'se']
```

Por supuesto, la línea pierde mucho de su impacto poético cuando se convierte en una lista de Python y se almacena en orden descendente según la longitud de las palabras.

10.3 Asignación de tuplas

Una de las características sintácticas únicas del lenguaje Python es la capacidad de tener una tupla en el lado izquierdo de una sentencia de asignación. Esto permite asignar más de una variable a la vez cuando hay una secuencia del lado izquierdo.

En este ejemplo tenemos una lista de dos elementos (la cual es una secuencia) y asignamos el primer y segundo elementos de la secuencia a las variables `x` y `y` en una única sentencia.

```
>>> m = [ 'pásalo', 'bien' ]
>>> x, y = m
>>> x
'pásalo'
>>> y
'bien'
>>>
```

No es magia, Python traduce *aproximadamente* la sintaxis de asignación de la tupla de este modo::[2]

```
>>> m = [ 'pásalo', 'bien' ]
>>> x = m[0]
>>> y = m[1]
>>> x
'pásalo'
>>> y
'bien'
>>>
```

Estilísticamente, cuando se utiliza una tupla en el lado izquierdo de la asignación, se omiten los paréntesis, pero lo que se muestra a continuación es una sintaxis igualmente válida:

```
>>> m = [ 'pásalo', 'bien' ]
>>> (x, y) = m
>>> x
'pásalo'
>>> y
'bien'
>>>
```

[2] Python no traduce la sintaxis literalmente. Por ejemplo, si se trata de hacer esto con un diccionario, no va a funcionar como se podría esperar.

Una aplicación particularmente ingeniosa de asignación con tuplas permite *intercambiar* los valores de dos variables en una sola sentencia:

```
>>> a, b = b, a
```

Ambos lados de la sentencia son tuplas, pero el lado izquierdo es una tupla de variables; el lado derecho es una tupla de expresiones. Cada valor en el lado derecho es asignado a su respectiva variable en el lado izquierdo. Todas las expresiones en el lado derecho son evaluadas antes de realizar cualquier asignación.

El número de variables en el lado izquierdo y el número de valores en el lado derecho deben ser iguales:

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

Generalizando más, el lado derecho puede ser cualquier tipo de secuencia (cadena, lista, o tupla). Por ejemplo, para dividir una dirección de e-mail en nombre de usuario y dominio, se podría escribir:

```
>>> dir = 'monty@python.org'
>>> nombreus, dominio = dir.split('@')
```

El valor de retorno de `split` es una lista con dos elementos; el primer elemento es asignado a `nombreus`, el segundo a `dominio`.

```
>>> print(nombreus)
monty
>>> print(dominio)
python.org
```

10.4 Diccionarios y tuplas

Los diccionarios tienen un método llamado `items` que retorna una lista de tuplas, donde cada tupla es un par clave-valor:

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = list(d.items())
>>> print(t)
[('b', 1), ('a', 10), ('c', 22)]
```

Como sería de esperar en un diccionario, los elementos no tienen ningún orden en particular.

Aun así, puesto que la lista de tuplas es una lista, y las tuplas son comparables, ahora se puede ordenar la lista de tuplas. Convertir un diccionario en una lista de tuplas es una forma de obtener el contenido de un diccionario ordenado según sus claves:

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = list(d.items())
>>> t
[('b', 1), ('a', 10), ('c', 22)]
>>> t.sort()
>>> t
[('a', 10), ('b', 1), ('c', 22)]
```

La nueva lista está ordenada en orden alfabético ascendente de acuerdo al valor de sus claves.

10.5 Asignación múltiple con diccionarios

La combinación de `items`, asignación de tuplas, y `for`, produce un buen patrón de diseño de código para recorrer las claves y valores de un diccionario en un único bucle:

```
for clave, valor in list(d.items()):
    print(valor, clave)
```

Este bucle tiene dos *variables de iteración*, debido a que `items` retorna una lista de tuplas y `clave, valor` es una asignación en tupla que itera sucesivamente a través de cada uno de los pares clave-valor del diccionario.

Para cada iteración a través del bucle, tanto `clave` y `valor` van pasando al siguiente par clave-valor del diccionario (todavía en orden de dispersión).

La salida de este bucle es:

```
10 a
1 b
22 c
```

De nuevo, las claves están en orden de dispersión (es decir, ningún orden en particular).

Si se combinan esas dos técnicas, se puede imprimir el contenido de un diccionario ordenado por el *valor* almacenado en cada par clave-valor.

Para hacer esto, primero se crea una lista de tuplas donde cada tupla es `(valor, clave)`. El método `items` dará una lista de tuplas `(clave, valor)`, pero esta vez se pretende ordenar por valor, no por clave. Una vez que se ha construido la lista con las tuplas clave-valor, es sencillo ordenar la lista en orden inverso e imprimir la nueva lista ordenada.

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> l = list()
>>> for clave, valor in d.items() :
...     l.append( (valor, clave) )
...
```

```
>>> l
[(10, 'a'), (1, 'b'), (22, 'c')]
>>> l.sort(reverse=True)
>>> l
[(22, 'c'), (10, 'a'), (1, 'b')]
>>>
```

Al construir cuidadosamente la lista de tuplas para tener el valor como el primer elemento de cada tupla, es posible ordenar la lista de tuplas y obtener el contenido de un diccionario ordenado por valor.

10.6 Las palabras más comunes

Volviendo al ejemplo anterior del texto de *Romeo y Julieta*, Acto 2, Escena 2, podemos mejorar nuestro programa para hacer uso de esta técnica para imprimir las diez palabras más comunes en el texto, como se ve a continuación:

```
import string
manejador = open('romeo-full.txt')
contadores = dict()
for linea in manejador:
    linea = linea.translate(str.maketrans('', '', string.punctuation))
    linea = linea.lower()
    palabras = linea.split()
    for palabra in palabras:
        if palabra not in contadores:
            contadores[palabra] = 1
        else:
            contadores[palabra] += 1

# Ordenar el diccionario por valor
lst = list()
for clave, valor in list(contadores.items()):
    lst.append((valor, clave))

lst.sort(reverse=True)

for clave, valor in lst[:10]:
    print(clave, valor)

# Code: http://www.py4e.com/code3/count3.py
```

La primera parte del programa, la cual lee un archivo y construye un diccionario que mapea cada palabra con la cantidad de veces que se repite esa palabra en el documento, no ha cambiado. Pero en lugar de imprimir simplemente `contadores` y terminar el programa, ahora construimos una lista de tuplas `(val, key)` y luego se ordena la lista en orden inverso.

Puesto que el valor está primero, será utilizado para las comparaciones. Si hay más de una tupla con el mismo valor, se tendrá en cuenta el segundo elemento (la

clave), de forma que las tuplas cuyo valor es el mismo serán también ordenadas en orden alfabético según su clave.

Al final escribimos un elegante bucle `for` que hace una iteración con asignación múltiple e imprime las diez palabras más comunes, iterando a través de una parte de la lista (`lst[:10]`).

Ahora la salida finalmente tiene el aspecto que queríamos para nuestro análisis de frecuencia de palabras.

```
61 i
42 and
40 romeo
34 to
34 the
32 thou
32 juliet
30 that
29 my
24 thee
```

El hecho de que este complejo análisis y procesado de datos pueda ser realizado con un programa de Python de 19 líneas fácil de entender, es una razón de por qué Python es una buena elección como lenguaje para explorar información.

10.7 Uso de tuplas como claves en diccionarios

Dado que las tuplas son **dispersables** *(hashable)* y las listas no, si se quiere crear una clave **compuesta** para usar en un diccionario, se debe utilizar una tupla como clave.

Usaríamos por ejemplo una clave compuesta si quisiéramos crear un directorio telefónico que mapea pares appellido, nombre con números telefónicos. Asumiendo que hemos definido las variables `apellido`, `nombre`, y `número`, podríamos escribir una sentencia de asignación de diccionario como sigue:

```
directorio[apellido,nombre] = numero
```

La expresión entre corchetes es una tupla. Podríamos utilizar asignación de tuplas en un bucle `for` para recorrer este diccionario.

```
for apellido, nombre in directorio:
    print(nombre, apellido, directorio[apellido,nombre])
```

Este bucle recorre las claves en `directorio`, las cuales son tuplas. Asigna los elementos de cada tupla a `apellido` y `nombre`, después imprime el nombre y el número telefónico correspondiente.

10.8 Secuencias: cadenas, listas, y tuplas - ¡Dios mío!

Me he enfocado en listas de tuplas, pero casi todos los ejemplos de este capítulo funcionan también con listas de listas, tuplas de tuplas, y tuplas de listas. Para evitar enumerar todas las combinaciones posibles, a veces es más sencillo hablar de secuencias de secuencias.

En muchos contextos, los diferentes tipos de secuencias (cadenas, listas, y tuplas) pueden intercambiarse. Así que, ¿cómo y por qué elegir uno u otro?

Para comenzar con lo más obvio, las cadenas están más limitadas que otras secuencias, debido a que los elementos tienen que ser caracteres. Además, son inmutables. Si necesitas la capacidad de cambiar los caracteres en una cadena (en vez de crear una nueva), quizá prefieras utilizar una lista de caracteres.

Las listas son más comunes que las tuplas, principalmente porque son mutables. Pero hay algunos casos donde es preferible utilizar tuplas:

1. En algunos contextos, como una sentencia return, resulta sintácticamente más simple crear una tupla que una lista. En otros contextos, es posible que prefieras una lista.

2. Si quieres utilizar una secuencia como una clave en un diccionario, debes usar un tipo inmutable como una tupla o una cadena.

3. Si estás pasando una secuencia como argumento de una función, el uso de tuplas reduce la posibilidad de comportamientos inesperados debido a la creación de alias.

Dado que las tuplas son inmutables, no proporcionan métodos como `sort` y `reverse`, que modifican listas ya existentes. Sin embargo, Python proporciona las funciones internas `sorted` y `reversed`, que toman una secuencia como parámetro y devuelven una secuencia nueva con los mismos elementos en un orden diferente.

10.9 Depuración

Las listas, diccionarios y tuplas son conocidas de forma genérica como *estructuras de datos*; en este capítulo estamos comenzando a ver estructuras de datos compuestas, como listas de tuplas, y diccionarios que contienen tuplas como claves y listas como valores. Las estructuras de datos compuestas son útiles, pero también son propensas a lo que yo llamo *errores de modelado*; es decir, errores causados cuando una estructura de datos tiene el tipo, tamaño o composición incorrecto, o quizás al escribir una parte del código se nos olvidó cómo era el modelado de los datos y se introdujo un error. Por ejemplo, si estás esperando una lista con un entero y recibes un entero solamente (no en una lista), no funcionará.

10.10 Glosario

comparable Un tipo en el cual un valor puede ser revisado para ver si es mayor que, menor que, o igual a otro valor del mismo tipo. Los tipos que son comparables pueden ser puestos en una lista y ordenados.

estructura de datos Una collección de valores relacionados, normalmente organizados en listas, diccionarios, tuplas, etc.

DSU Abreviatura de "decorate-sort-undecorate (decorar-ordenar-quitar la decoración)", un patrón de diseño que implica construir una lista de tuplas, ordenarlas, y extraer parte del resultado.

reunir La operación de tratar una secuencia como una lista de argumentos.

hashable (dispersable) Un tipo que tiene una función de dispersión. Los tipos inmutables, como enteros, flotantes y cadenas son dispersables (hashables); los tipos mutables como listas y diccionarios no lo son.

dispersar La operación de tratar una secuencia como una lista de argumentos.

modelado (de una estructura de datos) Un resumen del tipo, tamaño, y composición de una estructura de datos.

singleton Una lista (u otra secuencia) con un único elemento.

tupla Una secuencia inmutable de elementos.

asignación por tuplas Una asignación con una secuencia en el lado derecho y una tupla de variables en el izquierdo. El lado derecho es evaluado y luego sus elementos son asignados a las variables en el lado izquierdo.

10.11 Ejercicios

Ejercicio 1: Revisa el programa anterior de este modo: Lee y analiza las líneas "From" y extrae las direcciones de correo. Cuenta el número de mensajes de cada persona utilizando un diccionario.

Después de que todos los datos hayan sido leídos, imprime la persona con más envíos, creando una lista de tuplas (contador, email) del diccionario. Después ordena la lista en orden inverso e imprime la persona que tiene más envíos.

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

Ingresa un nombre de archivo: mbox-short.txt
cwen@iupui.edu 5

Ingresa un nombre de archivo: mbox.txt
zqian@umich.edu 195
```

Ejercicio 2: Este programa cuenta la distribución de la hora del día para cada uno de los mensajes. Puedes extraer la hora de la línea "From" buscando la cadena horaria y luego dividiendo la cadena en partes utilizando el carácter colon. Una vez que hayas acumulado las cuentas para cada hora, imprime las cuentas, una por línea, ordenadas por hora tal como se muestra debajo.

```
python timeofday.py
Ingresa un nombre de archivo: mbox-short.txt
04 3
06 1
07 1
09 2
10 3
11 6
14 1
15 2
16 4
17 2
18 1
19 1
```

Ejercicio 3: Escribe un programa que lee un archivo e imprime las *letras* en order decreciente de frecuencia. El programa debe convertir todas las entradas a minúsculas y contar solamente las letras a-z. El programa no debe contar espacios, dígitos, signos de puntuación, o cualquier cosa que no sean las letras a-z. Encuentra ejemplos de texto en idiomas diferentes, y observa cómo la frecuencia de letras es diferente en cada idioma. Compara tus resultados con las tablas en https://es.wikipedia.org/wiki/Frecuencia_de_aparici%C3%B3n_de_letras.

Chapter 11

Expresiones regulares

Hasta ahora hemos leído archivos, buscando patrones y extrayendo varias secciones de líneas que hemos encontrado interesantes. Hemos usado métodos de cadenas como `split` y `find`, así como rebanado de listas y cadenas para extraer trozos de las líneas.

Esta tarea de buscar y extraer es tan común que Python tiene una librería muy poderosa llamada *expresiones regulares* que maneja varias de estas tareas de manera bastante elegante. La razón por la que no presentamos las expresiones regulares antes se debe a que, aunque son muy poderosas, son un poco más complicadas y toma algo de tiempo acostumbrarse a su sintaxis.

Las expresiones regulares casi son su propio lenguaje de programación en miniatura para buscar y analizar cadenas. De hecho, se han escrito libros enteros sobre las expresiones regulares. En este capítulo, solo cubriremos los aspectos básicos de las expresiones regulares. Para más información al respecto, recomendamos ver:

https://es.wikipedia.org/wiki/Expresi%C3%B3n_regular

https://docs.python.org/library/re.html

Se debe importar la librería de expresiones regulares `re` a tu programa antes de que puedas usarlas. La forma más simple de usar la librería de expresiones regulares es la función `search()` (N. del T.: "search" significa búsqueda). El siguiente programa demuestra una forma muy sencilla de usar esta función.

```
# Búsqueda de líneas que contengan 'From'
import re
man = open('mbox-short.txt')
for linea in man:
    linea = linea.rstrip()
    if re.search('From:', linea):
        print(linea)

# Code: http://www.py4e.com/code3/re01.py
```

Abrimos el archivo, revisamos cada línea, y usamos la expresión regular `search()` para imprimir solo las líneas que contengan la cadena "From". Este programa no

toma ventaja del auténtico poder de las expresiones regulares, ya que podríamos simplemente haber usado `line.find()` para lograr el mismo resultado.

El poder de las expresiones regulares se manifiesta cuando agregamos caracteres especiales a la cadena de búsqueda que nos permite controlar de manera más precisa qué líneas calzan con la cadena. Agregar estos caracteres especiales a nuestra expresión regular nos permitirá buscar coincidencias y extraer datos usando unas pocas líneas de código.

Por ejemplo, el signo de intercalación (N. del T.: "caret" en inglés, corresponde al signo ^) se utiliza en expresiones regulares para encontrar "el comienzo" de una lína. Podríamos cambiar nuestro programa para que solo retorne líneas en que tengan "From:" al comienzo, de la siguiente manera:

```
# Búsqueda de líneas que contengan 'From'
import re
man = open('mbox-short.txt')
for linea in man:
    linea = linea.rstrip()
    if re.search('^From:', linea):
        print(linea)

# Code: http://www.py4e.com/code3/re02.py
```

Ahora solo retornará líneas que *comiencen con* la cadena "From:". Este sigue siendo un ejemplo muy sencillo que podríamos haber implementado usando el método `startswith()` de la librería de cadenas. Pero sirve para presentar la idea de que las expresiones regulares contienen caracteres especiales que nos dan mayor control sobre qué coincidencias retornará la expresión regular.

11.1 Coincidencia de caracteres en expresiones regulares

Existen varios caracteres especiales que nos permiten construir expresiones regulares incluso más poderosas. El más común es el punto, que coincide con cualquier carácter.

En el siguiente ejemplo, la expresión regular `F..m:` coincidiría con las cadenas "From:", "Fxxm:", "F12m:", o "F!@m:", ya que los caracteres de punto en la expresión regular coinciden con cualquier carácter.

```
# # Búsqueda de líneas que comiencen con 'F', seguidas de
# 2 caracteres, seguidos de 'm:'
import re
man = open('mbox-short.txt')
for linea in man:
    linea = linea.rstrip()
    if re.search('^F..m:', linea):
        print(linea)

# Code: http://www.py4e.com/code3/re03.py
```

Esto resulta particularmente poderoso cuando se le combina con la habilidad de indicar que un carácter puede repetirse cualquier cantidad de veces usando los caracteres `*` o `+` en tu expresión regular. Estos caracteres especiales indican que en lugar de coincidir con un solo carácter en la cadena de búsqueda, coinciden con cero o más caracteres (en el caso del asterisco) o con uno o más caracteres (en el caso del signo de suma).

Podemos reducir más las líneas que coincidan usando un carácter *comodín* en el siguiente ejemplo:

```
# Búsqueda de líneas que comienzan con From y tienen una arroba
import re
man = open('mbox-short.txt')
for linea in man:
    linea = linea.rstrip()
    if re.search('^From:.+@', linea):
        print(linea)

# Code: http://www.py4e.com/code3/re04.py
```

La cadena `^From:.+@` retornará coincidencias con líneas que empiecen con "From:", seguidas de uno o más caracteres (`.+`), seguidas de un carácter @. Por lo tanto, la siguiente línea coincidirá:

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

Puede considerarse que el comodín `.+` se expande para abarcar todos los caracteres entre los signos : y @.

```
From:.+@
```

Conviene considerar que los signos de suma y los asteriscos "empujan". Por ejemplo, la siguiente cadena marcaría una coincidencia con el último signo @, ya que el `.+` "empujan" hacia afuera, como se muestra a continuación:

```
From: stephen.marquard@uct.ac.za, csev@umich.edu, and cwen @iupui.edu
```

Es posible indicar a un asterisco o signo de suma que no debe ser tan "ambicioso" agregando otro carácter. Revisa la documentación para obtener información sobre cómo desactivar este comportamiento ambicioso.

11.2 Extrayendo datos usando expresiones regulares

Si queremos extraer datos de una cadena en Python podemos usar el método `findall()` para extraer todas las subcadenas que coincidan con una expresión regular. Usemos el ejemplo de querer extraer cualquier secuencia que parezca una dirección email en cualquier línea, sin importar su formato. Por ejemplo, queremos extraer la dirección email de cada una de las siguientes líneas:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
          for <source@collab.sakaiproject.org>;
Received: (from apache@localhost)
Author: stephen.marquard@uct.ac.za
```

No queremos escribir código para cada tipo de líneas, dividiendo y rebanando de manera distinta en cada una. El siguiente programa usa `findall()` para encontrar las líneas que contienen direcciones de email y extraer una o más direcciones de cada línea.

```
import re
s = 'Un mensaje de csev@umich.edu para cwen@iupui.edu acerca de una junta @2PM'
lst = re.findall(r'\S+@\S+', s)
print(lst)

# Code: http://www.py4e.com/code3/re05.py
```

El método `findall()` busca en la cadena en el segundo argumento y retorna una lista de todas las cadenas que parecen ser direcciones de email. Estamos usando una secuencia de dos caracteres que coincide con un carácter distinto a un espacio en blanco (`\S`).

El resultado de la ejecución del programa debiera ser:

```
['csev@umich.edu', 'cwen@iupui.edu']
```

Traduciendo la expresión regular al castellano, estamos buscando subcadenas que tengan al menos un carácter que no sea un espacio, seguido de una @, seguido de al menos un carácter que no sea un espacio. La expresión `\S+` coincidirá con cuantos caracteres distintos de un espacio sea posible.

La expresión regular retornaría dos coincidencias (csev@umich.edu y cwen@iupui.edu), pero no coincidiría con la cadena "@2PM" porque no hay caracteres que no sean espacios en blanco *antes* del signo @. Podemos usar esta expresión regular en un programa para leer todas las líneas en un archivo e imprimir cualquier subcadena que pudiera ser una dirección de email de la siguiente manera:

```
# Búsqueda de líneas que tengan una arroba entre caracteres
import re
man = open('mbox-short.txt')
for linea in man:
    linea = linea.rstrip()
    x = re.findall(r'\S+@\S+', linea)
    if len(x) > 0:
        print(x)

# Code: http://www.py4e.com/code3/re06.py
```

Con esto, leemos cada línea y luego extraemos las subcadenas que coincidan con nuestra expresión regular. Dado que `findall()` retorna una lista, simplemente

revisamos si el número de elementos en ésta es mayor a cero e imprimir solo líneas donde encontramos al menos una subcadena que pudiera ser una dirección de email.

Si ejecutamos el programa en *mbox.txt* obtendremos el siguiente resultado:

```
['wagnermr@iupui.edu']
['cwen@iupui.edu']
['<postmaster@collab.sakaiproject.org>']
['<200801032122.m03LMFo4005148@nakamura.uits.iupui.edu>']
['<source@collab.sakaiproject.org>;']
['<source@collab.sakaiproject.org>;']
['<source@collab.sakaiproject.org>;']
['apache@localhost)']
['source@collab.sakaiproject.org;']
```

Algunas de las direcciones tienen caracteres incorrectos como "<" o ";" al comienzo o al final. Declaremos que solo estamos interesados en la parte de la cadena que comience y termine con una letra o un número.

Para lograr esto, usamos otra característica de las expresiones regulares. Los corchetes se usan para indicar un conjunto de caracteres que queremos aceptar como coincidencias. La secuencia `\S` retornará el conjunto de "caracteres que no sean un espacio en blanco". Ahora seremos un poco más explícitos en cuanto a los caracteres respecto de los cuales buscamos coincidencias.

Esta será nuestra nueva expresión regular:

```
[a-zA-Z0-9]\S*@\S*[a-zA-Z]
```

Esto se está complicando un poco; puedes ver por qué decimos que las expresiones regulares son un lenguaje en sí mismas. Traduciendo esta expresión regular, estamos buscando subcadenas que comiencen con *una* letra minúscula, letra mayúscula, o número "[a-zA-Z0-9]", seguida de cero o más caracteres que no sean un espacio (`\S*`), seguidos de un signo @, seguido de cero o más caracteres que no sean espacios en blanco (`\S*`), seguidos por una letra mayúscula o minúscula. Nótese que hemos cambiado de `+` a `*` para indicar cero o más caracteres que no sean espacios, ya que `[a-zA-Z0-9]` implica un carácter distinto de un espacio. Recuerda que el `*` o `+` se aplica al carácter inmediatamente a la izquierda del signo de suma o del asterisco.

Si usamos esta expresión en nuestro programas, nuestros datos quedarán mucho más depurados:

```
# Búsqueda de líneas que tengan una arroba entre caracteres
# Los caracteres deben ser una letra o un número
import re
man = open('mbox-short.txt')
for linea in man:
    linea = linea.rstrip()
    x = re.findall(r'[a-zA-Z0-9]\S+@\S+[a-zA-Z]', linea)
    if len(x) > 0:
        print(x)

# Code: http://www.py4e.com/code3/re07.py
```

```
...
['wagnermr@iupui.edu']
['cwen@iupui.edu']
['postmaster@collab.sakaiproject.org']
['200801032122.m03LMFo4005148@nakamura.uits.iupui.edu']
['source@collab.sakaiproject.org']
['source@collab.sakaiproject.org']
['source@collab.sakaiproject.org']
['apache@localhost']
```

Nótese que en las líneas donde aparece `source@collab.sakaiproject.org`, nuestra expresión regular eliminó dos caracteres al final de la cadena (">;"). Esto se debe a que, cuando agregamos `[a-zA-Z]` al final de nuestra expresión regular, estamos determinando que cualquier cadena que la expresión regular encuentre al analizar el texto debe terminar con una letra. Por lo tanto, cuando vea el ">" al final de "sakaiproject.org>;", simplemente se detiene en el último carácter que haya encontrado que coincida con ese criterio (en este caso, la "g" fue la última coincidencia).

Nótese también que el resultado de la ejecución del programa es una lista de Python que tiene una cadena como su único elemento.

11.3 Combinando búsqueda y extracción

Si quisiéramos encontrar los números en las líneas que empiecen con la cadena "X-", como por ejemplo:

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
```

no queremos cualquier número de coma flotante contenidos en cualquier línea. Solo queremos extraer los números de las líneas que tienen la sintaxis ya mencionada.

Podemos construir la siguiente expresión regular para seleccionar las líneas:

```
^X-.*: [0-9.]+
```

Traduciendo esto, estamos diciendo que queremos líneas que empiecen con `X-`, seguido por cero o más caracteres (`.*`), seguido por un carácter de dos puntos (`:`) y luego un espacio. Después del espacio, buscamos uno o más caracteres que sean, o bien un dígito (0-9), o bien un punto `[0-9.]+`. Nótese que al interior de los corchetes el punto efectivamente corresponde a un punto (es decir, no funciona como comodín entre corchetes).

La siguiente es una expresión bastante comprimida que solo retornará las líneas que nos interesan:

```
# Búsqueda de líneas que comiencen con 'X' seguida de cualquier caracter que
# no sea espacio y ':' seguido de un espacio y cualquier número.
# El número incluye decimales.
```

```
import re
man = open('mbox-short.txt')
for linea in man:
    linea = linea.rstrip()
    if re.search(r'^X\S*: [0-9.]+', linea):
        print(linea)

# Code: http://www.py4e.com/code3/re10.py
```

Cuando ejecutamos el programa, vemos que los datos han sido procesados, mostrando solo las líneas que buscamos.

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
X-DSPAM-Confidence: 0.6178
X-DSPAM-Probability: 0.0000
```

Ahora, debemos resolver el problema de extraer los númueros. Aunque sería bastante sencillo usar `split`, podemos echar mano a otra función de las expresiones regulares para buscar y analizar la línea a la vez.

Los paréntesis son otros caracteres especiales en las expresiones regulares. Al agregar paréntesis a una expresión regular, son ignorados a la hora de hacer coincidir la cadena. Pero cuando se usa `findall()`, los paréntesis indican que, aunque se quiere que toda la expresión coincida, solo interesa extraer una parte de la subcadena que coincida con la expresión regular.

Entonces, hacemos los siguientes cambios a nuestro programa:

```
# Búsqueda de líneas que comiencen con 'X' seguida de cualquier caracter que
# no sea espacio en blanco y ':' seguido de un espacio y un número.
# El número puede incluir decimales.
# Después imprimir el número si es mayor a cero.
import re
man = open('mbox-short.txt')
for linea in man:
    linea = linea.rstrip()
    x = re.findall(r'^X\S*: ([0-9.]+)', linea)
    if len(x) > 0:
        print(x)

# Code: http://www.py4e.com/code3/re11.py
```

En lugar de usar `search()`, agregamos paréntesis alrededos de la parte de la expresión regular que representa al número de coma flotante para indicar que solo queremos que `findall()` retorne la parte correspondiente a números de coma flotante de la cadena retornada.

El resultado de este programa es el siguiente:

```
['0.8475']
```

```
['0.0000']
['0.6178']
['0.0000']
['0.6961']
['0.0000']
..
```

Los números siguen estando en una lista y deben ser convertidos de cadenas a números de coma flotante, pero hemos usado las expresiones regulares para buscar y extraer la información que consideramos interesante.

Otro ejemplo de esta técnica: si revisan este archivo, verán una serie de líneas en el formulario:

```
Detalles: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
```

Si quisiéramos extraer todos los números de revisión (el número entero al final de esas líneas) usando la misma técnica del ejemplo anterior, podríamos escribir el siguiente programa:

```
# Búsqueda de líneas que comiencen con 'Details: rev='
# seguido de números y '.'
# Después imprimir el número si es mayor a cero
import re
man = open('mbox-short.txt')
for linea in man:
    linea = linea.rstrip()
    x = re.findall('^Details:.*rev=([0-9.]+)', linea)
    if len(x) > 0:
        print(x)

# Code: http://www.py4e.com/code3/re12.py
```

Traducción de la expresión regular: estamos buscando líneas que empiecen con `Details:`, seguida de cualquier número de caracteres (`.*`), seguida de `rev=`, y después de uno o más dígitos. Queremos encontrar líneas que coincidan con toda la expresión pero solo queremos extraer el número entero al final de la línea, por lo que ponemos `[0-9]+` entre paréntesis.

Al ejecutar el programa, obtenemos el siguiente resultado:

```
['39772']
['39771']
['39770']
['39769']
...
```

Recuerda que la expresión `[0-9]+` es "ambiciosa" e intentará formar una cadena de dígitos lo más larga posible antes de extraerlos. Este comportamiento "ambicioso" es la razón por la que obtenemos los cinco dígitos de cada número. La expresiones

regular se expande en ambas direcciones hasta que encuentra, o bien un carácter que no sea un dígito, o bien el comienzo o final de una línea.

Ahora podemos usar expresiones regulares para volver a resolver un ejercicio que hicimos antes, en el que nos interesaba la hora de cada email. En su momento, buscamos las líneas:

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

con la intención de extraer la hora del día en cada línea. Antes logramos esto haciendo dos llamadas a `split`. Primero se dividía la línea en palabras y luego tomábamos la quinta palabra y la dividíamos de nuevo en el carácter ":" para obtener los dos caracteres que nos interesaban.

Aunque esta solución funciona, es el resultado de código bastante frágil, que depende de asumir que las líneas tienen el formato adecuado. Si bien puedes agregar chequeos (o un gran bloque de try/except) para asegurarte que el programa no falle al encontrar líneas mal formateadas, esto hará que el programa aumente a 10 o 15 líneas de código, que además serán difíciles de leer.

Podemos lograr el resultado de forma mucho más simple utilizando la siguiente expresión regular:

```
^From .* [0-9][0-9]:
```

La traducción de esta expresión regular sería que estamos buscando líneas que empiecen con `From` (nótese el espacio), seguido de cualquier número de caracteres (`.*`), seguidos de un espacio en blanco, seguido de dos dígitos `[0-9][0-9]`, seguidos de un carácter ":". Esa es la definición de la clase de líneas que buscamos.

Para extraer solo la hora usando `findall()`, agregamos paréntesis alrededor de los dos dígitos, de la siguiente manera:

```
^From .* ([0-9][0-9]):
```

Esto resultará en el siguiente programa:

```
# Búsqueda de líneas que comienzan con From y un caracter seguido
# de un número de dos dígitos entre 00 y 99 seguido de ':'
# Después imprimir el número si este es mayor a cero
import re
man = open('mbox-short.txt')
for linea in man:
    linea = linea.rstrip()
    x = re.findall('^From .* ([0-9][0-9]):', linea)
    if len(x) > 0: print(x)

# Code: http://www.py4e.com/code3/re13.py
```

Al ejecutar el programa, obtendremos el siguiente resultado:

```
['09']
['18']
['16']
['15']
...
```

11.4 Escapado de Caracteres

Dado que en expresiones regulares usamos caracteres especiales para encontrar el comienzo o final de una línea o especificar comodines, necesitamos una forma de indicar que esos caracteres son "normales" y queremos encontrar la coincidencia con el carácter específico, como un "$" o "^".

Podemos indicar que queremos encontrar la coincidencia con un carácter anteponiéndole una barra invertida. Por ejemplo, podemos encontrar cantidades de dinero utilizando la siguiente expresión regular:

```
import re
x = 'We just received $10.00 for cookies.'
y = re.findall('\$[0-9.]+',x)
```

Dado que antepusimos la barra invertida al signo "$", encuentra una coincidencia con el signo en la cadena en lugar de con el final de la línea, y el resto de la expresión regular coincide con uno o más dígitos del carácter "." *Nota:* dentro de los corchetes, los caracteres no se consideran "especiales". Por tanto, al escribir `[0-9.]`, efectivamente significa dígitos o un punto. Cuando no está entre corchetes, el punto es el carácter "comodín" que coincide con cualquier carácter. Cuando está dentro de corchetes, un punto es un punto.

11.5 Resumen

Aunque solo hemos escarbado la superficie de las expresiones regulares, hemos aprendido un poco sobre su lenguaje. Son cadenas de búsqueda con caracteres especiales en su interior, los que comunican tus deseos al sistema de expresiones regulares respecto de qué se considera una coincidencia y qué información es extraída de las cadenas coincidentes. A continuación tenemos algunos de estos caracteres y secuencias especiales:

`^` Coincide con el comienzo de la línea.

`$` Coincide con el final de la línea

`.` Coincide con cualquier carácter (un comodín).

`\s` Coincide con un espacio en blanco.

`\S` Coincide con un carácter que no sea un espacio en blanco (el opuesto a \s).

`*` Se aplica al carácter o caracteres inmediatamente anteriores, indicando que pueden coincidir cero o más veces.

`*?` Se aplica al carácter o caracteres inmediatamente anteriores, indicando que coinciden cero o más veces en modo "no ambicioso".

`+` Se aplica al carácter o caracteres inmediatamente anteriores, indicando que pueden coincidir una o más veces.

`+?` Se aplica al carácter o caracteres inmediatamente anteriores, indicando que pueden coincidir una o más veces en modo "no ambicioso".

`?` Se aplica al carácter o caracteres inmediatamente anteriores, indicando que puede coincidir cero o una vez.

`??` Se aplica al carácter o caracteres inmediatamente anteriores, indicando que puede coincidir cero o una vez en modo "no ambicioso".

`[aeiou]` Coincide con un solo carácter, siempre que éste se encuentre dentro del conjunto especificado. En este caso, coincidiría con "a", "e", "i", "o", o "u", pero no con otros caracteres.

`[a-z0-9]` Se pueden especificar rangos de caracteres utilizando el signo menos. En este caso, sería un solo carácter que debe ser una letra minúscula o un dígito.

`[^A-Za-z]` Cuando el primer carácter en la notación del conjunto es "^", invierte la lógica. En este ejemplo, habría coincidencia con un solo carácter que *no sea* una letra mayúscula o una letra minúscula.

`( )` Cuando se agregan paréntesis a una expresión regular, son ignorados para propósitos de encontrar coincidencias, pero permiten extraer un subconjunto determinado de la cadena en que se encuentra la coincidencia, en lugar de toda la cadena como cuando se utiliza `findall()`.

`\b` Coincide con una cadena vacía, pero solo al comienzo o al final de una palabra.

`\B` Concide con una cadena vacía, pero no al comienzo o al final de una palabra.

`\d` Coincide con cualquier dígito decimal; equivalente al conjunto [0-9].

`\D` Coincide con cualquier carácter que no sea un dígito; equivalente al conjunto [^0-9].

11.6 Sección adicional para usuarios de Unix / Linux

El soporte para buscar archivos usando expresiones regulares viene incluido en el sistema operativo Unix desde la década de 1960, y está disponible en prácticamente todos los lenguajes de programación de una u otra forma.

De hecho, hay un programa de línea de comandos incluido en Unix llamado *grep* (Generalized Regular Expression Parser// Analizador Generalizado de Expresiones Regulares) que hace prácticamente lo mismo que los ejemplos que hemos dado en este capítulo que utilizan `search()`. Por tanto, si usas un sistema Macintosh o Linux, puedes probar los siguientes comandos en tu ventana de línea de comandos.

```
$ grep '^From:' mbox-short.txt
From: stephen.marquard@uct.ac.za
```

```
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
```

Esto ordena a `grep` mostrar las líneas que comienzan con la cadena "From:" en el archivo *mbox-short.txt*. Si experimentas un poco con el comando `grep` y lees su documentación, encontrarás algunas sutiles diferencias entre el soporte de expresiones regulares en Python y en `grep`. Por ejemplo, `grep` no reconoce el carácter de no espacio en blanco `\S`, por lo que deberás usar la notación de conjuntos `[^ ]`, que es un poco más compleja, y que significa que encontrará una coincidencia con cualquier carácter que no sea un espacio en blanco.

11.7 Depuración

Python incluye una documentación simple y rudimentaria que puede ser de gran ayuda si necesitas revisar para encontrar el nombre exacto de algún método. Esta documentación puede revisarse en el intérprete de Python en modo interactivo.

Para mostrar el sistema de ayuda interactivo, se utiliza el comando `help()`.

```
>>> help()

help> modules
```

Si sabes qué método quieres usar, puedes utilizar el comando `dir()` para encontrar los métodos que contiene el módulo, de la siguiente manera:

```
>>> import re
>>> dir(re)
[.. 'compile', 'copy_reg', 'error', 'escape', 'findall',
'finditer', 'match', 'purge', 'search', 'split', 'sre_compile',
'sre_parse', 'sub', 'subn', 'sys', 'template']
```

También puedes obtener una pequeña porción de la documentación de un método en particular usando el comando dir.

```
>>> help (re.search)
Help on function search in module re:

search(pattern, string, flags=0)
    Scan through string looking for a match to the pattern, returning
    a match object, or None if no match was found.
>>>
```

La documentación incluida no es muy exhaustiva, pero puede ser útil si estás con prisa o no tienes acceso a un navegador o motor de búsqueda.

11.8 Glosario

código frágil Código que funciona cuando los datos se encuentran en un formato específico pero tiene a romperse si éste no se cumple. Lo llamamos "código frágil" porque se rompe con facilidad.

coincidencia ambiciosa La idea de que los caracteres `+` y `*` en una expresión regular se expanden hacia afuera para coincidir con la mayor cadena posible.

grep Un comando disponible en la mayoría de los sistemas Unix que busca en archivos de texto, buscando líneas que coincidan con expresiones regulares. El nombre del comando significa "Generalized Regular Expression Parser, o bien"Analizador Generalizado de Expresiones Regulares".

expresión regular Un lenguaje para encontrar cadenas más complejas en una búsqueda. Una expresión regular puede contener caracteres especiales que indiquen que una búsqueda solo coincida al comienzo o al final de una línea, junto con muchas otras capacidades similares.

comodín Un carácter especial que coincide con cualquier carácter. En expresiones regulares, el carácter comodín es el punto.

11.9 Ejercicios

Ejercicio uno: escribe un programa simple que simule la operación del comando `grep` en Unix. Debe pedir al usuario que ingrese una expresión regular y cuente el número de líneas que coincidan con ésta:

```
$ python grep.py
Ingresa una expresión regular: ^Author
mbox.txt tiene 1798 líneas que coinciden con ^Author

$ python grep.py
Ingresa una expresión regular: ^X-
mbox.txt tiene 14368 líneas que coinciden con ^X-

$ python grep.py
Ingresa una expresión regular: java$
mbox.txt tiene 4175 líneas que coinciden con java$
```

Ejercicio 2: escribe un programa que busque líneas en la forma:

```
New Revision: 39772
```

Extrae el número de cada línea usando una expresión regular y el método `findall()`. Registra el promedio de esos números e imprímelo.

```
Ingresa nombre de archivo: mbox.txt
38444.0323119

Ingresa nombre de archivo: mbox-short.txt
39756.9259259
```

Chapter 12

Programas en red

Aunque muchos de los ejemplos en este libro se han enfocado en leer archivos y buscar datos en ellos, hay muchas fuentes de información diferentes si se tiene en cuenta el Internet.

En este capítulo fingiremos ser un navegador web y recuperaremos páginas web utilizando el Protocolo de Transporte de Hipertexto (HyperText Transfer Protocol - HTTP). Luego revisaremos los datos de esas páginas web y los analizaremos.

12.1 Protocolo de Transporte de Hipertexto - HTTP

El protocolo de red que hace funcionar la web es en realidad bastante simple, y existe un soporte integrado en Python llamado `sockets`, el cual hace que resulte muy fácil realizar conexiones de red y recuperar datos a través de esas conexiones desde un programa de Python.

Un socket es muy parecido a un archivo, a excepción de que proporciona una conexión de doble sentido entre dos programas. Es posible tanto leer como escribir en un mismo socket. Si se escribe algo en un socket, es enviado a la aplicación que está al otro lado de éste. Si se lee desde el socket, se obtienen los datos que la otra aplicación ha enviado.

Pero si intentas leer un socket cuando el programa que está del otro lado del socket no ha enviado ningún dato, puedes sentarte y esperar. Si los programas en ambos extremos del socket simplemente esperan por datos sin enviar nada, van a esperar por mucho, mucho tiempo, así que una parte importante de los programas que se comunican a través de internet consiste en tener algún tipo de protocolo.

Un protocolo es un conjunto de reglas precisas que determinan quién va primero, qué debe hacer, cuáles son las respuestas siguientes para ese mensaje, quién envía a continuación, etcétera. En cierto sentido las aplicaciones a ambos lados del socket están interpretando un baile y cada una debe estar segura de no pisar los pies de la otra.

Hay muchos documentos que describen estos protocolos de red. El Protocolo de Transporte de Hipertext está descrito en el siguiente documento:

https://www.w3.org/Protocols/rfc2616/rfc2616.txt

Se trata de un documento largo y complejo de 176 páginas, con un montón de detalles. Si lo encuentras interesante, siéntete libre de leerlo completo. Pero si echas un vistazo alrededor de la página 36 del RFC2616, encontrarás la sintaxis para las peticiones GET. Para pedir un documento a un servidor web, hacemos una conexión al servidor `www.pr4e.org` en el puerto 80, y luego enviamos una línea como esta:

```
GET http://data.pr4e.org/romeo.txt HTTP/1.0
```

en la cual el segundo parámetro es la página web que estamos solicitando, y a continuación enviamos una línea en blanco. El servidor web responderá con una cabecera que contiene información acerca del documento y una línea en blanco, seguido por el contenido del documento.

12.2 El navegador web más sencillo del mundo

Quizá la manera más sencilla de demostrar cómo funciona el protocolo HTTP sea escribir un programa en Python muy sencillo, que realice una conexión a un servidor web y siga las reglas del protocolo HTTP para solicitar un documento y mostrar lo que el servidor envía de regreso.

```
import socket

misock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
misock.connect(('data.pr4e.org', 80))
cmd = 'GET http://data.pr4e.org/romeo.txt HTTP/1.0\r\n\r\n'.encode()
misock.send(cmd)

while True:
    datos = misock.recv(512)
    if len(datos) < 1:
        break
    print(datos.decode(),end='')

misock.close()

# Code: http://www.py4e.com/code3/socket1.py
```

En primer lugar, el programa realiza una conexión al puerto 80 del servidor www.py4e.com. Puesto que nuestro programa está jugando el rol de "navegador web", el protocolo HTTP dice que debemos enviar el comando GET seguido de una línea en blanco. `\r\n` representa un salto de línea (end of line, o EOL en inglés), así que `\r\n\r\n` significa que no hay nada entre dos secuencias de salto de línea. Ese es el equivalente de una línea en blanco.

Una vez que enviamos esa línea en blanco, escribimos un bucle que recibe los datos desde el socket en bloques de 512 caracteres y los imprime en pantalla hasta que

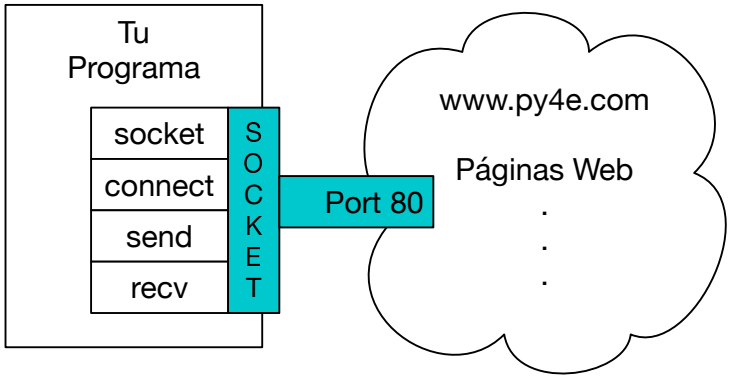

Figure 12.1: Conexión de un socket

no quedan más datos por leer (es decir, la llamada a recv() devuelve una cadena vacía).

El programa produce la salida siguiente:

```
HTTP/1.1 200 OK
Date: Wed, 11 Apr 2018 18:52:55 GMT
Server: Apache/2.4.7 (Ubuntu)
Last-Modified: Sat, 13 May 2017 11:22:22 GMT
ETag: "a7-54f6609245537"
Accept-Ranges: bytes
Content-Length: 167
Cache-Control: max-age=0, no-cache, no-store, must-revalidate
Pragma: no-cache
Expires: Wed, 11 Jan 1984 05:00:00 GMT
Connection: close
Content-Type: text/plain

But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

La salida comienza con la cabecera que el servidor envía para describir el documento. Por ejemplo, la cabecera `Content-Type` indica que el documento es un documento de texto sin formato (`text/plain`).

Después de que el servidor nos envía la cabecera, añade una línea en blanco para indicar el final de la cabecera, y después envía los datos reales del archivo *romeo.txt*.

Este ejemplo muestra cómo hacer una conexión de red de bajo nivel con sockets. Los sockets pueden ser usados para comunicarse con un servidor web, con un servidor de correo, o con muchos otros tipos de servidores. Todo lo que se necesita es encontrar el documento que describe el protocolo correspondiente y escribir el código para enviar y recibir los datos de acuerdo a ese protocolo.

Sin embargo, como el protocolo que se usa con más frecuencia es el protocolo web HTTP, Python tiene una librería especial diseñada especialmente para trabajar con éste para recibir documentos y datos a través de la web.

Uno de los requisitos para utilizar el protocolo HTTP es la necesidad de enviar y recibir datos como objectos binarios, en vez de cadenas. En el ejemplo anterior, los métodos `encode()` y `decode()` convierten cadenas en objectos binarios y viceversa.

El siguiente ejemplo utiliza la notación `b''` para especificar que una variable debe ser almacenada como un objeto binario. `encode()` y `b''` son equivalentes.

```
>>> b'Hola mundo'
b'Hola mundo'
>>> 'Hola mundo'.encode()
b'Hola mundo'
```

12.3 Recepción de una imagen mediante HTTP

En el ejemplo anterior, recibimos un archivo de texto sin formato que tenía saltos de línea en su interior, y lo único que hicimos cuando el programa se ejecutó fue copiar los datos a la pantalla. Podemos utilizar un programa similar para recibir una imagen utilizando HTTP. En vez de copiar los datos a la pantalla conforme va funcionando el programa, vamos a guardar los datos en una cadena, remover la cabecera, y después guardar los datos de la imagen en un archivo tal como se muestra a continuación:

```
import socket
import time

SERVIDOR = 'data.pr4e.org'
PUERTO = 80
misock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
misock.connect((SERVIDOR, PUERTO))
misock.sendall(b'GET http://data.pr4e.org/cover3.jpg HTTP/1.0\r\n\r\n')
contador = 0
imagen = b""

while True:
    datos = misock.recv(5120)
    if len(datos) < 1: break
    #time.sleep(0.25)
    contador = contador + len(datos)
    print(len(datos), contador)
    imagen = imagen + datos

misock.close()

# Búsqueda del final de la cabecera (2 CRLF)
pos = imagen.find(b"\r\n\r\n")
print('Header length', pos)
print(imagen[:pos].decode())

# Ignorar la cabera y guardar los datos de la imagen
imagen = imagen[pos+4:]
```

```
fhand = open("cosa.jpg", "wb")
fhand.write(imagen)
fhand.close()

# Code: http://www.py4e.com/code3/urljpeg.py
```

Cuando el programa corre, produce la siguiente salida:

```
$ python urljpeg.py
5120 5120
5120 10240
4240 14480
5120 19600
...
5120 214000
3200 217200
5120 222320
5120 227440
3167 230607
Header length 394
HTTP/1.1 200 OK
Date: Fri, 21 Feb 2020 01:45:41 GMT
Server: Apache/2.4.18 (Ubuntu)
Last-Modified: Mon, 15 May 2017 12:27:40 GMT
ETag: "38342-54f8f2e5b6277"
Accept-Ranges: bytes
Content-Length: 230210
Vary: Accept-Encoding
Cache-Control: max-age=0, no-cache, no-store, must-revalidate
Pragma: no-cache
Expires: Wed, 11 Jan 1984 05:00:00 GMT
Connection: close
Content-Type: image/jpeg
```

Puedes observar que para esta url, la cabecera `Content-Type` indica que el cuerpo del documento es una imagen (`image/jpeg`). Una vez que el programa termina, puedes ver los datos de la imagen abriendo el archivo `cosa.jpg` en un visor de imágenes.

Al ejecutar el programa, se puede ver que no se obtienen 5120 caracteres cada vez que llamamos el método `recv()`. Se obtienen tantos caracteres como hayan sido transferidos por el servidor web hacia nosotros a través de la red en el momento de la llamada a `recv()`. En este emeplo, se obtienen al menos 3200 caracteres cada vez que solicitamos hasta 5120 caracteres de datos.

Los resultados pueden variar dependiendo de tu velocidad de internet. Además, observa que en la última llamada a `recv()` obtenemos 3167 bytes, lo cual es el final de la cadena, y en la siguiente llamada a `recv()` obtenemos una cadena de longitud cero que indica que el servidor ya ha llamado `close()` en su lado del socket, y por lo tanto no quedan más datos pendientes por recibir.

Podemos retardar las llamadas sucesivas a `recv()` al descomentar la llamada a `time.sleep()`. De esta forma, esperamos un cuarto de segundo después de cada llamada de modo que el servidor puede "adelantarse" a nosotros y enviarnos más

datos antes de que llamemos de nuevo a `recv()`. Con el retraso, esta vez el programa se ejecuta así:

```
$ python urljpeg.py
5120 5120
5120 10240
5120 15360
...
5120 225280
5120 230400
208 230608
Header length 394
HTTP/1.1 200 OK
Date: Fri, 21 Feb 2020 01:57:31 GMT
Server: Apache/2.4.18 (Ubuntu)
Last-Modified: Mon, 15 May 2017 12:27:40 GMT
ETag: "38342-54f8f2e5b6277"
Accept-Ranges: bytes
Content-Length: 230210
Vary: Accept-Encoding
Cache-Control: max-age=0, no-cache, no-store, must-revalidate
Pragma: no-cache
Expires: Wed, 11 Jan 1984 05:00:00 GMT
Connection: close
Content-Type: image/jpeg
```

Ahora todas las llamadas a `recv()`, excepto la primera y la última, nos dan 5120 caracteres cada vez que solicitamos más datos.

Existe un buffer entre el servidor que hace las peticiones `send()` y nuestra aplicación que hace las peticiones `recv()`. Cuando ejecutamos el programa con el retraso activado, en algún momento el servidor podría llenar el buffer del socket y verse forzado a detenerse hasta que nuestro programa empiece a vaciar ese buffer. La detención de la aplicación que envía los datos o de la que los recibe se llama "control de flujo".

12.4 Recepción de páginas web con `urllib`

Aunque podemos enviar y recibir datos manualmente mediante HTTP utilizando la librería socket, existe una forma mucho más simple para realizar esta habitual tarea en Python, utilizando la librería `urllib`.

Utilizando `urllib`, es posible tratar una página web de forma parecida a un archivo. Se puede indicar simplemente qué página web se desea recuperar y `urllib` se encargará de manejar todos los detalles referentes al protocolo HTTP y a la cabecera.

El código equivalente para leer el archivo *romeo.txt* desde la web usando `urllib` es el siguiente:

```
import urllib.request

man_a = urllib.request.urlopen('http://data.pr4e.org/romeo.txt')
```

```
for linea in man_a:
    print(linea.decode().strip())

# Code: http://www.py4e.com/code3/urllib1.py
```

Una vez que la página web ha sido abierta con `urllib.urlopen`, se puede tratar como un archivo y leer a través de ella utilizando un bucle `for`.

Cuando el programa se ejecuta, en su salida sólo vemos el contenido del archivo. Las cabeceras siguen enviándose, pero el código de `urllib` se encarga de manejarlas y solamente nos devuelve los datos.

```
But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

Como ejemplo, podemos escribir un programa para obtener los datos de `romeo.txt` y calcular la frecuencia de cada palabra en el archivo de la forma siguiente:

```
import urllib.request, urllib.parse, urllib.error

man_a = urllib.request.urlopen('http://data.pr4e.org/romeo.txt')

contadores = dict()
for linea in man_a:
    palabras = linea.decode().split()
    for palabra in palabras:
        contadores[palabra] = contadores.get(palabra, 0) + 1

print(contadores)

# Code: http://www.py4e.com/code3/urlwords.py
```

De nuevo vemos que, una vez abierta la página web, se puede leer como si fuera un archivo local.

12.5 Leyendo archivos binarios con `urllib`

A veces se desea obtener un archivo que no es de texto (o binario) tal como una imagen o un archivo de video. Los datos en esos archivos generalmente no son útiles para ser impresos en pantalla, pero se puede hacer fácilmente una copia de una URL a un archivo local en el disco duro utilizando `urllib`.

El método consiste en abrir la dirección URL y utilizar `read` para descargar todo el contenido del documento en una cadena (`img`) para después escribir esa información a un archivo local, tal como se muestra a continuación:

```
import urllib.request, urllib.parse, urllib.error

img = urllib.request.urlopen('http://data.pr4e.org/cover3.jpg').read()
man_a = open('portada.jpg', 'wb')
man_a.write(img)
man_a.close()

# Code: http://www.py4e.com/code3/curl1.py
```

Este programa lee todos los datos que recibe de la red y los almacena en la variable `img` en la memoria principal de la computadora. Después, abre el archivo `portada.jpg` y escribe los datos en el disco. El argumento `wb` en la función `open()` abre un archivo binario en modo de escritura solamente. Este programa funcionará siempre y cuando el tamaño del archivo sea menor que el tamaño de la memoria de la computadora.

Aún asi, si es un archivo grande de audio o video, este programa podría fallar o al menos ejecutarse sumamente lento cuando la memoria de la computadora se haya agotado. Para evitar que la memoria se termine, almacenamos los datos en bloques (o buffers) y luego escribimos cada bloque en el disco antes de obtener el siguiente bloque. De esta forma, el programa puede leer archivos de cualquier tamaño sin utilizar toda la memoria disponible en la computadora.

```
import urllib.request, urllib.parse, urllib.error

img = urllib.request.urlopen('http://data.pr4e.org/cover3.jpg')
man_a = open('portada.jpg', 'wb')
tamano = 0
while True:
    info = img.read(100000)
    if len(info) < 1: break
    tamano = tamano + len(info)
    man_a.write(info)

print(tamano, 'caracteres copiados.')
man_a.close()

# Code: http://www.py4e.com/code3/curl2.py
```

En este ejemplo, leemos solamente 100,000 caracteres a la vez, y después los escribimos al archivo `portada.jpg` antes de obtener los siguientes 100,000 caracteres de datos desde la web.

Este programa se ejecuta como se observa a continuación:

```
python curl2.py
230210 caracteres copiados.
```

12.6 Análisis the HTML y rascado de la web

Uno de los usos más comunes de las capacidades de `urllib` en Python es *rascar* la web. El rascado de la web es cuando escribimos un programa que pretende ser un navegador web y recupera páginas, examinando luego los datos de esas páginas para encontrar ciertos patrones.

Por ejemplo, un motor de búsqueda como Google buscará el código de una página web, extraerá los enlaces a otras paginas y las recuperará, extrayendo los enlaces que haya en ellas y así sucesivamente. Utilizando esta técnica, las *arañas* de Google alcanzan a casi todas las páginas de la web.

Google utiliza también la frecuencia con que las páginas que encuentra enlazan hacia una página concreta para calcular la "importancia" de esa página, y la posición en la que debe aparecer dentro de sus resultados de búsqueda.

12.7 Análisis de HTML mediante expresiones regulares

Una forma sencilla de analizar HTML consiste en utilizar expresiones regulares para hacer búsquedas repetitivas que extraigan subcadenas coincidentes con un patrón en particular.

Aquí tenemos una página web simple:

```
<h1>La Primera Página</h1>
<p>
Si quieres, puedes visitar la
<a href="http://www.dr-chuck.com/page2.htm">
Segunda Página</a>.
</p>
```

Podemos construir una expresión regular bien formada para buscar y extraer los valores de los enlaces del texto anterior, de esta forma:

```
href="http[s]?://.+?"
```

Nuestra expresión regular busca cadenas que comiencen con "href="http://" o "href="https://", seguido de uno o más caracteres (`.+?`), seguidos por otra comilla doble. El signo de interrogación después de `[s]?` indica que la coincidencia debe ser hecha en modo "no-codicioso", en vez de en modo "codicioso". Una búsqueda no-codiciosa intenta encontrar la cadena coincidente *más pequeña* posible, mientras que una búsqueda codiciosa intentaría localizar la cadena coincidente *más grande*.

Añadimos paréntesis a nuestra expresión regular para indicar qué parte de la cadena localizada queremos extraer, y obtenemos el siguiente programa:

```
# Búsqueda de valores de enlaces dentro de una URL ingresada
import urllib.request, urllib.parse, urllib.error
```

```
import re
import ssl

# Ignorar errores de certificado SSL
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

url = input('Introduzca - ')
html = urllib.request.urlopen(url).read()
enlaces = re.findall(b'href="(http[s]?://.*?)"', html)
for enlace in enlaces:
    print(enlace.decode())

# Code: http://www.py4e.com/code3/urlregex.py
```

La librería `ssl` permite a nuestro programa acceder a los sitios web que estrictamente requieren HTTPS. El método `read` devuelve código fuente en HTML como un objeto binario en vez de devolver un objeto HTTPResponse. El método de expresiones regulares `findall` nos da una lista de todas las cadenas que coinciden con la expresión regular, devolviendo solamente el texto del enlace entre las comillas dobles.

Cuando corremos el programa e ingresamos una URL, obtenemos lo siguiente:

```
Introduzca - https://docs.python.org
https://docs.python.org/3/index.html
https://www.python.org/
https://devguide.python.org/docquality/#helping-with-documentation
https://docs.python.org/3.9/
https://docs.python.org/3.8/
https://docs.python.org/3.7/
https://docs.python.org/3.6/
https://docs.python.org/3.5/
https://docs.python.org/2.7/
https://www.python.org/doc/versions/
https://www.python.org/dev/peps/
https://wiki.python.org/moin/BeginnersGuide
https://wiki.python.org/moin/PythonBooks
https://www.python.org/doc/av/
https://devguide.python.org/
https://www.python.org/
https://www.python.org/psf/donations/
https://docs.python.org/3/bugs.html
https://www.sphinx-doc.org/
```

Las expresiones regulares funcionan muy bien cuando el HTML está bien formateado y es predecible. Pero dado que ahí afuera hay muchas páginas con HTML "defectuoso", una solución que solo utilice expresiones regulares podría perder algunos enlaces válidos, o bien terminar obteniendo datos erróneos.

Esto se puede resolver utilizando una librería robusta de análisis de HTML.

12.8 Análisis de HTML mediante BeautifulSoup

A pesar de que HTML es parecido a XML[1] y que algunas páginas son construidas cuidadosamente para ser XML, la mayoría del HTML generalmente está incompleto, de modo que puede causar que un analizador de XML rechace una página HTML completa por estar formada inadecuadamente.

Hay varias librerías en Python que pueden ayudarte a analizar HTML y extraer datos de las páginas. Cada una tiene sus fortalezas y debilidades, por lo que puedes elegir una basada en tus necesidades.

Por ejemplo, vamos a analizar una entrada HTML cualquiera y a extraer enlaces utilizando la librería *BeautifulSoup*. BeautifulSoup tolera código HTML bastante defectuoso y aún así te deja extraer los datos que necesitas. Puedes descargar e instalar el código de BeautifulSoup desde:

https://pypi.python.org/pypi/beautifulsoup4

La información acerca de la instalación de BeautifulSoup utilizando la herramienta de Python Package Index (Índice de Paquete de Python) `pip`, disponible en:

https://packaging.python.org/tutorials/installing-packages/

Vamos a utilizar `urllib` para leer la página y después utilizaremos `BeautifulSoup` para extraer los atributos `href` de las etiquetas de anclaje (`a`).

```
# Para ejecutar este programa descarga BeautifulSoup
# https://pypi.python.org/pypi/beautifulsoup4

# O descarga el archivo
# http://www.py4e.com/code3/bs4.zip
# y descomprimelo en el mismo directorio que este archivo

import urllib.request, urllib.parse, urllib.error
from bs4 import BeautifulSoup
import ssl

# Ignorar errores de certificado SSL
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

url = input('Introduzca - ')
html = urllib.request.urlopen(url, context=ctx).read()
sopa = BeautifulSoup(html, 'html.parser')

# Recuperar todas las etiquetas de anclaje
etiquetas = sopa('a')
for etiqueta in etiquetas:
    print(etiqueta.get('href', None))

# Code: http://www.py4e.com/code3/urllinks.py
```

[1] El formato XML es descrito en el siguiente capítulo.

El programa solicita una dirección web, luego la abre, lee los datos y se los pasa al analizador BeautifulSoup. Luego, recupera todas las etiquetas de anclaje e imprime en pantalla el atributo `href` de cada una de ellas.

Cuando el programa se ejecuta, produce lo siguiente:

```
Introduzca - https://docs.python.org
genindex.html
py-modindex.html
https://www.python.org/
#
whatsnew/3.8.html
whatsnew/index.html
tutorial/index.html
library/index.html
reference/index.html
using/index.html
howto/index.html
installing/index.html
distributing/index.html
extending/index.html
c-api/index.html
faq/index.html
py-modindex.html
genindex.html
glossary.html
search.html
contents.html
bugs.html
https://devguide.python.org/docquality/#helping-with-documentation
about.html
license.html
copyright.html
download.html
https://docs.python.org/3.9/
https://docs.python.org/3.8/
https://docs.python.org/3.7/
https://docs.python.org/3.6/
https://docs.python.org/3.5/
https://docs.python.org/2.7/
https://www.python.org/doc/versions/
https://www.python.org/dev/peps/
https://wiki.python.org/moin/BeginnersGuide
https://wiki.python.org/moin/PythonBooks
https://www.python.org/doc/av/
https://devguide.python.org/
genindex.html
py-modindex.html
https://www.python.org/
#
copyright.html
https://www.python.org/psf/donations/
https://docs.python.org/3/bugs.html
https://www.sphinx-doc.org/
```

Esta lista es mucho más larga porque algunas de las etiquetas de anclaje son rutas relativas (e.g., tutorial/index.html) o referencias dentro de la página (p. ej., '#') que no incluyen "http://" o "https://", lo cual era un requerimiento en nuestra expresión regular.

También puedes utilizar BeautifulSoup para extraer varias partes de cada etiqueta de este modo:

```
# Para ejecutar este programa descarga BeautifulSoup
# https://pypi.python.org/pypi/beautifulsoup4

# O descarga el archivo
# http://www.py4e.com/code3/bs4.zip
# y descomprimelo en el mismo directorio que este archivo

from urllib.request import urlopen
from bs4 import BeautifulSoup
import ssl

# Ignorar errores de certificado SSL
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

url = input('Introduzca - ')
html = urlopen(url, context=ctx).read()
sopa = BeautifulSoup(html, "html.parser")

# Obtener todas las etiquetas de anclaje
etiquetas = sopa('a')
for etiqueta in etiquetas:
    # Look at the parts of a tag
    print('ETIQUETA:', etiqueta)
    print('URL:', etiqueta.get('href', None))
    print('Contenido:', etiqueta.contents[0])
    print('Atributos:', etiqueta.attrs)

# Code: http://www.py4e.com/code3/urllink2.py
```

```
python urllink2.py
Introduzca - http://www.dr-chuck.com/page1.htm
ETIQUETA: <a href="http://www.dr-chuck.com/page2.htm">
Second Page</a>
URL: http://www.dr-chuck.com/page2.htm
Contenido:
Second Page
Atributos: {'href': 'http://www.dr-chuck.com/page2.htm'}
```

`html.parser` es el analizador de HTML incluido en la librería estándar de Python 3. Para más información acerca de otros analizadores de HTML, lee:

http://www.crummy.com/software/BeautifulSoup/bs4/doc/#installing-a-parser

Estos ejemplo tan sólo muestran un poco de la potencia de BeautifulSoup cuando se trata de analizar HTML.

12.9 Sección extra para usuarios de Unix / Linux

Si tienes una computadora Linux, Unix, o Macintosh, probablemente tienes comandos nativos de tu sistema operativo para obtener tanto archivos de texto plano como archivos binarios utilizando los procolos HTTP o de Transferencia de Archivos (File Transfer - FTP). Uno de esos comandos es `curl`:

```
$ curl -O http://www.py4e.com/cover.jpg
```

El comando `curl` es una abreviación de "copiar URL" y por esa razón los dos ejemplos vistos anteriormente para obtener archivos binarios con `urllib` son astutamente llamados `curl1.py` y `curl2.py` en www.py4e.com/code3 debido a que ellos implementan una funcionalidad similar a la del comando `curl`. Existe también un programa de ejemplo `curl3.py` que realiza la misma tarea de forma un poco más eficiente, en caso de que quieras usar de verdad este diseño en algún programa que estés escribiendo.

Un segundo comando que funciona de forma similar es `wget`:

```
$ wget http://www.py4e.com/cover.jpg
```

Ambos comandos hacen que obtener páginas web y archivos remotos se vuelva una tarea fácil.

12.10 Glosario

BeautifulSoup Una librería de Python para analizar documentos HTML y extraer datos de ellos, que compensa la mayoría de las imperfecciones que los navegadores HTML normalmente ignoran. Puedes descargar el código de BeautifulSoup desde www.crummy.com.

puerto Un número que generalmente indica qué aplicación estás contactando cuando realizas una conexión con un socket en un servidor. Por ejemplo, el tráfico web normalmente usa el puerto 80, mientras que el tráfico del correo electrónico usa el puerto 25.

rascado Cuando un programa simula ser un navegador web y recupera una página web, para luego realizar una búsqueda en su contenido. A menudo los programas siguen los enlaces en una página para encontrar la siguiente, de modo que pueden atravesar una red de páginas o una red social.

rastrear La acción de un motor de búsqueda web que consiste en recuperar una página y luego todas las páginas enlazadas por ella, continuando así sucesivamente hasta que tienen casi todas las páginas de Internet, que usan a continuación para construir su índice de búsqueda.

socket Una conexión de red entre dos aplicaciones, en la cual dichas aplicaciones pueden enviar y recibir datos en ambas direcciones.

12.11 Ejercicios

Ejercicio 1: Cambia el programa del socket `socket1.py` para que le pida al usuario la URL, de modo que pueda leer cualquier página web. Puedes usar `split('/')` para dividir la URL en las partes que la componen, de modo que puedas extraer el nombre del host para la llamada a `connect` del socket. Añade comprobación de errores utilizando `try` y `except` para contemplar la posibilidad de que el usuario introduzca una URL mal formada o inexistente.

Ejercicio 2: Cambia el programa del socket para que cuente el número de caracteres que ha recibido y se detenga, con un texto en pantalla, después de que se hayan mostrado 3000 caracteres. El programa debe recuperar el documento completo y contar el número total de caracteres, mostrando ese total al final del documento.

Ejercicio 3: Utiliza `urllib` para rehacer el ejercicio anterior de modo que (1) reciba el documento de una URL, (2) muestre hasta 3000 caracteres, y (3) cuente la cantidad total de caracteres en el documento. No te preocupes de las cabeceras en este ejercicio, simplemente muesta los primeros 3000 caracteres del contenido del documento.

Ejercicio 4: Cambia el programa `urllinks.py` para extraer y contar las etiquetas de párrafo (p) del documento HTML recuperado y mostrar el total de párrafos como salida del programa. No muestres el texto de los párrafos, sólo cuéntalos. Prueba el programa en varias páginas web pequeñas, y también en otras más grandes.

Ejercicio 5: (Avanzado) Cambia el programa del socket de modo que solamente muestra datos después de que se haya recibido la cabecera y la línea en blanco. Recuerda que `recv` recibe caracteres (saltos de línea incluidos), no líneas.

Chapter 13

Uso de Servicios Web

Una vez que recuperar documentos a través de HTTP y analizarlos usando programas se convirtió en algo sencillo, no se tardó mucho en desarrollar un modelo consistente en la producción de documentos específicamente diseñados para ser consumidos por otros programas (es decir, no únicamente HTML para ser mostrado en un navegador).

Existen dos formatos habituales que se usan para el intercambio de datos a través de la web. El "eXtensible Markup Language" (lenguaje extensible de marcas), o XML, ha sido utilizado durante mucho tiempo, y es el más adecuado para intercambiar datos del tipo documento. Cuando los programas simplemente quieren intercambiar diccionarios, listas u otra información interna, usan "JavaScript Object Notation", o JSON (Notación de Objetos Javascript; consulta www.json.org). Nosotros examinaremos ambos formatos.

13.1 eXtensible Markup Language - XML

XML tiene un aspecto similar a HTML, pero más estructurado. Este es un ejemplo de un documento XML:

```
<person>
  <name>Chuck</name>
  <phone type="intl">
    +1 734 303 4456
  </phone>
  <email hide="yes" />
</person>
```

Cada par de etiquetas de apertura (p. ej., ‘’) y de cierre (p. ej., ‘’) representan un *elemento* o *nodo* con el mismo nombre de la etiqueta (p. ej., ‘person’). Cada elemento puede contener texto, atributos (p. ej., ‘hide’) y otros elementos anidados. Si un elemento XML está vacío (es decir, no tiene contenido), puede representarse con una etiqueta auto-cerrada (p. ej., ‘’).

A menudo resulta útil pensar en un documento XML como en la estructura de un árbol, donde hay una etiqueta superior (en este caso 'person'), y otras etiquetas como 'phone' que se muestran como *hijas* de sus nodos *padres*.

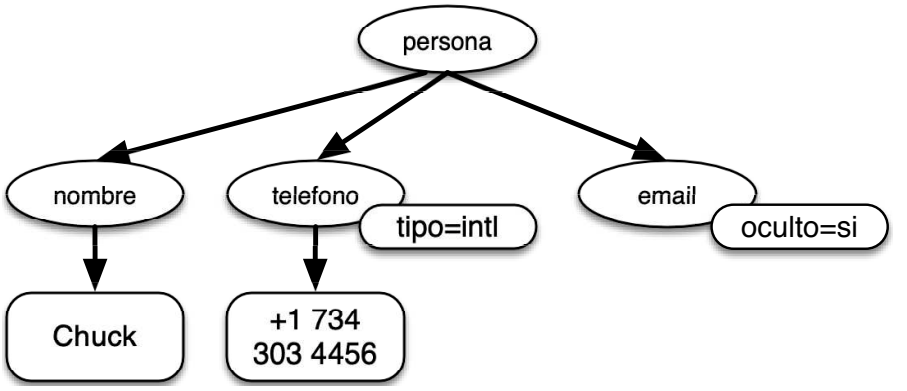

Figure 13.1: A Tree Representation of XML

13.2 Análisis de XML

He aquí una aplicación sencilla que analiza un archivo XML y extrae algunos elementos de él:

```
import xml.etree.ElementTree as ET

data = '''
<person>
  <name>Chuck</name>
  <phone type="intl">
    +1 734 303 4456
  </phone>
  <email hide="yes" />
</person>'''

tree = ET.fromstring(data)
print('Name:', tree.find('name').text)
print('Attr:', tree.find('email').get('hide'))

# Code: http://www.py4e.com/code3/xml1.py
```

Tanto la triple comilla simple ('''') como la triple comilla doble (""") permiten la creación de cadenas que abarquen varias líneas.

La llamada a 'fromstring' convierte la representación de cadena del XML en un "árbol" de nodos XML. Una vez tenemos el XML como un árbol, disponemos de una serie de métodos que podemos llamar para extraer porciones de datos de ese XML. La función 'find' busca a través del árbol XML y recupera el nodo que coincide con la etiqueta especificada.

```
Name: Chuck
Attr: yes
```

El usar un analizador de XML como 'ElementTree' tiene la ventaja de que, a pesar de que el XML de este ejemplo es bastante sencillo, resulta que hay muchas reglas relativas a la validez del XML, y el uso de 'ElementTree' nos permite extraer datos del XML sin preocuparnos por esas reglas de sintaxis.

13.3 Desplazamiento a través de los nodos

A menudo el XML tiene múltiples nodos y tenemos que escribir un bucle para procesarlos todos. En el programa siguiente, usamos un bucle para recorrer todos los nodos 'user':

```
import xml.etree.ElementTree as ET

input = '''
<stuff>
  <users>
    <user x="2">
      <id>001</id>
      <name>Chuck</name>
    </user>
    <user x="7">
      <id>009</id>
      <name>Brent</name>
    </user>
  </users>
</stuff>'''

stuff = ET.fromstring(input)
lst = stuff.findall('users/user')
print('User count:', len(lst))

for item in lst:
    print('Name', item.find('name').text)
    print('Id', item.find('id').text)
    print('Attribute', item.get('x'))

# Code: http://www.py4e.com/code3/xml2.py
```

El método 'findall' devuelve una lista de subárboles que representan las estructuras 'user' del árbol XML. A continuación podemos escribir un bucle 'for' que busque en cada uno de los nodos usuario, e imprima el texto de los elementos 'name' e 'id', además del atributo 'x' de cada nodo usuario.

```
User count: 2
Name Chuck
```

```
Id 001
Attribute 2
Name Brent
Id 009
Attribute 7
```

Es importante incluir todos los elementos base en la declaración de 'findall' exceptuando aquel que se encuentra en el nivel superior (p. ej., 'users/user'). De lo contrario, Python no encontrará ninguno de los nodos que buscamos.

```
import xml.etree.ElementTree as ET

input = '''
<stuff>
  <users>
    <user x="2">
      <id>001</id>
      <name>Chuck</name>
    </user>
    <user x="7">
      <id>009</id>
      <name>Brent</name>
    </user>
  </users>
</stuff>'''

stuff = ET.fromstring(input)

lst = stuff.findall('users/user')
print('User count:', len(lst))

lst2 = stuff.findall('user')
print('User count:', len(lst2))
```

'lst' almacena todos los elementos 'user' anidados dentro de su base 'users'. 'lst2' busca los elementos 'user' que no se encuentren anidados dentro del elemento de nivel superior 'stuff' donde no hay ninguno.

```
User count: 2
User count: 0
```

13.4 JavaScript Object Notation - JSON

El formato JSON se inspiró en el formato de objetos y arrays que se usa en el lenguaje JavaScript. Pero como Python se inventó antes que JavaScript, la sintaxis usada en Python para los diccionarios y listas influyeron la sintaxis de JSON. De modo que el formato del JSON es casi idéntico a la combinación de listas y diccionarios de Python.

He aquí una codificación JSON que es más o menos equivalente al XML del ejemplo anterior:

```
{
  "name" : "Chuck",
  "phone" : {
    "type" : "intl",
    "number" : "+1 734 303 4456"
   },
   "email" : {
     "hide" : "yes"
   }
}
```

Si te fijas, encontrarás ciertas diferencias. Primero, en XML se pueden añadir atributos como "intl" a la etiqueta "phone". En JSON, simplemente tenemos parejas clave-valor. Además, la etiqueta "person" de XML ha desaparecido, reemplazada por un conjunto de llaves exteriores.

En general, las estructuras JSON son más simples que las de XML, debido a que JSON tiene menos capacidades. Pero JSON tiene la ventaja de que mapea *directamente* hacia una combinación de diccionarios y listas. Y, dado que casi todos los lenguajes de programación tienen algo equivalente a los diccionarios y listas de Python, JSON es un formato muy intuitivo para que dos programas que vayan a cooperar intercambien datos.

JSON se está convirtiendo rápidamente en el formato elegido para casi todos los intercambios de datos entre aplicaciones, debido a su relativa simplicidad comparado con XML.

13.5 Análisis de JSON

El JSON se construye anidando diccionarios y listas según se necesite. En este ejemplo, vamos a representar una lista de usuarios en la que cada usuario es un conjunto de parejas clave-valor (es decir, un diccionario). De modo que tendremos una lista de diccionarios.

En el programa siguiente, usaremos la librería integrada 'json' para analizar el JSON y leer los datos. Compáralo cuidadosamente con los datos y código XML equivalentes que usamos antes. El JSON tiene menos detalles, de modo que podemos saber de antemano que vamos a obtener una lista y que la lista es de usuarios y además que cada usuario es un conjunto de parejas clave-valor. El JSON es más conciso (una ventaja), pero también es menos auto-descriptivo (una desventaja).

```
import json

data = '''
[
  { "id" : "001",
    "x" : "2",
```

```
    "name" : "Chuck"
  } ,
  { "id" : "009",
    "x" : "7",
    "name" : "Brent"
  }
]'''

info = json.loads(data)
print('User count:', len(info))

for item in info:
    print('Name', item['name'])
    print('Id', item['id'])
    print('Attribute', item['x'])

# Code: http://www.py4e.com/code3/json2.py
```

Si comparas el código que extrae los datos del JSON analizado y el del XML, verás que lo que obtenemos de 'json.loads()' es una lista de Python que recorreremos con un bucle for, y cada elemento dentro de esa lista es un diccionario de Python. Una vez analizado el JSON, podemos usar el operador índice de Python para extraer los distintos fragmentos de datos de cada usuario. No tenemos que usar la librería JSON para rebuscar a través del JSON analizado, ya que los datos devueltos son sencillamente estructuras nativas de Python.

La salida de este programa es exactamente la misma que la de la versión XML anterior.

```
User count: 2
Name Chuck
Id 001
Attribute 2
Name Brent
Id 009
Attribute 7
```

En general, hay una tendencia en la industria a apartarse del XML y pasar al JSON para los servicios web. Como el JSON es más sencillo, y se mapea de forma más directa hacia estructuras de datos nativas que ya tenemos en los lenguajes de programación, el código de análisis y extracción de datos normalmente es más sencillo y directo usando JSON. Sin embargo, XML es más auto-descriptivo, y por eso hay ciertas aplicaciones en las cuales mantiene su ventaja. Por ejemplo, la mayoría de los procesadores de texto almacenan sus documentos internamente usando XML en vez de JSON.

13.6 Interfaces de programación de aplicaciones

Ahora tenemos la capacidad de intercambiar datos entre aplicaciones usando el Protocolo de Transporte de Hipertexto (HTTP), y un modo de representar estructuras

de datos complejas para poder enviar y recibir los datos entre esas aplicaciones, a través del eXtensibleMarkup Language (XML) o del JavaScript Object Notation (JSON).

El paso siguiente es empezar a definir y documentar "contratos" entre aplicaciones usando estas técnicas. El nombre habitual para estos contratos entre aplicaciones es *Interfaces de Programación de Aplicaciones* (Application Program Interfaces), o APIs. Cuando se utiliza una API, normalmente un programa crea un conjunto de *servicios* disponibles para que los usen otras aplicaciones y publica las APIs (es decir, las "reglas") que deben ser seguidas para acceder a los servicios proporcionados por el programa.

Cuando comenzamos a construir programas con funcionalidades que incluyen el acceso a servicios proporcionados por otros programas, el enfoque se denomina *Service-Oriented Architecture* (Arquitectura Orientada a Servicios), o SOA. Un enfoque SOA es aquel en el cual nuestra aplicación principal usa los servicios de otras aplicaciones. Un planteamiento no-SOA es aquel en el cual tenemos una única aplicación independiente que contiene todo el código necesario para su implementación.

Podemos encontrar multitud de ejemplos de SOAs cuando utilizamos servicios de la web. Podemos ir a un único sitio web y reservar viajes en avión, hoteles y automóviles, todo ello desde el mismo sitio. Los datos de los hoteles no están almacenados en los equipos de la compañía aérea. En vez de eso, los computadores de la aerolínea contactan con los servicios de los computadores de los hoteles, recuperan los datos de éstos, y se los presentan al usuario. Cuando el usuario acepta realizar una reserva de un hotel usando el sitio web de una aerolínea, ésta utiliza otro servicio web en los sistemas de los hoteles para realizar la reserva real. Y cuando llega el momento de cargar en tu tarjeta de crédito el importe de la transacción completa, hay todavía otros equipos diferentes involucrados en el proceso.

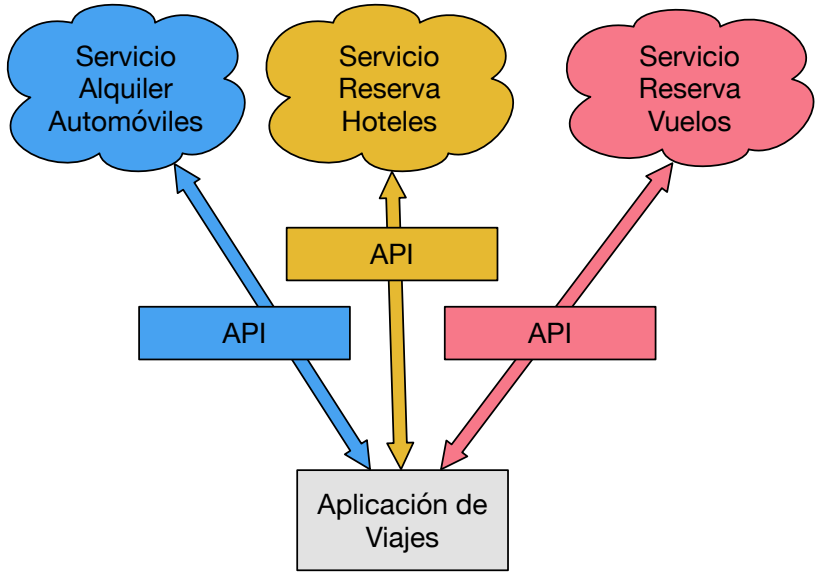

Figure 13.2: Service-oriented architecture

Una Arquitectura Orientada a Servicios tiene muchas ventajas, que incluyen: (1) siempre se mantiene una única copia de los datos (lo cual resulta particularmente importante en ciertas áreas como las reservas hoteleras, donde no queremos duplicar excesivamente la información) y (2) los propietarios de los datos pueden imponer reglas acerca del uso de esos datos. Con estas ventajas, un sistema SOA debe ser diseñado cuidadosamente para tener buen rendimiento y satisfacer las necesidades de los usuarios.

Cuando una aplicación ofrece un conjunto de servicios en su API disponibles a través de la web, los llamamos *servicios web*.

13.7 Seguridad y uso de APIs

Resulta bastante frecuente que se necesite algún tipo de "clave API" para hacer uso de una API comercial. La idea general es que ellos quieren saber quién está usando sus servicios y cuánto los utiliza cada usuario. Tal vez tienen distintos niveles (gratuitos y de pago) de servicios, o una política que limita el número de peticiones que un único usuario puede realizar durante un determinado periodo de tiempo.

En ocasiones, una vez que tienes tu clave API, tan sólo debes incluirla como parte de los datos POST, o tal vez como parámetro dentro de la URL que usas para llamar a la API.

Otras veces, el vendedor quiere aumentar la certeza sobre el origen de las peticiones, de modo que además espera que envíes mensajes firmados criptográficamente, usando claves compartidas y secretas. Una tecnología muy habitual que se utiliza para firmar peticiones en Internet se llama *OAuth*. Puedes leer más acerca del protocolo OAuth en www.oauth.net.

Afortunadamente, hay varias librerías OAuth útiles y gratuitas, de modo que te puedes ahorrar el tener que escribir una implementación OAuth desde cero leyendo las especificaciones. Estas librerías tienen distintos niveles de complejidad, así como variedad de características. El sitio web OAuth tiene información sobre varias librerías OAuth.

13.8 Glossary

API Application Programming Interface (Interfaz de Programación de Aplicaciones) - Un contrato entre aplicaciones que define las pautas de interacción entre los componentes de dos aplicaciones.

ElementTree Una librería interna de Python que se utiliza para analizar datos XML.

JSON JavaScript Object Notation (Notación de Objetos JavaScript). Un formato que permite el envío de estructuras de datos basadas en la sintaxis de los Objetos JavaScript.

SOA Service-Oriented Architecture (Arquitectura Orientada a Servicios). Cuando una aplicación está formada por componentes conectados a través de una red.

XML eXtensible Markup Language (Lenguaje de Marcas eXtensible). Un formato que permite el envío de datos estructurados.

13.9 Aplicación Nº 1: Servicio web de geocodificación de Google

Google tiene un excelente servicio web que nos permite hacer uso de su enorme base de datos de información geográfica. Podemos enviar una cadena de búsqueda geográfica, como "Ann Arbor, MI" a su API de geocodificación y conseguir que Google nos devuelva su mejor suposición sobre donde podría estar nuestra cadena de búsqueda en un mapa, además de los puntos de referencia en los alrededores.

El servicio de geocodificación es gratuito, pero limitado, de modo que no se puede hacer un uso intensivo de esta API en una aplicación comercial. Pero si tienes ciertos datos estadísticos en los cuales un usuario final ha introducido una localización en formato libre en un cuadro de texto, puedes utilizar esta API para limpiar esos datos de forma bastante efectiva.

Cuando se usa una API libre, como la API de geocodificación de Google, se debe ser respetuoso con el uso de los recursos. Si hay demasiada gente que abusa del servicio, Google puede interrumpir o restringir significativamente su uso gratuito.

Puedes leer la documentación online de este servicio, pero es bastante sencillo y puedes incluso probarlo desde un navegador, simplemente tecleando la siguiente URL en él:

http://maps.googleapis.com/maps/api/geocode/json?address=Ann+Arbor%2C+MI

Asegúrate de limpiar la URL y eliminar cualquier espacio de ella antes de pegarla en tu navegador.

La siguiente es una aplicación sencilla que pide al usuario una cadena de búsqueda, llama a la API de geocodificación de Google y extrae información del JSON que nos devuelve.

```
import urllib.request, urllib.parse, urllib.error
import json
import ssl

api_key = False
# If you have a Google Places API key, enter it here
# api_key = 'AIzaSy___IDByT70'
# https://developers.google.com/maps/documentation/geocoding/intro

if api_key is False:
    api_key = 42
    serviceurl = 'http://py4e-data.dr-chuck.net/json?'
else :
    serviceurl = 'https://maps.googleapis.com/maps/api/geocode/json?'

# Ignore SSL certificate errors
```

```
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

while True:
    address = input('Enter location: ')
    if len(address) < 1: break

    parms = dict()
    parms['address'] = address
    if api_key is not False: parms['key'] = api_key
    url = serviceurl + urllib.parse.urlencode(parms)

    print('Retrieving', url)
    uh = urllib.request.urlopen(url, context=ctx)
    data = uh.read().decode()
    print('Retrieved', len(data), 'characters')

    try:
        js = json.loads(data)
    except:
        js = None

    if not js or 'status' not in js or js['status'] != 'OK':
        print('==== Failure To Retrieve ====')
        print(data)
        continue

    print(json.dumps(js, indent=4))

    lat = js['results'][0]['geometry']['location']['lat']
    lng = js['results'][0]['geometry']['location']['lng']
    print('lat', lat, 'lng', lng)
    location = js['results'][0]['formatted_address']
    print(location)

# Code: http://www.py4e.com/code3/geojson.py
```

El programa toma la cadena de búsqueda y construye una URL codificándola como parámetro dentro de ella, utilizando luego 'urllib' para recuperar el texto de la API de geocodificación de Google. A diferencia de una página web estática, los datos que obtengamos dependerán de los parámetros que enviemos y de los datos geográficos almacenados en los servidores de Google.

Una vez recuperados los datos JSON, los analizamos con la librería 'json' y revisamos para asegurarnos de que hemos recibido datos válidos. Finalmente, extraemos la información que buscábamos.

La salida del programa es la siguiente (parte del JSON recibido ha sido eliminado):

```
$ python3 geojson.py
```

```
Enter location: Ann Arbor, MI
Retrieving http://py4e-data.dr-chuck.net/json?address=Ann+Arbor%2C+MI&key=42
Retrieved 1736 characters
```


```
{
    "results": [
        {
            "address_components": [
                {
                    "long_name": "Ann Arbor",
                    "short_name": "Ann Arbor",
                    "types": [
                        "locality",
                        "political"
                    ]
                },
                {
                    "long_name": "Washtenaw County",
                    "short_name": "Washtenaw County",
                    "types": [
                        "administrative_area_level_2",
                        "political"
                    ]
                },
                {
                    "long_name": "Michigan",
                    "short_name": "MI",
                    "types": [
                        "administrative_area_level_1",
                        "political"
                    ]
                },
                {
                    "long_name": "United States",
                    "short_name": "US",
                    "types": [
                        "country",
                        "political"
                    ]
                }
            ],
            "formatted_address": "Ann Arbor, MI, USA",
            "geometry": {
                "bounds": {
                    "northeast": {
                        "lat": 42.3239728,
                        "lng": -83.6758069
                    },
                    "southwest": {
                        "lat": 42.222668,
                        "lng": -83.799572
```

```
                    }
                },
                "location": {
                    "lat": 42.2808256,
                    "lng": -83.7430378
                },
                "location_type": "APPROXIMATE",
                "viewport": {
                    "northeast": {
                        "lat": 42.3239728,
                        "lng": -83.6758069
                    },
                    "southwest": {
                        "lat": 42.222668,
                        "lng": -83.799572
                    }
                }
            },
            "place_id": "ChIJMx9D1A2wPIgR4rXIhkb5Cds",
            "types": [
                "locality",
                "political"
            ]
        }
    ],
    "status": "OK"
}
lat 42.2808256 lng -83.7430378
Ann Arbor, MI, USA

Enter location:
```


Puedes descargar www.py4e.com/code3/geoxml.py para revisar las variantes JSON y XML de la API de geocodificación de Google.

Ejercicio 1: Modifica geojson.py o geoxml.py para imprimir en pantalla el código de país de dos caracteres de los datos recuperados. Añade comprobación de errores, de modo que tu programa no rastree los datos si el código del país no está presente. Una vez que lo tengas funcionando, busca "Océano Atlántico" y asegúrate de que es capaz de gestionar ubicaciones que no estén dentro de ningún país.

13.10 Aplicación 2: Twitter

A medida que la API de Twitter se vuelve más valiosa, Twitter pasó de una API pública y abierta a una que requiere el uso de firmas OAuth en cada solicitud.

Para el programa de ejemplo siguiente, descarga los archivos *twurl.py*, *hidden.py*, *oauth.py* y *twitter1.py* desde www.py4e.com/code y ponlos todos en una misma carpeta en tu equipo.

Para usar estos programas debes tener una cuenta de Twitter, y autorizar a tu código Python como aplicación permitida, estableciendo diversos parámetros (key, secret, token y token secret). Luego deberás editar el archivo *hidden.py* y colocar esas cuatro cadenas en las variables apropiadas dentro del archivo:

```
# Keep this file separate

# https://apps.twitter.com/
# Create new App and get the four strings

def oauth():
    return {"consumer_key": "h7Lu...Ng",
            "consumer_secret": "dNKenAC3New...mmn7Q",
            "token_key": "10185562-eibxCp9n2...P4GEQQOSGI",
            "token_secret": "H0ycCFemmC4wyf1...qoIpBo"}

# Code: http://www.py4e.com/code3/hidden.py
```

Se puede acceder al servicio web de Twitter mediante una URL como ésta:

https://api.twitter.com/1.1/statuses/user_timeline.json

Pero una vez añadida la información de seguridad, la URL se parecerá más a esto:

```
https://api.twitter.com/1.1/statuses/user_timeline.json?count=2 &oauth_version=1.0&oauth_token=101...SGI&screen_name=drch
```

Puedes leer la especificación OAuth si quieres saber más acerca del significado de los distintos parámetros que hemos añadido para cumplir con los requerimientos de seguridad de OAuth.

Para los programas que ejecutamos con Twitter, ocultamos toda la complejidad dentro de los archivos *oauth.py* y *twurl.py*. Simplemente ajustamos los parámetros secretos en *hidden.py*, enviamos la URL deseada a la función *twurl.augment()*, y el código de la librería añade todos los parámetros necesarios a la URL.

Este programa recupera la línea de tiempo de un usuario de Twitter concreto y nos la devuelve en formato JSON como una cadena. Vamos a imprimir simplemente los primeros 250 caracteres de esa cadena:

```
import urllib.request, urllib.parse, urllib.error
import twurl
import ssl

# https://apps.twitter.com/
# Create App and get the four strings, put them in hidden.py

TWITTER_URL = 'https://api.twitter.com/1.1/statuses/user_timeline.json'

# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE
```

```
while True:
    print('')
    acct = input('Enter Twitter Account:')
    if (len(acct) < 1): break
    url = twurl.augment(TWITTER_URL,
                        {'screen_name': acct, 'count': '2'})
    print('Retrieving', url)
    connection = urllib.request.urlopen(url, context=ctx)
    data = connection.read().decode()
    print(data[:250])
    headers = dict(connection.getheaders())
    # print headers
    print('Remaining', headers['x-rate-limit-remaining'])

# Code: http://www.py4e.com/code3/twitter1.py
```

Cuando el programa se ejecuta, produce la salida siguiente:

```
Enter Twitter Account:drchuck
Retrieving https://api.twitter.com/1.1/ ...
[{"created_at":"Sat Sep 28 17:30:25 +0000 2013","
id":384007200990982144,"id_str":"384007200990982144",
"text":"RT @fixpert: See how the Dutch handle traffic
intersections: http:\/\/t.co\/tIiVWtEhj4\n#brilliant",
"source":"web","truncated":false,"in_rep
Remaining 178

Enter Twitter Account:fixpert
Retrieving https://api.twitter.com/1.1/ ...
[{"created_at":"Sat Sep 28 18:03:56 +0000 2013",
"id":384015634108919808,"id_str":"384015634108919808",
"text":"3 months after my freak bocce ball accident,
my wedding ring fits again! :)\n\nhttps:\/\/t.co\/2XmHPx7kgX",
"source":"web","truncated":false,
Remaining 177

Enter Twitter Account:
```

Junto con los datos de la línea del tiempo, Twitter también devuelve metadatos sobre la petición en las cabeceras de respuesta HTTP. Una cabecera en particular, 'x-rate-limit-remaining', nos informa sobre cuántas peticiones podremos hacer antes de que seamos bloqueados por un corto periodo de tiempo. Puedes ver que cada vez que realizamos una petición a la API nuestros intentos restantes van disminuyendo.

En el ejemplo siguiente, recuperamos los amigos de un usuario en Twitter, analizamos el JSON devuelto y extraemos parte de la información sobre esos amigos. Después de analizar el JSON e "imprimirlo bonito", realizamos un volcado completo con un indentado de cuatro caracteres, que nos permite estudiar minuciosamente los datos en caso de que queramos extraer más campos.

```
import urllib.request, urllib.parse, urllib.error
import twurl
```

```
import json
import ssl

# https://apps.twitter.com/
# Create App and get the four strings, put them in hidden.py

TWITTER_URL = 'https://api.twitter.com/1.1/friends/list.json'

# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

while True:
    print('')
    acct = input('Enter Twitter Account:')
    if (len(acct) < 1): break
    url = twurl.augment(TWITTER_URL,
                        {'screen_name': acct, 'count': '5'})
    print('Retrieving', url)
    connection = urllib.request.urlopen(url, context=ctx)
    data = connection.read().decode()

    js = json.loads(data)
    print(json.dumps(js, indent=2))

    headers = dict(connection.getheaders())
    print('Remaining', headers['x-rate-limit-remaining'])

    for u in js['users']:
        print(u['screen_name'])
        if 'status' not in u:
            print('   * No status found')
            continue
        s = u['status']['text']
        print('  ', s[:50])

# Code: http://www.py4e.com/code3/twitter2.py
```

Dado que el JSON se transforma en un conjunto de listas y diccionarios de Python anidados, podemos usar una combinación del operador índice junto con bucles 'for' para movernos a través de las estructuras de datos devueltas con muy poco código de Python.

La salida del programa se parece a la siguiente (parte de los datos se han acortado para que quepan en la página):

```
Enter Twitter Account:drchuck
Retrieving https://api.twitter.com/1.1/friends ...
Remaining 14
```

```
{
  "next_cursor": 1444171224491980205,
  "users": [
    {
      "id": 662433,
      "followers_count": 28725,
      "status": {
        "text": "@jazzychad I just bought one .__.",
        "created_at": "Fri Sep 20 08:36:34 +0000 2013",
        "retweeted": false,
      },
      "location": "San Francisco, California",
      "screen_name": "leahculver",
      "name": "Leah Culver",
    },
    {
      "id": 40426722,
      "followers_count": 2635,
      "status": {
        "text": "RT @WSJ: Big employers like Google ...",
        "created_at": "Sat Sep 28 19:36:37 +0000 2013",
      },
      "location": "Victoria Canada",
      "screen_name": "_valeriei",
      "name": "Valerie Irvine",
    }
  ],
 "next_cursor_str": "1444171224491980205"
}
```


```
leahculver
   @jazzychad I just bought one .__.
_valeriei
   RT @WSJ: Big employers like Google, AT&amp;amp;T are h
ericbollens
   RT @lukew: sneak peek: my LONG take on the good &a
halherzog
   Learning Objects is 10. We had a cake with the LO,
scweeker
   @DeviceLabDC love it! Now where so I get that "etc

Enter Twitter Account:
```

El último trozo de la salida es donde podemos ver cómo el bucle for lee los cinco "amigos" más nuevos de la cuenta de Twitter *@drchuck* e imprime el estado más reciente de cada uno de ellos. Hay muchos más datos disponibles en el JSON devuelto. Si miras la salida del programa, podrás ver que el "encuentra a los amigos" de una cuenta particular tiene una limitación de usos distinta a la del número de consultas de líneas de tiempo que está permitido realizar durante un periodo de tiempo.

Estas claves de seguridad de la API permiten a Twitter tener la certeza de que

sabe quién está usando su API de datos, y a qué nivel. El enfoque de límite de usos nos permite hacer capturas de datos sencillas, pero no crear un producto que extraiga datos de esa API millones de veces al día.

Chapter 14

Object-oriented programming

14.1 Managing larger programs

At the beginning of this book, we came up with four basic programming patterns which we use to construct programs:

- Sequential code
- Conditional code (if statements)
- Repetitive code (loops)
- Store and reuse (functions)

In later chapters, we explored simple variables as well as collection data structures like lists, tuples, and dictionaries.

As we build programs, we design data structures and write code to manipulate those data structures. There are many ways to write programs and by now, you probably have written some programs that are "not so elegant" and other programs that are "more elegant". Even though your programs may be small, you are starting to see how there is a bit of art and aesthetic to writing code.

As programs get to be millions of lines long, it becomes increasingly important to write code that is easy to understand. If you are working on a million-line program, you can never keep the entire program in your mind at the same time. We need ways to break large programs into multiple smaller pieces so that we have less to look at when solving a problem, fix a bug, or add a new feature.

In a way, object oriented programming is a way to arrange your code so that you can zoom into 50 lines of the code and understand it while ignoring the other 999,950 lines of code for the moment.

14.2 Getting started

Like many aspects of programming, it is necessary to learn the concepts of object oriented programming before you can use them effectively. You should approach this chapter as a way to learn some terms and concepts and work through a few simple examples to lay a foundation for future learning.

The key outcome of this chapter is to have a basic understanding of how objects are constructed and how they function and most importantly how we make use of the capabilities of objects that are provided to us by Python and Python libraries.

14.3 Using objects

As it turns out, we have been using objects all along in this book. Python provides us with many built-in objects. Here is some simple code where the first few lines should feel very simple and natural to you.

```
stuff = list()
stuff.append('python')
stuff.append('chuck')
stuff.sort()
print (stuff[0])
print (stuff.__getitem__(0))
print (list.__getitem__(stuff,0))

# Code: http://www.py4e.com/code3/party1.py
```

Instead of focusing on what these lines accomplish, let's look at what is really happening from the point of view of object-oriented programming. Don't worry if the following paragraphs don't make any sense the first time you read them because we have not yet defined all of these terms.

The first line *constructs* an object of type `list`, the second and third lines *call* the `append()` *method*, the fourth line calls the `sort()` method, and the fifth line *retrieves* the item at position 0.

The sixth line calls the `__getitem__()` method in the `stuff` list with a parameter of zero.

```
print (stuff.__getitem__(0))
```

The seventh line is an even more verbose way of retrieving the 0th item in the list.

```
print (list.__getitem__(stuff,0))
```

In this code, we call the `__getitem__` method in the `list` class and *pass* the list and the item we want retrieved from the list as parameters.

The last three lines of the program are equivalent, but it is more convenient to simply use the square bracket syntax to look up an item at a particular position in a list.

We can take a look at the capabilities of an object by looking at the output of the `dir()` function:

```
>>> stuff = list()
>>> dir(stuff)
['__add__', '__class__', '__contains__', '__delattr__',
'__delitem__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__',
'__gt__', '__hash__', '__iadd__', '__imul__', '__init__',
'__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__rmul__', '__setattr__',
'__setitem__', '__sizeof__', '__str__', '__subclasshook__',
'append', 'clear', 'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
>>>
```

The rest of this chapter will define all of the above terms so make sure to come back after you finish the chapter and re-read the above paragraphs to check your understanding.

14.4 Starting with programs

A program in its most basic form takes some input, does some processing, and produces some output. Our elevator conversion program demonstrates a very short but complete program showing all three of these steps.

```
usf = input('Enter the US Floor Number: ')
wf = int(usf) - 1
print('Non-US Floor Number is',wf)

# Code: http://www.py4e.com/code3/elev.py
```

If we think a bit more about this program, there is the "outside world" and the program. The input and output aspects are where the program interacts with the outside world. Within the program we have code and data to accomplish the task the program is designed to solve.

One way to think about object-oriented programming is that it separates our program into multiple "zones." Each zone contains some code and data (like a program) and has well defined interactions with the outside world and the other zones within the program.

If we look back at the link extraction application where we used the BeautifulSoup library, we can see a program that is constructed by connecting different objects together to accomplish a task:

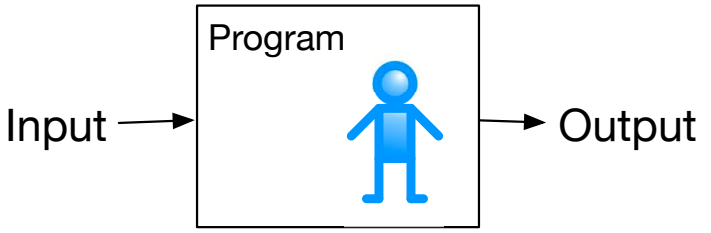

Figure 14.1: A Program

```
# Para ejecutar este programa descarga BeautifulSoup
# https://pypi.python.org/pypi/beautifulsoup4

# O descarga el archivo
# http://www.py4e.com/code3/bs4.zip
# y descomprimelo en el mismo directorio que este archivo

import urllib.request, urllib.parse, urllib.error
from bs4 import BeautifulSoup
import ssl

# Ignorar errores de certificado SSL
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

url = input('Introduzca - ')
html = urllib.request.urlopen(url, context=ctx).read()
sopa = BeautifulSoup(html, 'html.parser')

# Recuperar todas las etiquetas de anclaje
etiquetas = sopa('a')
for etiqueta in etiquetas:
    print(etiqueta.get('href', None))

# Code: http://www.py4e.com/code3/urllinks.py
```

We read the URL into a string and then pass that into `urllib` to retrieve the data from the web. The `urllib` library uses the `socket` library to make the actual network connection to retrieve the data. We take the string that `urllib` returns and hand it to BeautifulSoup for parsing. BeautifulSoup makes use of the object `html.parser`[1] and returns an object. We call the `tags()` method on the returned object that returns a dictionary of tag objects. We loop through the tags and call the `get()` method for each tag to print out the `href` attribute.

We can draw a picture of this program and how the objects work together.

The key here is not to understand perfectly how this program works but to see how we build a network of interacting objects and orchestrate the movement of

[1] https://docs.python.org/3/library/html.parser.html

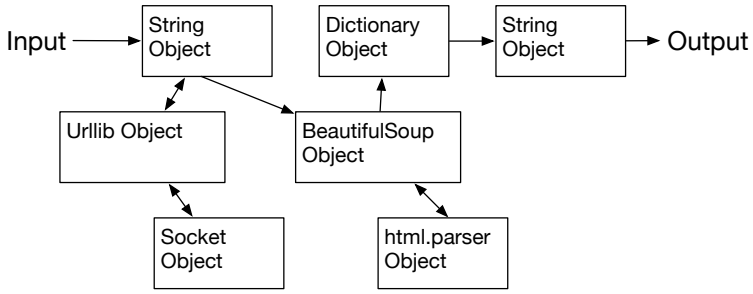

Figure 14.2: A Program as Network of Objects

information between the objects to create a program. It is also important to note that when you looked at that program several chapters back, you could fully understand what was going on in the program without even realizing that the program was "orchestrating the movement of data between objects." It was just lines of code that got the job done.

14.5 Subdividing a problem

One of the advantages of the object-oriented approach is that it can hide complexity. For example, while we need to know how to use the `urllib` and BeautifulSoup code, we do not need to know how those libraries work internally. This allows us to focus on the part of the problem we need to solve and ignore the other parts of the program.

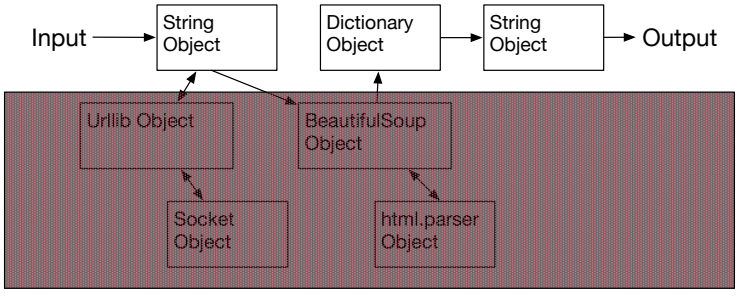

Figure 14.3: Ignoring Detail When Using an Object

This ability to focus exclusively on the part of a program that we care about and ignore the rest is also helpful to the developers of the objects that we use. For example, the programmers developing BeautifulSoup do not need to know or care about how we retrieve our HTML page, what parts we want to read, or what we plan to do with the data we extract from the web page.

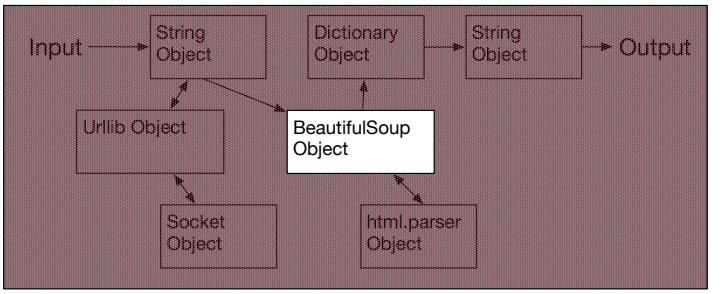

Figure 14.4: Ignoring Detail When Building an Object

14.6 Our first Python object

At a basic level, an object is simply some code plus data structures that are smaller than a whole program. Defining a function allows us to store a bit of code and give it a name and then later invoke that code using the name of the function.

An object can contain a number of functions (which we call *methods*) as well as data that is used by those functions. We call data items that are part of the object *attributes*.

We use the `class` keyword to define the data and code that will make up each of the objects. The class keyword includes the name of the class and begins an indented block of code where we include the attributes (data) and methods (code).

```
class PartyAnimal:
   x = 0

   def party(self) :
     self.x = self.x + 1
     print("So far",self.x)

an = PartyAnimal()
an.party()
an.party()
an.party()
PartyAnimal.party(an)

# Code: http://www.py4e.com/code3/party2.py
```

Each method looks like a function, starting with the `def` keyword and consisting of an indented block of code. This object has one attribute (`x`) and one method (`party`). The methods have a special first parameter that we name by convention `self`.

Just as the `def` keyword does not cause function code to be executed, the `class` keyword does not create an object. Instead, the `class` keyword defines a template indicating what data and code will be contained in each object of type `PartyAnimal`. The class is like a cookie cutter and the objects created using the class are the

cookies[2]. You don't put frosting on the cookie cutter; you put frosting on the cookies, and you can put different frosting on each cookie.

Figure 14.5: A Class and Two Objects

If we continue through this sample program, we see the first executable line of code:

```
an = PartyAnimal()
```

This is where we instruct Python to construct (i.e., create) an *object* or *instance* of the class `PartyAnimal`. It looks like a function call to the class itself. Python constructs the object with the right data and methods and returns the object which is then assigned to the variable `an`. In a way this is quite similar to the following line which we have been using all along:

```
counts = dict()
```

Here we instruct Python to construct an object using the `dict` template (already present in Python), return the instance of dictionary, and assign it to the variable `counts`.

When the `PartyAnimal` class is used to construct an object, the variable `an` is used to point to that object. We use `an` to access the code and data for that particular instance of the `PartyAnimal` class.

Each Partyanimal object/instance contains within it a variable `x` and a method/function named `party`. We call the `party` method in this line:

```
an.party()
```

When the `party` method is called, the first parameter (which we call by convention `self`) points to the particular instance of the PartyAnimal object that `party` is called from. Within the `party` method, we see the line:

```
self.x = self.x + 1
```

[2]Cookie image copyright CC-BY https://www.flickr.com/photos/dinnerseries/23570475099

This syntax using the *dot* operator is saying 'the x within self.' Each time `party()` is called, the internal `x` value is incremented by 1 and the value is printed out.

The following line is another way to call the `party` method within the `an` object:

```
PartyAnimal.party(an)
```

In this variation, we access the code from within the class and explicitly pass the object pointer `an` as the first parameter (i.e., `self` within the method). You can think of `an.party()` as shorthand for the above line.

When the program executes, it produces the following output:

```
So far 1
So far 2
So far 3
So far 4
```

The object is constructed, and the `party` method is called four times, both incrementing and printing the value for `x` within the `an` object.

14.7 Classes as types

As we have seen, in Python all variables have a type. We can use the built-in `dir` function to examine the capabilities of a variable. We can also use `type` and `dir` with the classes that we create.

```
class PartyAnimal:
   x = 0

   def party(self) :
     self.x = self.x + 1
     print("So far",self.x)

an = PartyAnimal()
print ("Type", type(an))
print ("Dir ", dir(an))
print ("Type", type(an.x))
print ("Type", type(an.party))

# Code: http://www.py4e.com/code3/party3.py
```

When this program executes, it produces the following output:

```
Type <class '__main__.PartyAnimal'>
Dir  ['__class__', '__delattr__', ...
'__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'party', 'x']
Type <class 'int'>
Type <class 'method'>
```

You can see that using the `class` keyword, we have created a new type. From the `dir` output, you can see both the `x` integer attribute and the `party` method are available in the object.

14.8 Object lifecycle

In the previous examples, we define a class (template), use that class to create an instance of that class (object), and then use the instance. When the program finishes, all of the variables are discarded. Usually, we don't think much about the creation and destruction of variables, but often as our objects become more complex, we need to take some action within the object to set things up as the object is constructed and possibly clean things up as the object is discarded.

If we want our object to be aware of these moments of construction and destruction, we add specially named methods to our object:

```
class PartyAnimal:
   x = 0

   def __init__(self):
     print('I am constructed')

   def party(self) :
     self.x = self.x + 1
     print('So far',self.x)

   def __del__(self):
     print('I am destructed', self.x)

an = PartyAnimal()
an.party()
an.party()
an = 42
print('an contains',an)

# Code: http://www.py4e.com/code3/party4.py
```

When this program executes, it produces the following output:

```
I am constructed
So far 1
So far 2
I am destructed 2
an contains 42
```

As Python constructs our object, it calls our `__init__` method to give us a chance to set up some default or initial values for the object. When Python encounters the line:

```
an = 42
```

It actually "thows our object away" so it can reuse the `an` variable to store the value `42`. Just at the moment when our `an` object is being "destroyed" our destructor code (`__del__`) is called. We cannot stop our variable from being destroyed, but we can do any necessary cleanup right before our object no longer exists.

When developing objects, it is quite common to add a constructor to an object to set up initial values for the object. It is relatively rare to need a destructor for an object.

14.9 Multiple instances

So far, we have defined a class, constructed a single object, used that object, and then thrown the object away. However, the real power in object-oriented programming happens when we construct multiple instances of our class.

When we construct multiple objects from our class, we might want to set up different initial values for each of the objects. We can pass data to the constructors to give each object a different initial value:

```
class PartyAnimal:
   x = 0
   name = ''
   def __init__(self, nam):
     self.name = nam
     print(self.name,'constructed')

   def party(self) :
     self.x = self.x + 1
     print(self.name,'party count',self.x)

s = PartyAnimal('Sally')
j = PartyAnimal('Jim')

s.party()
j.party()
s.party()

# Code: http://www.py4e.com/code3/party5.py
```

The constructor has both a `self` parameter that points to the object instance and additional parameters that are passed into the constructor as the object is constructed:

```
s = PartyAnimal('Sally')
```

Within the constructor, the second line copies the parameter (`nam`) that is passed into the `name` attribute within the object instance.

```
self.name = nam
```

The output of the program shows that each of the objects (`s` and `j`) contain their own independent copies of `x` and `nam`:

```
Sally constructed
Sally party count 1
Jim constructed
Jim party count 1
Sally party count 2
```

14.10 Inheritance

Another powerful feature of object-oriented programming is the ability to create a new class by extending an existing class. When extending a class, we call the original class the *parent class* and the new class the *child class*.

For this example, we move our `PartyAnimal` class into its own file. Then, we can 'import' the `PartyAnimal` class in a new file and extend it, as follows:

```
from party import PartyAnimal

class CricketFan(PartyAnimal):
   points = 0
   def six(self):
      self.points = self.points + 6
      self.party()
      print(self.name,"points",self.points)

s = PartyAnimal("Sally")
s.party()
j = CricketFan("Jim")
j.party()
j.six()
print(dir(j))
```

 # Code: http://www.py4e.com/code3/party6.py

When we define the `CricketFan` class, we indicate that we are extending the `PartyAnimal` class. This means that all of the variables (`x`) and methods (`party`) from the `PartyAnimal` class are *inherited* by the `CricketFan` class. For example, within the `six` method in the `CricketFan` class, we call the `party` method from the `PartyAnimal` class.

As the program executes, we create `s` and `j` as independent instances of `PartyAnimal` and `CricketFan`. The `j` object has additional capabilities beyond the `s` object.

```
Sally constructed
```

```
Sally party count 1
Jim constructed
Jim party count 1
Jim party count 2
Jim points 6
['__class__', '__delattr__', ... '__weakref__',
'name', 'party', 'points', 'six', 'x']
```

In the `dir` output for the `j` object (instance of the `CricketFan` class), we see that it has the attributes and methods of the parent class, as well as the attributes and methods that were added when the class was extended to create the `CricketFan` class.

14.11 Summary

This is a very quick introduction to object-oriented programming that focuses mainly on terminology and the syntax of defining and using objects. Let's quickly review the code that we looked at in the beginning of the chapter. At this point you should fully understand what is going on.

```
stuff = list()
stuff.append('python')
stuff.append('chuck')
stuff.sort()
print (stuff[0])
print (stuff.__getitem__(0))
print (list.__getitem__(stuff,0))

# Code: http://www.py4e.com/code3/party1.py
```

The first line constructs a `list` *object*. When Python creates the `list` object, it calls the *constructor* method (named `__init__`) to set up the internal data attributes that will be used to store the list data. We have not passed any parameters to the *constructor*. When the constructor returns, we use the variable `stuff` to point to the returned instance of the `list` class.

The second and third lines call the `append` method with one parameter to add a new item at the end of the list by updating the attributes within `stuff`. Then in the fourth line, we call the `sort` method with no parameters to sort the data within the `stuff` object.

We then print out the first item in the list using the square brackets which are a shortcut to calling the `__getitem__` method within the `stuff`. This is equivalent to calling the `__getitem__` method in the `list` *class* and passing the `stuff` object as the first parameter and the position we are looking for as the second parameter.

At the end of the program, the `stuff` object is discarded but not before calling the *destructor* (named `__del__`) so that the object can clean up any loose ends as necessary.

Those are the basics of object-oriented programming. There are many additional details as to how to best use object-oriented approaches when developing large applications and libraries that are beyond the scope of this chapter.[3]

14.12 Glossary

attribute A variable that is part of a class.

class A template that can be used to construct an object. Defines the attributes and methods that will make up the object.

child class A new class created when a parent class is extended. The child class inherits all of the attributes and methods of the parent class.

constructor An optional specially named method (`__init__`) that is called at the moment when a class is being used to construct an object. Usually this is used to set up initial values for the object.

destructor An optional specially named method (`__del__`) that is called at the moment just before an object is destroyed. Destructors are rarely used.

inheritance When we create a new class (child) by extending an existing class (parent). The child class has all the attributes and methods of the parent class plus additional attributes and methods defined by the child class.

method A function that is contained within a class and the objects that are constructed from the class. Some object-oriented patterns use 'message' instead of 'method' to describe this concept.

object A constructed instance of a class. An object contains all of the attributes and methods that were defined by the class. Some object-oriented documentation uses the term 'instance' interchangeably with 'object'.

parent class The class which is being extended to create a new child class. The parent class contributes all of its methods and attributes to the new child class.

[3] If you are curious about where the `list` class is defined, take a look at (hopefully the URL won't change) https://github.com/python/cpython/blob/master/Objects/listobject.c - the list class is written in a language called "C". If you take a look at that source code and find it curious you might want to explore a few Computer Science courses.

Chapter 15

Using Databases and SQL

15.1 What is a database?

A *database* is a file that is organized for storing data. Most databases are organized like a dictionary in the sense that they map from keys to values. The biggest difference is that the database is on disk (or other permanent storage), so it persists after the program ends. Because a database is stored on permanent storage, it can store far more data than a dictionary, which is limited to the size of the memory in the computer.

Like a dictionary, database software is designed to keep the inserting and accessing of data very fast, even for large amounts of data. Database software maintains its performance by building *indexes* as data is added to the database to allow the computer to jump quickly to a particular entry.

There are many different database systems which are used for a wide variety of purposes including: Oracle, MySQL, Microsoft SQL Server, PostgreSQL, and SQLite. We focus on SQLite in this book because it is a very common database and is already built into Python. SQLite is designed to be *embedded* into other applications to provide database support within the application. For example, the Firefox browser also uses the SQLite database internally as do many other products.

http://sqlite.org/

SQLite is well suited to some of the data manipulation problems that we see in Informatics such as the Twitter spidering application that we describe in this chapter.

15.2 Database concepts

When you first look at a database it looks like a spreadsheet with multiple sheets. The primary data structures in a database are: *tables*, *rows*, and *columns*.

In technical descriptions of relational databases the concepts of table, row, and column are more formally referred to as *relation*, *tuple*, and *attribute*, respectively. We will use the less formal terms in this chapter.

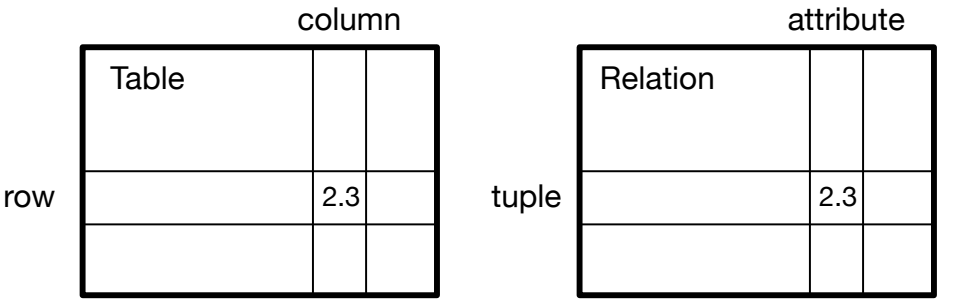

Figure 15.1: Relational Databases

15.3 Database Browser for SQLite

While this chapter will focus on using Python to work with data in SQLite database files, many operations can be done more conveniently using software called the *Database Browser for SQLite* which is freely available from:

http://sqlitebrowser.org/

Using the browser you can easily create tables, insert data, edit data, or run simple SQL queries on the data in the database.

In a sense, the database browser is similar to a text editor when working with text files. When you want to do one or very few operations on a text file, you can just open it in a text editor and make the changes you want. When you have many changes that you need to do to a text file, often you will write a simple Python program. You will find the same pattern when working with databases. You will do simple operations in the database manager and more complex operations will be most conveniently done in Python.

15.4 Creating a database table

Databases require more defined structure than Python lists or dictionaries[1].

When we create a database *table* we must tell the database in advance the names of each of the *columns* in the table and the type of data which we are planning to store in each *column*. When the database software knows the type of data in each column, it can choose the most efficient way to store and look up the data based on the type of data.

You can look at the various data types supported by SQLite at the following url:

http://www.sqlite.org/datatypes.html

Defining structure for your data up front may seem inconvenient at the beginning, but the payoff is fast access to your data even when the database contains a large amount of data.

[1] SQLite actually does allow some flexibility in the type of data stored in a column, but we will keep our data types strict in this chapter so the concepts apply equally to other database systems such as MySQL.

The code to create a database file and a table named `Tracks` with two columns in the database is as follows:

```
import sqlite3

conn = sqlite3.connect('music.sqlite')
cur = conn.cursor()

cur.execute('DROP TABLE IF EXISTS Tracks')
cur.execute('CREATE TABLE Tracks (title TEXT, plays INTEGER)')

conn.close()

# Code: http://www.py4e.com/code3/db1.py
```

The `connect` operation makes a "connection" to the database stored in the file `music.sqlite` in the current directory. If the file does not exist, it will be created. The reason this is called a "connection" is that sometimes the database is stored on a separate "database server" from the server on which we are running our application. In our simple examples the database will just be a local file in the same directory as the Python code we are running.

A *cursor* is like a file handle that we can use to perform operations on the data stored in the database. Calling `cursor()` is very similar conceptually to calling `open()` when dealing with text files.

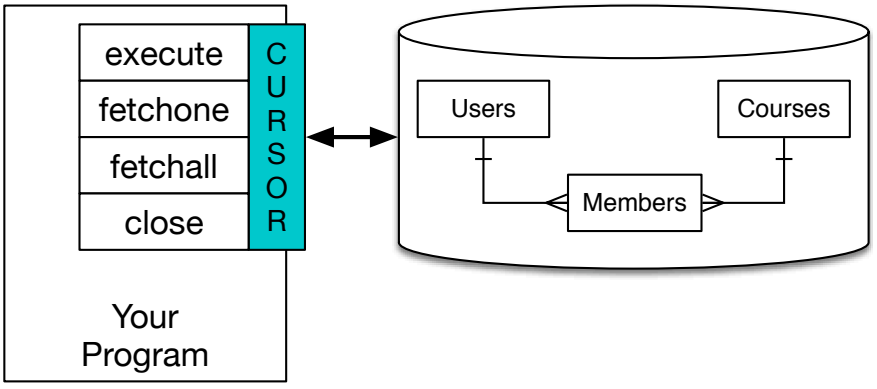

Figure 15.2: A Database Cursor

Once we have the cursor, we can begin to execute commands on the contents of the database using the `execute()` method.

Database commands are expressed in a special language that has been standardized across many different database vendors to allow us to learn a single database language. The database language is called *Structured Query Language* or *SQL* for short.

http://en.wikipedia.org/wiki/SQL

In our example, we are executing two SQL commands in our database. As a convention, we will show the SQL keywords in uppercase and the parts of the

command that we are adding (such as the table and column names) will be shown in lowercase.

The first SQL command removes the `Tracks` table from the database if it exists. This pattern is simply to allow us to run the same program to create the `Tracks` table over and over again without causing an error. Note that the `DROP TABLE` command deletes the table and all of its contents from the database (i.e., there is no "undo").

```
cur.execute('DROP TABLE IF EXISTS Tracks ')
```

The second command creates a table named `Tracks` with a text column named `title` and an integer column named `plays`.

```
cur.execute('CREATE TABLE Tracks (title TEXT, plays INTEGER)')
```

Now that we have created a table named `Tracks`, we can put some data into that table using the SQL `INSERT` operation. Again, we begin by making a connection to the database and obtaining the `cursor`. We can then execute SQL commands using the cursor.

The SQL `INSERT` command indicates which table we are using and then defines a new row by listing the fields we want to include `(title, plays)` followed by the `VALUES` we want placed in the new row. We specify the values as question marks `(?, ?)` to indicate that the actual values are passed in as a tuple `( 'My Way', 15 )` as the second parameter to the `execute()` call.

```
import sqlite3

conn = sqlite3.connect('music.sqlite')
cur = conn.cursor()

cur.execute('INSERT INTO Tracks (title, plays) VALUES (?, ?)',
    ('Thunderstruck', 20))
cur.execute('INSERT INTO Tracks (title, plays) VALUES (?, ?)',
    ('My Way', 15))
conn.commit()

print('Tracks:')
cur.execute('SELECT title, plays FROM Tracks')
for row in cur:
     print(row)

cur.execute('DELETE FROM Tracks WHERE plays < 100')
conn.commit()

cur.close()

# Code: http://www.py4e.com/code3/db2.py
```

Tracks

title	plays
Thunderstruck	20
My Way	15

Figure 15.3: Rows in a Table

First we `INSERT` two rows into our table and use `commit()` to force the data to be written to the database file.

Then we use the `SELECT` command to retrieve the rows we just inserted from the table. On the `SELECT` command, we indicate which columns we would like `(title, plays)` and indicate which table we want to retrieve the data from. After we execute the `SELECT` statement, the cursor is something we can loop through in a `for` statement. For efficiency, the cursor does not read all of the data from the database when we execute the `SELECT` statement. Instead, the data is read on demand as we loop through the rows in the `for` statement.

The output of the program is as follows:

```
Tracks:
('Thunderstruck', 20)
('My Way', 15)
```

Our `for` loop finds two rows, and each row is a Python tuple with the first value as the `title` and the second value as the number of `plays`.

Note: You may see strings starting with `u'` *in other books or on the Internet. This was an indication in Python 2 that the strings are* Unicode* strings that are capable of storing non-Latin character sets. In Python 3, all strings are unicode strings by default.*

At the very end of the program, we execute an SQL command to `DELETE` the rows we have just created so we can run the program over and over. The `DELETE` command shows the use of a `WHERE` clause that allows us to express a selection criterion so that we can ask the database to apply the command to only the rows that match the criterion. In this example the criterion happens to apply to all the rows so we empty the table out so we can run the program repeatedly. After the `DELETE` is performed, we also call `commit()` to force the data to be removed from the database.

15.5 Structured Query Language summary

So far, we have been using the Structured Query Language in our Python examples and have covered many of the basics of the SQL commands. In this section, we look at the SQL language in particular and give an overview of SQL syntax.

Since there are so many different database vendors, the Structured Query Language (SQL) was standardized so we could communicate in a portable manner to database systems from multiple vendors.

A relational database is made up of tables, rows, and columns. The columns generally have a type such as text, numeric, or date data. When we create a table, we indicate the names and types of the columns:

```
CREATE TABLE Tracks (title TEXT, plays INTEGER)
```

To insert a row into a table, we use the SQL `INSERT` command:

```
INSERT INTO Tracks (title, plays) VALUES ('My Way', 15)
```

The `INSERT` statement specifies the table name, then a list of the fields/columns that you would like to set in the new row, and then the keyword `VALUES` and a list of corresponding values for each of the fields.

The SQL `SELECT` command is used to retrieve rows and columns from a database. The `SELECT` statement lets you specify which columns you would like to retrieve as well as a `WHERE` clause to select which rows you would like to see. It also allows an optional `ORDER BY` clause to control the sorting of the returned rows.

```
SELECT * FROM Tracks WHERE title = 'My Way'
```

Using `*` indicates that you want the database to return all of the columns for each row that matches the `WHERE` clause.

Note, unlike in Python, in a SQL `WHERE` clause we use a single equal sign to indicate a test for equality rather than a double equal sign. Other logical operations allowed in a `WHERE` clause include `<`, `>`, `<=`, `>=`, `!=`, as well as `AND` and `OR` and parentheses to build your logical expressions.

You can request that the returned rows be sorted by one of the fields as follows:

```
SELECT title,plays FROM Tracks ORDER BY title
```

To remove a row, you need a `WHERE` clause on an SQL `DELETE` statement. The `WHERE` clause determines which rows are to be deleted:

```
DELETE FROM Tracks WHERE title = 'My Way'
```

It is possible to `UPDATE` a column or columns within one or more rows in a table using the SQL `UPDATE` statement as follows:

```
UPDATE Tracks SET plays = 16 WHERE title = 'My Way'
```

The `UPDATE` statement specifies a table and then a list of fields and values to change after the `SET` keyword and then an optional `WHERE` clause to select the rows that are to be updated. A single `UPDATE` statement will change all of the rows that match the `WHERE` clause. If a `WHERE` clause is not specified, it performs the `UPDATE` on all of the rows in the table.

These four basic SQL commands (INSERT, SELECT, UPDATE, and DELETE) allow the four basic operations needed to create and maintain data.

15.6 Spidering Twitter using a database

In this section, we will create a simple spidering program that will go through Twitter accounts and build a database of them. *Note: Be very careful when running this program. You do not want to pull too much data or run the program for too long and end up having your Twitter access shut off.*

One of the problems of any kind of spidering program is that it needs to be able to be stopped and restarted many times and you do not want to lose the data that you have retrieved so far. You don't want to always restart your data retrieval at the very beginning so we want to store data as we retrieve it so our program can start back up and pick up where it left off.

We will start by retrieving one person's Twitter friends and their statuses, looping through the list of friends, and adding each of the friends to a database to be retrieved in the future. After we process one person's Twitter friends, we check in our database and retrieve one of the friends of the friend. We do this over and over, picking an "unvisited" person, retrieving their friend list, and adding friends we have not seen to our list for a future visit.

We also track how many times we have seen a particular friend in the database to get some sense of their "popularity".

By storing our list of known accounts and whether we have retrieved the account or not, and how popular the account is in a database on the disk of the computer, we can stop and restart our program as many times as we like.

This program is a bit complex. It is based on the code from the exercise earlier in the book that uses the Twitter API.

Here is the source code for our Twitter spidering application:

```
from urllib.request import urlopen
import urllib.error
import twurl
import json
import sqlite3
import ssl

TWITTER_URL = 'https://api.twitter.com/1.1/friends/list.json'

conn = sqlite3.connect('spider.sqlite')
cur = conn.cursor()

cur.execute('''
            CREATE TABLE IF NOT EXISTS Twitter
            (name TEXT, retrieved INTEGER, friends INTEGER)''')

# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE
```

```
while True:
    acct = input('Enter a Twitter account, or quit: ')
    if (acct == 'quit'): break
    if (len(acct) < 1):
        cur.execute('SELECT name FROM Twitter WHERE retrieved = 0 LIMIT 1')
        try:
            acct = cur.fetchone()[0]
        except:
            print('No unretrieved Twitter accounts found')
            continue

    url = twurl.augment(TWITTER_URL, {'screen_name': acct, 'count': '5'})
    print('Retrieving', url)
    connection = urlopen(url, context=ctx)
    data = connection.read().decode()
    headers = dict(connection.getheaders())

    print('Remaining', headers['x-rate-limit-remaining'])
    js = json.loads(data)
    # Debugging
    # print json.dumps(js, indent=4)

    cur.execute('UPDATE Twitter SET retrieved=1 WHERE name = ?', (acct, ))

    countnew = 0
    countold = 0
    for u in js['users']:
        friend = u['screen_name']
        print(friend)
        cur.execute('SELECT friends FROM Twitter WHERE name = ? LIMIT 1',
                    (friend, ))
        try:
            count = cur.fetchone()[0]
            cur.execute('UPDATE Twitter SET friends = ? WHERE name = ?',
                        (count+1, friend))
            countold = countold + 1
        except:
            cur.execute('''INSERT INTO Twitter (name, retrieved, friends)
                        VALUES (?, 0, 1)''', (friend, ))
            countnew = countnew + 1
    print('New accounts=', countnew, ' revisited=', countold)
    conn.commit()

cur.close()

# Code: http://www.py4e.com/code3/twspider.py
```

Our database is stored in the file `spider.sqlite` and it has one table named `Twitter`. Each row in the `Twitter` table has a column for the account name, whether we have retrieved the friends of this account, and how many times this account has been "friended".

In the main loop of the program, we prompt the user for a Twitter account name or "quit" to exit the program. If the user enters a Twitter account, we retrieve the list of friends and statuses for that user and add each friend to the database if not already in the database. If the friend is already in the list, we add 1 to the `friends` field in the row in the database.

If the user presses enter, we look in the database for the next Twitter account that we have not yet retrieved, retrieve the friends and statuses for that account, add them to the database or update them, and increase their `friends` count.

Once we retrieve the list of friends and statuses, we loop through all of the `user` items in the returned JSON and retrieve the `screen_name` for each user. Then we use the `SELECT` statement to see if we already have stored this particular `screen_name` in the database and retrieve the friend count (`friends`) if the record exists.

```
countnew = 0
countold = 0
for u in js['users'] :
    friend = u['screen_name']
    print(friend)
    cur.execute('SELECT friends FROM Twitter WHERE name = ? LIMIT 1',
        (friend, ) )
    try:
        count = cur.fetchone()[0]
        cur.execute('UPDATE Twitter SET friends = ? WHERE name = ?',
            (count+1, friend) )
        countold = countold + 1
    except:
        cur.execute('''INSERT INTO Twitter (name, retrieved, friends)
            VALUES ( ?, 0, 1 )''', ( friend, ) )
        countnew = countnew + 1
print('New accounts=',countnew,' revisited=',countold)
conn.commit()
```

Once the cursor executes the `SELECT` statement, we must retrieve the rows. We could do this with a `for` statement, but since we are only retrieving one row (`LIMIT 1`), we can use the `fetchone()` method to fetch the first (and only) row that is the result of the `SELECT` operation. Since `fetchone()` returns the row as a *tuple* (even though there is only one field), we take the first value from the tuple using to get the current friend count into the variable `count`.

If this retrieval is successful, we use the SQL `UPDATE` statement with a `WHERE` clause to add 1 to the `friends` column for the row that matches the friend's account. Notice that there are two placeholders (i.e., question marks) in the SQL, and the second parameter to the `execute()` is a two-element tuple that holds the values to be substituted into the SQL in place of the question marks.

If the code in the `try` block fails, it is probably because no record matched the `WHERE name = ?` clause on the SELECT statement. So in the `except` block, we use the SQL `INSERT` statement to add the friend's `screen_name` to the table with an indication that we have not yet retrieved the `screen_name` and set the friend count to zero.

So the first time the program runs and we enter a Twitter account, the program runs as follows:

```
Enter a Twitter account, or quit: drchuck
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 20  revisited= 0
Enter a Twitter account, or quit: quit
```

Since this is the first time we have run the program, the database is empty and we create the database in the file `spider.sqlite` and add a table named `Twitter` to the database. Then we retrieve some friends and add them all to the database since the database is empty.

At this point, we might want to write a simple database dumper to take a look at what is in our `spider.sqlite` file:

```
import sqlite3

conn = sqlite3.connect('spider.sqlite')
cur = conn.cursor()
cur.execute('SELECT * FROM Twitter')
count = 0
for row in cur:
    print(row)
    count = count + 1
print(count, 'rows.')
cur.close()

# Code: http://www.py4e.com/code3/twdump.py
```

This program simply opens the database and selects all of the columns of all of the rows in the table `Twitter`, then loops through the rows and prints out each row.

If we run this program after the first execution of our Twitter spider above, its output will be as follows:

```
('opencontent', 0, 1)
('lhawthorn', 0, 1)
('steve_coppin', 0, 1)
('davidkocher', 0, 1)
('hrheingold', 0, 1)
...
20 rows.
```

We see one row for each `screen_name`, that we have not retrieved the data for that `screen_name`, and everyone in the database has one friend.

Now our database reflects the retrieval of the friends of our first Twitter account (*drchuck*). We can run the program again and tell it to retrieve the friends of the next "unprocessed" account by simply pressing enter instead of a Twitter account as follows:

```
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 18  revisited= 2
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 17  revisited= 3
Enter a Twitter account, or quit: quit
```

Since we pressed enter (i.e., we did not specify a Twitter account), the following code is executed:

```
if ( len(acct) < 1 ) :
    cur.execute('SELECT name FROM Twitter WHERE retrieved = 0 LIMIT 1')
    try:
        acct = cur.fetchone()[0]
    except:
        print('No unretrieved twitter accounts found')
        continue
```

We use the SQL `SELECT` statement to retrieve the name of the first (`LIMIT 1`) user who still has their "have we retrieved this user" value set to zero. We also use the `fetchone()[0]` pattern within a try/except block to either extract a `screen_name` from the retrieved data or put out an error message and loop back up.

If we successfully retrieved an unprocessed `screen_name`, we retrieve their data as follows:

```
url=twurl.augment(TWITTER_URL,{'screen_name': acct,'count': '20'})
print('Retrieving', url)
connection = urllib.urlopen(url)
data = connection.read()
js = json.loads(data)

cur.execute('UPDATE Twitter SET retrieved=1 WHERE name = ?',(acct, ))
```

Once we retrieve the data successfully, we use the `UPDATE` statement to set the `retrieved` column to 1 to indicate that we have completed the retrieval of the friends of this account. This keeps us from retrieving the same data over and over and keeps us progressing forward through the network of Twitter friends.

If we run the friend program and press enter twice to retrieve the next unvisited friend's friends, then run the dumping program, it will give us the following output:

```
('opencontent', 1, 1)
('lhawthorn', 1, 1)
('steve_coppin', 0, 1)
('davidkocher', 0, 1)
('hrheingold', 0, 1)
...
('cnxorg', 0, 2)
('knoop', 0, 1)
```

```
('kthanos', 0, 2)
('LectureTools', 0, 1)
...
55 rows.
```

We can see that we have properly recorded that we have visited `lhawthorn` and `opencontent`. Also the accounts `cnxorg` and `kthanos` already have two followers. Since we now have retrieved the friends of three people (`drchuck`, `opencontent`, and `lhawthorn`) our table has 55 rows of friends to retrieve.

Each time we run the program and press enter it will pick the next unvisited account (e.g., the next account will be `steve_coppin`), retrieve their friends, mark them as retrieved, and for each of the friends of `steve_coppin` either add them to the end of the database or update their friend count if they are already in the database.

Since the program's data is all stored on disk in a database, the spidering activity can be suspended and resumed as many times as you like with no loss of data.

15.7 Basic data modeling

The real power of a relational database is when we create multiple tables and make links between those tables. The act of deciding how to break up your application data into multiple tables and establishing the relationships between the tables is called *data modeling*. The design document that shows the tables and their relationships is called a *data model*.

Data modeling is a relatively sophisticated skill and we will only introduce the most basic concepts of relational data modeling in this section. For more detail on data modeling you can start with:

http://en.wikipedia.org/wiki/Relational_model

Let's say for our Twitter spider application, instead of just counting a person's friends, we wanted to keep a list of all of the incoming relationships so we could find a list of everyone who is following a particular account.

Since everyone will potentially have many accounts that follow them, we cannot simply add a single column to our `Twitter` table. So we create a new table that keeps track of pairs of friends. The following is a simple way of making such a table:

```
CREATE TABLE Pals (from_friend TEXT, to_friend TEXT)
```

Each time we encounter a person who `drchuck` is following, we would insert a row of the form:

```
INSERT INTO Pals (from_friend,to_friend) VALUES ('drchuck', 'lhawthorn')
```

As we are processing the 20 friends from the `drchuck` Twitter feed, we will insert 20 records with "drchuck" as the first parameter so we will end up duplicating the string many times in the database.

This duplication of string data violates one of the best practices for *database normalization* which basically states that we should never put the same string data in the database more than once. If we need the data more than once, we create a numeric *key* for the data and reference the actual data using this key.

In practical terms, a string takes up a lot more space than an integer on the disk and in the memory of our computer, and takes more processor time to compare and sort. If we only have a few hundred entries, the storage and processor time hardly matters. But if we have a million people in our database and a possibility of 100 million friend links, it is important to be able to scan data as quickly as possible.

We will store our Twitter accounts in a table named `People` instead of the `Twitter` table used in the previous example. The `People` table has an additional column to store the numeric key associated with the row for this Twitter user. SQLite has a feature that automatically adds the key value for any row we insert into a table using a special type of data column (`INTEGER PRIMARY KEY`).

We can create the `People` table with this additional `id` column as follows:

```
CREATE TABLE People
    (id INTEGER PRIMARY KEY, name TEXT UNIQUE, retrieved INTEGER)
```

Notice that we are no longer maintaining a friend count in each row of the `People` table. When we select `INTEGER PRIMARY KEY` as the type of our `id` column, we are indicating that we would like SQLite to manage this column and assign a unique numeric key to each row we insert automatically. We also add the keyword `UNIQUE` to indicate that we will not allow SQLite to insert two rows with the same value for `name`.

Now instead of creating the table `Pals` above, we create a table called `Follows` with two integer columns `from_id` and `to_id` and a constraint on the table that the *combination* of `from_id` and `to_id` must be unique in this table (i.e., we cannot insert duplicate rows) in our database.

```
CREATE TABLE Follows
    (from_id INTEGER, to_id INTEGER, UNIQUE(from_id, to_id) )
```

When we add `UNIQUE` clauses to our tables, we are communicating a set of rules that we are asking the database to enforce when we attempt to insert records. We are creating these rules as a convenience in our programs, as we will see in a moment. The rules both keep us from making mistakes and make it simpler to write some of our code.

In essence, in creating this `Follows` table, we are modelling a "relationship" where one person "follows" someone else and representing it with a pair of numbers indicating that (a) the people are connected and (b) the direction of the relationship.

15.8 Programming with multiple tables

We will now redo the Twitter spider program using two tables, the primary keys, and the key references as described above. Here is the code for the new version of the program:

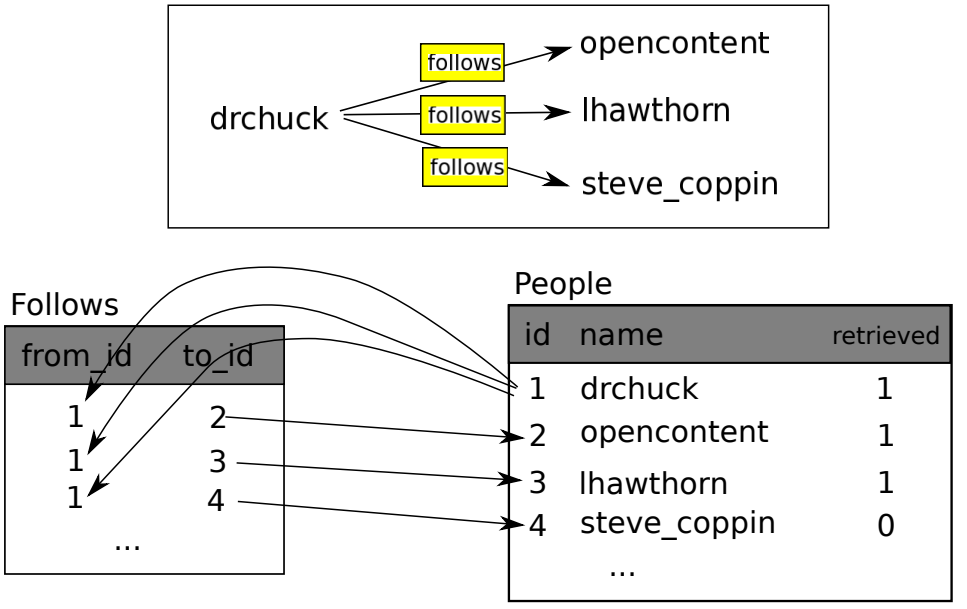

Figure 15.4: Relationships Between Tables

```
import urllib.request, urllib.parse, urllib.error
import twurl
import json
import sqlite3
import ssl

TWITTER_URL = 'https://api.twitter.com/1.1/friends/list.json'

conn = sqlite3.connect('friends.sqlite')
cur = conn.cursor()

cur.execute('''CREATE TABLE IF NOT EXISTS People
            (id INTEGER PRIMARY KEY, name TEXT UNIQUE, retrieved INTEGER)''')
cur.execute('''CREATE TABLE IF NOT EXISTS Follows
            (from_id INTEGER, to_id INTEGER, UNIQUE(from_id, to_id))''')

# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

while True:
    acct = input('Enter a Twitter account, or quit: ')
    if (acct == 'quit'): break
    if (len(acct) < 1):
        cur.execute('SELECT id, name FROM People WHERE retrieved=0 LIMIT 1')
        try:
            (id, acct) = cur.fetchone()
```

```
        except:
            print('No unretrieved Twitter accounts found')
            continue
    else:
        cur.execute('SELECT id FROM People WHERE name = ? LIMIT 1',
                    (acct, ))
        try:
            id = cur.fetchone()[0]
        except:
            cur.execute('''INSERT OR IGNORE INTO People
                        (name, retrieved) VALUES (?, 0)''', (acct, ))
            conn.commit()
            if cur.rowcount != 1:
                print('Error inserting account:', acct)
                continue
            id = cur.lastrowid

    url = twurl.augment(TWITTER_URL, {'screen_name': acct, 'count': '100'})
    print('Retrieving account', acct)
    try:
        connection = urllib.request.urlopen(url, context=ctx)
    except Exception as err:
        print('Failed to Retrieve', err)
        break

    data = connection.read().decode()
    headers = dict(connection.getheaders())

    print('Remaining', headers['x-rate-limit-remaining'])

    try:
        js = json.loads(data)
    except:
        print('Unable to parse json')
        print(data)
        break

    # Debugging
    # print(json.dumps(js, indent=4))

    if 'users' not in js:
        print('Incorrect JSON received')
        print(json.dumps(js, indent=4))
        continue

    cur.execute('UPDATE People SET retrieved=1 WHERE name = ?', (acct, ))

    countnew = 0
    countold = 0
    for u in js['users']:
        friend = u['screen_name']
```

```
        print(friend)
        cur.execute('SELECT id FROM People WHERE name = ? LIMIT 1',
                    (friend, ))
        try:
            friend_id = cur.fetchone()[0]
            countold = countold + 1
        except:
            cur.execute('''INSERT OR IGNORE INTO People (name, retrieved)
                        VALUES (?, 0)''', (friend, ))
            conn.commit()
            if cur.rowcount != 1:
                print('Error inserting account:', friend)
                continue
            friend_id = cur.lastrowid
            countnew = countnew + 1
        cur.execute('''INSERT OR IGNORE INTO Follows (from_id, to_id)
                    VALUES (?, ?)''', (id, friend_id))
    print('New accounts=', countnew, ' revisited=', countold)
    print('Remaining', headers['x-rate-limit-remaining'])
    conn.commit()
cur.close()

# Code: http://www.py4e.com/code3/twfriends.py
```

This program is starting to get a bit complicated, but it illustrates the patterns that we need to use when we are using integer keys to link tables. The basic patterns are:

1. Create tables with primary keys and constraints.
2. When we have a logical key for a person (i.e., account name) and we need the `id` value for the person, depending on whether or not the person is already in the `People` table we either need to: (1) look up the person in the `People` table and retrieve the `id` value for the person or (2) add the person to the `People` table and get the `id` value for the newly added row.
3. Insert the row that captures the "follows" relationship.

We will cover each of these in turn.

15.8.1 Constraints in database tables

As we design our table structures, we can tell the database system that we would like it to enforce a few rules on us. These rules help us from making mistakes and introducing incorrect data into out tables. When we create our tables:

```
cur.execute('''CREATE TABLE IF NOT EXISTS People
    (id INTEGER PRIMARY KEY, name TEXT UNIQUE, retrieved INTEGER)''')
cur.execute('''CREATE TABLE IF NOT EXISTS Follows
    (from_id INTEGER, to_id INTEGER, UNIQUE(from_id, to_id))''')
```

We indicate that the `name` column in the `People` table must be `UNIQUE`. We also indicate that the combination of the two numbers in each row of the `Follows` table must be unique. These constraints keep us from making mistakes such as adding the same relationship more than once.

We can take advantage of these constraints in the following code:

```
cur.execute('''INSERT OR IGNORE INTO People (name, retrieved)
    VALUES ( ?, 0)''', ( friend, ) )
```

We add the `OR IGNORE` clause to our `INSERT` statement to indicate that if this particular `INSERT` would cause a violation of the "`name` must be unique" rule, the database system is allowed to ignore the `INSERT`. We are using the database constraint as a safety net to make sure we don't inadvertently do something incorrect.

Similarly, the following code ensures that we don't add the exact same `Follows` relationship twice.

```
cur.execute('''INSERT OR IGNORE INTO Follows
    (from_id, to_id) VALUES (?, ?)''', (id, friend_id) )
```

Again, we simply tell the database to ignore our attempted `INSERT` if it would violate the uniqueness constraint that we specified for the `Follows` rows.

15.8.2 Retrieve and/or insert a record

When we prompt the user for a Twitter account, if the account exists, we must look up its `id` value. If the account does not yet exist in the `People` table, we must insert the record and get the `id` value from the inserted row.

This is a very common pattern and is done twice in the program above. This code shows how we look up the `id` for a friend's account when we have extracted a `screen_name` from a `user` node in the retrieved Twitter JSON.

Since over time it will be increasingly likely that the account will already be in the database, we first check to see if the `People` record exists using a `SELECT` statement.

If all goes well[2] inside the `try` section, we retrieve the record using `fetchone()` and then retrieve the first (and only) element of the returned tuple and store it in `friend_id`.

If the `SELECT` fails, the `fetchone()[0]` code will fail and control will transfer into the `except` section.

```
    friend = u['screen_name']
    cur.execute('SELECT id FROM People WHERE name = ? LIMIT 1',
        (friend, ) )
    try:
```

[2] In general, when a sentence starts with "if all goes well" you will find that the code needs to use try/except.

```
        friend_id = cur.fetchone()[0]
        countold = countold + 1
    except:
        cur.execute('''INSERT OR IGNORE INTO People (name, retrieved)
            VALUES ( ?, 0)''', ( friend, ) )
        conn.commit()
        if cur.rowcount != 1 :
            print('Error inserting account:',friend)
            continue
        friend_id = cur.lastrowid
        countnew = countnew + 1
```

If we end up in the `except` code, it simply means that the row was not found, so we must insert the row. We use `INSERT OR IGNORE` just to avoid errors and then call `commit()` to force the database to really be updated. After the write is done, we can check the `cur.rowcount` to see how many rows were affected. Since we are attempting to insert a single row, if the number of affected rows is something other than 1, it is an error.

If the `INSERT` is successful, we can look at `cur.lastrowid` to find out what value the database assigned to the `id` column in our newly created row.

15.8.3 Storing the friend relationship

Once we know the key value for both the Twitter user and the friend in the JSON, it is a simple matter to insert the two numbers into the `Follows` table with the following code:

```
cur.execute('INSERT OR IGNORE INTO Follows (from_id, to_id) VALUES (?, ?)',
    (id, friend_id) )
```

Notice that we let the database take care of keeping us from "double-inserting" a relationship by creating the table with a uniqueness constraint and then adding `OR IGNORE` to our `INSERT` statement.

Here is a sample execution of this program:

```
Enter a Twitter account, or quit:
No unretrieved Twitter accounts found
Enter a Twitter account, or quit: drchuck
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 20  revisited= 0
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 17  revisited= 3
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 17  revisited= 3
Enter a Twitter account, or quit: quit
```

We started with the drchuck account and then let the program automatically pick the next two accounts to retrieve and add to our database.

The following is the first few rows in the People and Follows tables after this run is completed:

```
People:
(1, 'drchuck', 1)
(2, 'opencontent', 1)
(3, 'lhawthorn', 1)
(4, 'steve_coppin', 0)
(5, 'davidkocher', 0)
55 rows.
Follows:
(1, 2)
(1, 3)
(1, 4)
(1, 5)
(1, 6)
60 rows.
```

You can see the id, name, and visited fields in the People table and you see the numbers of both ends of the relationship in the Follows table. In the People table, we can see that the first three people have been visited and their data has been retrieved. The data in the Follows table indicates that drchuck (user 1) is a friend to all of the people shown in the first five rows. This makes sense because the first data we retrieved and stored was the Twitter friends of drchuck. If you were to print more rows from the Follows table, you would see the friends of users 2 and 3 as well.

15.9 Three kinds of keys

Now that we have started building a data model putting our data into multiple linked tables and linking the rows in those tables using *keys*, we need to look at some terminology around keys. There are generally three kinds of keys used in a database model.

- A *logical key* is a key that the "real world" might use to look up a row. In our example data model, the name field is a logical key. It is the screen name for the user and we indeed look up a user's row several times in the program using the name field. You will often find that it makes sense to add a UNIQUE constraint to a logical key. Since the logical key is how we look up a row from the outside world, it makes little sense to allow multiple rows with the same value in the table.
- A *primary key* is usually a number that is assigned automatically by the database. It generally has no meaning outside the program and is only used to link rows from different tables together. When we want to look up a row in a table, usually searching for the row using the primary key is the fastest way to find the row. Since primary keys are integer numbers, they take up very little storage and can be compared or sorted very quickly. In our data model, the id field is an example of a primary key.

- A *foreign key* is usually a number that points to the primary key of an associated row in a different table. An example of a foreign key in our data model is the `from_id`.

We are using a naming convention of always calling the primary key field name `id` and appending the suffix `_id` to any field name that is a foreign key.

15.10 Using JOIN to retrieve data

Now that we have followed the rules of database normalization and have data separated into two tables, linked together using primary and foreign keys, we need to be able to build a `SELECT` that reassembles the data across the tables.

SQL uses the `JOIN` clause to reconnect these tables. In the `JOIN` clause you specify the fields that are used to reconnect the rows between the tables.

The following is an example of a `SELECT` with a `JOIN` clause:

```
SELECT * FROM Follows JOIN People
    ON Follows.from_id = People.id WHERE People.id = 1
```

The `JOIN` clause indicates that the fields we are selecting cross both the `Follows` and `People` tables. The `ON` clause indicates how the two tables are to be joined: Take the rows from `Follows` and append the row from `People` where the field `from_id` in `Follows` is the same the `id` value in the `People` table.

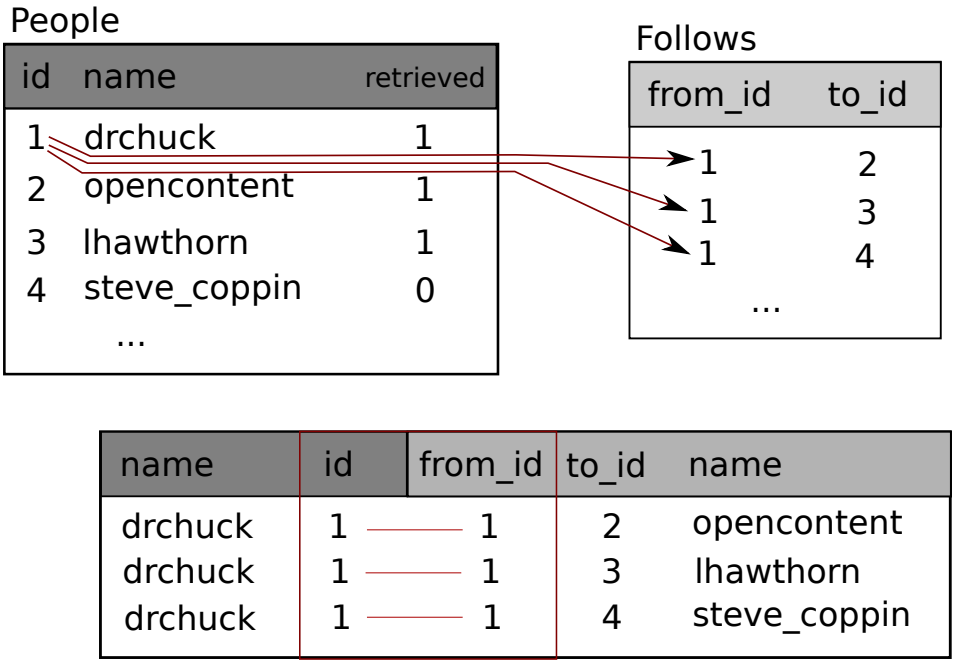

Figure 15.5: Connecting Tables Using JOIN

The result of the JOIN is to create extra-long "metarows" which have both the fields from `People` and the matching fields from `Follows`. Where there is more than one match between the `id` field from `People` and the `from_id` from `People`, then JOIN creates a metarow for *each* of the matching pairs of rows, duplicating data as needed.

The following code demonstrates the data that we will have in the database after the multi-table Twitter spider program (above) has been run several times.

```
import sqlite3

conn = sqlite3.connect('friends.sqlite')
cur = conn.cursor()

cur.execute('SELECT * FROM People')
count = 0
print('People:')
for row in cur:
    if count < 5: print(row)
    count = count + 1
print(count, 'rows.')

cur.execute('SELECT * FROM Follows')
count = 0
print('Follows:')
for row in cur:
    if count < 5: print(row)
    count = count + 1
print(count, 'rows.')

cur.execute('''SELECT * FROM Follows JOIN People
            ON Follows.to_id = People.id
            WHERE Follows.from_id = 2''')
count = 0
print('Connections for id=2:')
for row in cur:
    if count < 5: print(row)
    count = count + 1
print(count, 'rows.')

cur.close()

# Code: http://www.py4e.com/code3/twjoin.py
```

In this program, we first dump out the `People` and `Follows` and then dump out a subset of the data in the tables joined together.

Here is the output of the program:

```
python twjoin.py
People:
```

```
(1, 'drchuck', 1)
(2, 'opencontent', 1)
(3, 'lhawthorn', 1)
(4, 'steve_coppin', 0)
(5, 'davidkocher', 0)
55 rows.
Follows:
(1, 2)
(1, 3)
(1, 4)
(1, 5)
(1, 6)
60 rows.
Connections for id=2:
(2, 1, 1, 'drchuck', 1)
(2, 28, 28, 'cnxorg', 0)
(2, 30, 30, 'kthanos', 0)
(2, 102, 102, 'SomethingGirl', 0)
(2, 103, 103, 'ja_Pac', 0)
20 rows.
```

You see the columns from the `People` and `Follows` tables and the last set of rows is the result of the `SELECT` with the `JOIN` clause.

In the last select, we are looking for accounts that are friends of "opencontent" (i.e., `People.id=2`).

In each of the "metarows" in the last select, the first two columns are from the `Follows` table followed by columns three through five from the `People` table. You can also see that the second column (`Follows.to_id`) matches the third column (`People.id`) in each of the joined-up "metarows".

15.11 Summary

This chapter has covered a lot of ground to give you an overview of the basics of using a database in Python. It is more complicated to write the code to use a database to store data than Python dictionaries or flat files so there is little reason to use a database unless your application truly needs the capabilities of a database. The situations where a database can be quite useful are: (1) when your application needs to make small many random updates within a large data set, (2) when your data is so large it cannot fit in a dictionary and you need to look up information repeatedly, or (3) when you have a long-running process that you want to be able to stop and restart and retain the data from one run to the next.

You can build a simple database with a single table to suit many application needs, but most problems will require several tables and links/relationships between rows in different tables. When you start making links between tables, it is important to do some thoughtful design and follow the rules of database normalization to make the best use of the database's capabilities. Since the primary motivation for using a database is that you have a large amount of data to deal with, it is important to model your data efficiently so your programs run as fast as possible.

15.12 Debugging

One common pattern when you are developing a Python program to connect to an SQLite database will be to run a Python program and check the results using the Database Browser for SQLite. The browser allows you to quickly check to see if your program is working properly.

You must be careful because SQLite takes care to keep two programs from changing the same data at the same time. For example, if you open a database in the browser and make a change to the database and have not yet pressed the "save" button in the browser, the browser "locks" the database file and keeps any other program from accessing the file. In particular, your Python program will not be able to access the file if it is locked.

So a solution is to make sure to either close the database browser or use the *File* menu to close the database in the browser before you attempt to access the database from Python to avoid the problem of your Python code failing because the database is locked.

15.13 Glossary

attribute One of the values within a tuple. More commonly called a "column" or "field".

constraint When we tell the database to enforce a rule on a field or a row in a table. A common constraint is to insist that there can be no duplicate values in a particular field (i.e., all the values must be unique).

cursor A cursor allows you to execute SQL commands in a database and retrieve data from the database. A cursor is similar to a socket or file handle for network connections and files, respectively.

database browser A piece of software that allows you to directly connect to a database and manipulate the database directly without writing a program.

foreign key A numeric key that points to the primary key of a row in another table. Foreign keys establish relationships between rows stored in different tables.

index Additional data that the database software maintains as rows and inserts into a table to make lookups very fast.

logical key A key that the "outside world" uses to look up a particular row. For example in a table of user accounts, a person's email address might be a good candidate as the logical key for the user's data.

normalization Designing a data model so that no data is replicated. We store each item of data at one place in the database and reference it elsewhere using a foreign key.

primary key A numeric key assigned to each row that is used to refer to one row in a table from another table. Often the database is configured to automatically assign primary keys as rows are inserted.

relation An area within a database that contains tuples and attributes. More typically called a "table".

tuple A single entry in a database table that is a set of attributes. More typically called "row".

Chapter 16

Visualizing data

So far we have been learning the Python language and then learning how to use Python, the network, and databases to manipulate data.

In this chapter, we take a look at three complete applications that bring all of these things together to manage and visualize data. You might use these applications as sample code to help get you started in solving a real-world problem.

Each of the applications is a ZIP file that you can download and extract onto your computer and execute.

16.1 Building a Google map from geocoded data

In this project, we are using the Google geocoding API to clean up some user-entered geographic locations of university names and then placing the data on a Google map.

To get started, download the application from:

www.py4e.com/code3/geodata.zip

The first problem to solve is that the free Google geocoding API is rate-limited to a certain number of requests per day. If you have a lot of data, you might need to stop and restart the lookup process several times. So we break the problem into two phases.

In the first phase we take our input "survey" data in the file *where.data* and read it one line at a time, and retrieve the geocoded information from Google and store it in a database *geodata.sqlite*. Before we use the geocoding API for each user-entered location, we simply check to see if we already have the data for that particular line of input. The database is functioning as a local "cache" of our geocoding data to make sure we never ask Google for the same data twice.

You can restart the process at any time by removing the file *geodata.sqlite*.

Run the *geoload.py* program. This program will read the input lines in *where.data* and for each line check to see if it is already in the database. If we don't have the

Figure 16.1: A Google Map

data for the location, it will call the geocoding API to retrieve the data and store it in the database.

Here is a sample run after there is already some data in the database:

```
Found in database  Northeastern University
Found in database  University of Hong Kong, ...
Found in database  Technion
Found in database  Viswakarma Institute, Pune, India
Found in database  UMD
Found in database  Tufts University

Resolving Monash University
Retrieving http://maps.googleapis.com/maps/api/
    geocode/json?address=Monash+University
Retrieved 2063 characters {    "results" : [
{'status': 'OK', 'results': ... }

Resolving Kokshetau Institute of Economics and Management
Retrieving http://maps.googleapis.com/maps/api/
    geocode/json?address=Kokshetau+Inst ...
Retrieved 1749 characters {    "results" : [
{'status': 'OK', 'results': ... }
...
```

The first five locations are already in the database and so they are skipped. The program scans to the point where it finds new locations and starts retrieving them.

The *geoload.py* program can be stopped at any time, and there is a counter that you can use to limit the number of calls to the geocoding API for each run. Given that the *where.data* only has a few hundred data items, you should not run into the daily rate limit, but if you had more data it might take several runs over several days to get your database to have all of the geocoded data for your input.

Once you have some data loaded into *geodata.sqlite*, you can visualize the data using the *geodump.py* program. This program reads the database and writes the file *where.js* with the location, latitude, and longitude in the form of executable JavaScript code.

A run of the *geodump.py* program is as follows:

```
Northeastern University, ... Boston, MA 02115, USA 42.3396998 -71.08975
Bradley University, 1501 ... Peoria, IL 61625, USA 40.6963857 -89.6160811
...
Technion, Viazman 87, Kesalsaba, 32000, Israel 32.7775 35.0216667
Monash University Clayton ... VIC 3800, Australia -37.9152113 145.134682
Kokshetau, Kazakhstan 53.2833333 69.3833333
...
12 records written to where.js
Open where.html to view the data in a browser
```

The file *where.html* consists of HTML and JavaScript to visualize a Google map. It reads the most recent data in *where.js* to get the data to be visualized. Here is the format of the *where.js* file:

```
myData = [
[42.3396998,-71.08975, 'Northeastern Uni ... Boston, MA 02115'],
[40.6963857,-89.6160811, 'Bradley University, ... Peoria, IL 61625, USA'],
[32.7775,35.0216667, 'Technion, Viazman 87, Kesalsaba, 32000, Israel'],
   ...
];
```

This is a JavaScript variable that contains a list of lists. The syntax for JavaScript list constants is very similar to Python, so the syntax should be familiar to you.

Simply open *where.html* in a browser to see the locations. You can hover over each map pin to find the location that the geocoding API returned for the user-entered input. If you cannot see any data when you open the *where.html* file, you might want to check the JavaScript or developer console for your browser.

16.2 Visualizing networks and interconnections

In this application, we will perform some of the functions of a search engine. We will first spider a small subset of the web and run a simplified version of the Google page rank algorithm to determine which pages are most highly connected, and then visualize the page rank and connectivity of our small corner of the web. We will use the D3 JavaScript visualization library http://d3js.org/ to produce the visualization output.

You can download and extract this application from:

Figure 16.2: A Page Ranking

www.py4e.com/code3/pagerank.zip

The first program (*spider.py*) program crawls a web site and pulls a series of pages into the database (*spider.sqlite*), recording the links between pages. You can restart the process at any time by removing the *spider.sqlite* file and rerunning *spider.py*.

```
Enter web url or enter: http://www.dr-chuck.com/
['http://www.dr-chuck.com']
How many pages:2
1 http://www.dr-chuck.com/ 12
2 http://www.dr-chuck.com/csev-blog/ 57
How many pages:
```

In this sample run, we told it to crawl a website and retrieve two pages. If you restart the program and tell it to crawl more pages, it will not re-crawl any pages already in the database. Upon restart it goes to a random non-crawled page and starts there. So each successive run of *spider.py* is additive.

```
Enter web url or enter: http://www.dr-chuck.com/
['http://www.dr-chuck.com']
How many pages:3
3 http://www.dr-chuck.com/csev-blog 57
4 http://www.dr-chuck.com/dr-chuck/resume/speaking.htm 1
5 http://www.dr-chuck.com/dr-chuck/resume/index.htm 13
How many pages:
```

You can have multiple starting points in the same database—within the program,

these are called "webs". The spider chooses randomly amongst all non-visited links across all the webs as the next page to spider.

If you want to dump the contents of the *spider.sqlite* file, you can run *spdump.py* as follows:

```
(5, None, 1.0, 3, 'http://www.dr-chuck.com/csev-blog')
(3, None, 1.0, 4, 'http://www.dr-chuck.com/dr-chuck/resume/speaking.htm')
(1, None, 1.0, 2, 'http://www.dr-chuck.com/csev-blog/')
(1, None, 1.0, 5, 'http://www.dr-chuck.com/dr-chuck/resume/index.htm')
4 rows.
```

This shows the number of incoming links, the old page rank, the new page rank, the id of the page, and the url of the page. The *spdump.py* program only shows pages that have at least one incoming link to them.

Once you have a few pages in the database, you can run page rank on the pages using the *sprank.py* program. You simply tell it how many page rank iterations to run.

```
How many iterations:2
1 0.546848992536
2 0.226714939664
[(1, 0.559), (2, 0.659), (3, 0.985), (4, 2.135), (5, 0.659)]
```

You can dump the database again to see that page rank has been updated:

```
(5, 1.0, 0.985, 3, 'http://www.dr-chuck.com/csev-blog')
(3, 1.0, 2.135, 4, 'http://www.dr-chuck.com/dr-chuck/resume/speaking.htm')
(1, 1.0, 0.659, 2, 'http://www.dr-chuck.com/csev-blog/')
(1, 1.0, 0.659, 5, 'http://www.dr-chuck.com/dr-chuck/resume/index.htm')
4 rows.
```

You can run *sprank.py* as many times as you like and it will simply refine the page rank each time you run it. You can even run *sprank.py* a few times and then go spider a few more pages sith *spider.py* and then run *sprank.py* to reconverge the page rank values. A search engine usually runs both the crawling and ranking programs all the time.

If you want to restart the page rank calculations without respidering the web pages, you can use *spreset.py* and then restart *sprank.py*.

```
How many iterations:50
1 0.546848992536
2 0.226714939664
3 0.0659516187242
4 0.0244199333
5 0.0102096489546
6 0.00610244329379
...
42 0.000109076928206
43 9.91987599002e-05
```

```
44 9.02151706798e-05
45 8.20451504471e-05
46 7.46150183837e-05
47 6.7857770908e-05
48 6.17124694224e-05
49 5.61236959327e-05
50 5.10410499467e-05
[(512, 0.0296), (1, 12.79), (2, 28.93), (3, 6.808), (4, 13.46)]
```

For each iteration of the page rank algorithm it prints the average change in page rank per page. The network initially is quite unbalanced and so the individual page rank values change wildly between iterations. But in a few short iterations, the page rank converges. You should run *sprank.py* long enough that the page rank values converge.

If you want to visualize the current top pages in terms of page rank, run *spjson.py* to read the database and write the data for the most highly linked pages in JSON format to be viewed in a web browser.

```
Creating JSON output on spider.json...
How many nodes? 30
Open force.html in a browser to view the visualization
```

You can view this data by opening the file *force.html* in your web browser. This shows an automatic layout of the nodes and links. You can click and drag any node and you can also double-click on a node to find the URL that is represented by the node.

If you rerun the other utilities, rerun *spjson.py* and press refresh in the browser to get the new data from *spider.json*.

16.3 Visualizing mail data

Up to this point in the book, you have become quite familiar with our *mbox-short.txt* and *mbox.txt* data files. Now it is time to take our analysis of email data to the next level.

In the real world, sometimes you have to pull down mail data from servers. That might take quite some time and the data might be inconsistent, error-filled, and need a lot of cleanup or adjustment. In this section, we work with an application that is the most complex so far and pull down nearly a gigabyte of data and visualize it.

You can download this application from:

www.py4e.com/code3/gmane.zip

We will be using data from a free email list archiving service called www.gmane.org. This service is very popular with open source projects because it provides a nice searchable archive of their email activity. They also have a very liberal policy regarding accessing their data through their API. They have no rate limits, but ask that you don't overload their service and take only the data you need. You can read gmane's terms and conditions at this page:

Figure 16.3: A Word Cloud from the Sakai Developer List

http://gmane.org/export.php

It is very important that you make use of the gmane.org data responsibly by adding delays to your access of their services and spreading long-running jobs over a longer period of time. Do not abuse this free service and ruin it for the rest of us.

When the Sakai email data was spidered using this software, it produced nearly a Gigabyte of data and took a number of runs on several days. The file *README.txt* in the above ZIP may have instructions as to how you can download a pre-spidered copy of the *content.sqlite* file for a majority of the Sakai email corpus so you don't have to spider for five days just to run the programs. If you download the pre-spidered content, you should still run the spidering process to catch up with more recent messages.

The first step is to spider the gmane repository. The base URL is hard-coded in the *gmane.py* and is hard-coded to the Sakai developer list. You can spider another repository by changing that base url. Make sure to delete the *content.sqlite* file if you switch the base url.

The *gmane.py* file operates as a responsible caching spider in that it runs slowly and retrieves one mail message per second so as to avoid getting throttled by gmane. It stores all of its data in a database and can be interrupted and restarted as often as needed. It may take many hours to pull all the data down. So you may need to restart several times.

Here is a run of *gmane.py* retrieving the last five messages of the Sakai developer list:

```
How many messages:10
```

```
http://download.gmane.org/gmane.comp.cms.sakai.devel/51410/51411 9460
    nealcaidin@sakaifoundation.org 2013-04-05 re: [building ...
http://download.gmane.org/gmane.comp.cms.sakai.devel/51411/51412 3379
    samuelgutierrezjimenez@gmail.com 2013-04-06 re: [building ...
http://download.gmane.org/gmane.comp.cms.sakai.devel/51412/51413 9903
    da1@vt.edu 2013-04-05 [building sakai] melete 2.9 oracle ...
http://download.gmane.org/gmane.comp.cms.sakai.devel/51413/51414 349265
    m.shedid@elraed-it.com 2013-04-07 [building sakai] ...
http://download.gmane.org/gmane.comp.cms.sakai.devel/51414/51415 3481
    samuelgutierrezjimenez@gmail.com 2013-04-07 re: ...
http://download.gmane.org/gmane.comp.cms.sakai.devel/51415/51416 0

Does not start with From
```

The program scans *content.sqlite* from one up to the first message number not already spidered and starts spidering at that message. It continues spidering until it has spidered the desired number of messages or it reaches a page that does not appear to be a properly formatted message.

Sometimes gmane.org is missing a message. Perhaps administrators can delete messages or perhaps they get lost. If your spider stops, and it seems it has hit a missing message, go into the SQLite Manager and add a row with the missing id leaving all the other fields blank and restart *gmane.py*. This will unstick the spidering process and allow it to continue. These empty messages will be ignored in the next phase of the process.

One nice thing is that once you have spidered all of the messages and have them in *content.sqlite*, you can run *gmane.py* again to get new messages as they are sent to the list.

The *content.sqlite* data is pretty raw, with an inefficient data model, and not compressed. This is intentional as it allows you to look at *content.sqlite* in the SQLite Manager to debug problems with the spidering process. It would be a bad idea to run any queries against this database, as they would be quite slow.

The second process is to run the program *gmodel.py*. This program reads the raw data from *content.sqlite* and produces a cleaned-up and well-modeled version of the data in the file *index.sqlite*. This file will be much smaller (often 10X smaller) than *content.sqlite* because it also compresses the header and body text.

Each time *gmodel.py* runs it deletes and rebuilds *index.sqlite*, allowing you to adjust its parameters and edit the mapping tables in *content.sqlite* to tweak the data cleaning process. This is a sample run of *gmodel.py*. It prints a line out each time 250 mail messages are processed so you can see some progress happening, as this program may run for a while processing nearly a Gigabyte of mail data.

```
Loaded allsenders 1588 and mapping 28 dns mapping 1
1 2005-12-08T23:34:30-06:00 ggolden22@mac.com
251 2005-12-22T10:03:20-08:00 tpamsler@ucdavis.edu
501 2006-01-12T11:17:34-05:00 lance@indiana.edu
751 2006-01-24T11:13:28-08:00 vrajgopalan@ucmerced.edu
...
```

The *gmodel.py* program handles a number of data cleaning tasks.

Domain names are truncated to two levels for .com, .org, .edu, and .net. Other domain names are truncated to three levels. So si.umich.edu becomes umich.edu and caret.cam.ac.uk becomes cam.ac.uk. Email addresses are also forced to lower case, and some of the @gmane.org address like the following

```
arwhyte-63aXycvo3TyHXe+LvDLADg@public.gmane.org
```

are converted to the real address whenever there is a matching real email address elsewhere in the message corpus.

In the *mapping.sqlite* database there are two tables that allow you to map both domain names and individual email addresses that change over the lifetime of the email list. For example, Steve Githens used the following email addresses as he changed jobs over the life of the Sakai developer list:

```
s-githens@northwestern.edu
sgithens@cam.ac.uk
swgithen@mtu.edu
```

We can add two entries to the Mapping table in *mapping.sqlite* so *gmodel.py* will map all three to one address:

```
s-githens@northwestern.edu ->  swgithen@mtu.edu
sgithens@cam.ac.uk -> swgithen@mtu.edu
```

You can also make similar entries in the DNSMapping table if there are multiple DNS names you want mapped to a single DNS. The following mapping was added to the Sakai data:

```
iupui.edu -> indiana.edu
```

so all the accounts from the various Indiana University campuses are tracked together.

You can rerun the *gmodel.py* over and over as you look at the data, and add mappings to make the data cleaner and cleaner. When you are done, you will have a nicely indexed version of the email in *index.sqlite*. This is the file to use to do data analysis. With this file, data analysis will be really quick.

The first, simplest data analysis is to determine "who sent the most mail?" and "which organization sent the most mail"? This is done using *gbasic.py*:

```
How many to dump? 5
Loaded messages= 51330 subjects= 25033 senders= 1584

Top 5 Email list participants
steve.swinsburg@gmail.com 2657
azeckoski@unicon.net 1742
ieb@tfd.co.uk 1591
csev@umich.edu 1304
david.horwitz@uct.ac.za 1184
```

```
Top 5 Email list organizations
gmail.com 7339
umich.edu 6243
uct.ac.za 2451
indiana.edu 2258
unicon.net 2055
```

Note how much more quickly *gbasic.py* runs compared to *gmane.py* or even *gmodel.py*. They are all working on the same data, but *gbasic.py* is using the compressed and normalized data in *index.sqlite*. If you have a lot of data to manage, a multistep process like the one in this application may take a little longer to develop, but will save you a lot of time when you really start to explore and visualize your data.

You can produce a simple visualization of the word frequency in the subject lines in the file *gword.py*:

```
Range of counts: 33229 129
Output written to gword.js
```

This produces the file *gword.js* which you can visualize using *gword.htm* to produce a word cloud similar to the one at the beginning of this section.

A second visualization is produced by *gline.py*. It computes email participation by organizations over time.

```
Loaded messages= 51330 subjects= 25033 senders= 1584
Top 10 Oranizations
['gmail.com', 'umich.edu', 'uct.ac.za', 'indiana.edu',
'unicon.net', 'tfd.co.uk', 'berkeley.edu', 'longsight.com',
'stanford.edu', 'ox.ac.uk']
Output written to gline.js
```

Its output is written to *gline.js* which is visualized using *gline.htm*.

This is a relatively complex and sophisticated application and has features to do some real data retrieval, cleaning, and visualization.

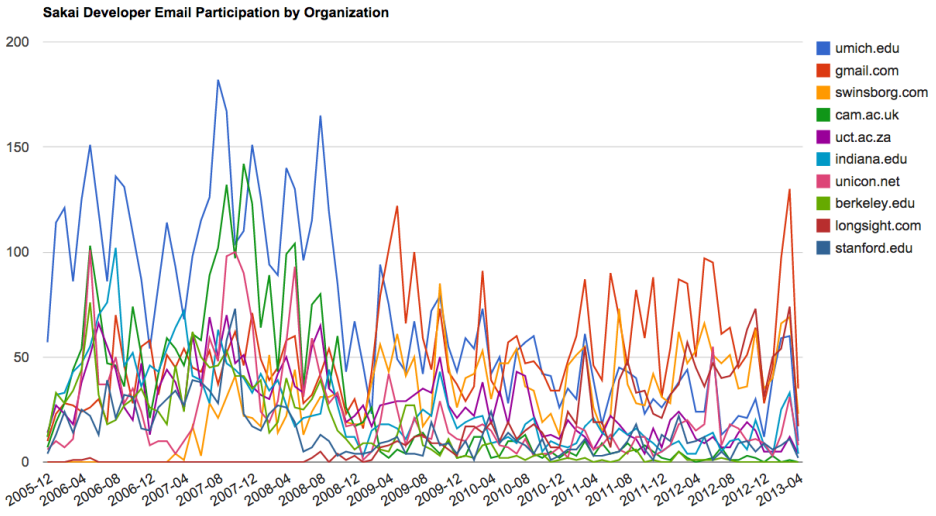

Figure 16.4: Sakai Mail Activity by Organization

Appendix A

Colaboraciones

A.1 Contributor List for Python para todos

Juan Carlos Perez Castellanos, Juan Dougnac, Daniel Merino Echeverría, Jaime Bermeo Ramírez, and Fernando Tardío.

A.2 Contributor List for Python for Everybody

Elliott Hauser, Stephen Catto, Sue Blumenberg, Tamara Brunnock, Mihaela Mack, Chris Kolosiwsky, Dustin Farley, Jens Leerssen, Naveen KT, Mirza Ibrahimovic, Naveen (@togarnk), Zhou Fangyi, Alistair Walsh, Erica Brody, Jih-Sheng Huang, Louis Luangkesorn, and Michael Fudge

You can see contribution details at:

https://github.com/csev/py4e/graphs/contributors https://github.com/csev-es/py4e/graphs/contributors

A.3 Lista de colaboradores de "Python para Informáticos"

Bruce Shields for copy editing early drafts, Sarah Hegge, Steven Cherry, Sarah Kathleen Barbarow, Andrea Parker, Radaphat Chongthammakun, Megan Hixon, Kirby Urner, Sarah Kathleen Barbrow, Katie Kujala, Noah Botimer, Emily Alinder, Mark Thompson-Kular, James Perry, Eric Hofer, Eytan Adar, Peter Robinson, Deborah J. Nelson, Jonathan C. Anthony, Eden Rassette, Jeannette Schroeder, Justin Feezell, Chuanqi Li, Gerald Gordinier, Gavin Thomas Strassel, Ryan Clement, Alissa Talley, Caitlin Holman, Yong-Mi Kim, Karen Stover, Cherie Edmonds, Maria Seiferle, Romer Kristi D. Aranas (RK), Grant Boyer, and Hedemarrie Dussan.

A.4 Prefacio para "Think Python"

La extraña historia de "Think Python"

(Allen B. Downey)

En Enero de 1999, estaba preparándome para enseñar una clase de introducción a la programación en Java. Había impartido el curso tres veces y me estaba frustrando. La tasa de fracaso en la clase era demasiado alta e, incluso aquellos estudiantes que aprobaban, lo hacían con un nivel general de conocimientos demasiado bajo.

Me di cuenta de que uno de los problemas eran los libros. Eran demasiado grandes, con demasiados detalles innecesarios de Java, y sin suficiente orientación de alto nivel sobre cómo programar. Y todos ellos sufrían el mismo efecto trampilla: comenzaban siendo muy fáciles, avanzaban poco a poco, y en algún lugar alrededor del Capítulo 5 el suelo desaparecía. Los estudiantes recibían demasiado material nuevo demasiado rápido, y yo tenía que pasar el resto del semestre recogiendo los pedazos.

Dos semanas antes del primer día de clase, decidí escribir mi propio libro. Mis objetivos eran:

- Hacerlo breve. Para los estudiantes es mejor leer 10 páginas que no tener que leer 50.
- Ser cuidadoso con el vocabulario. Intenté minimizar la jerga y definir cada término al usarlo la primera vez.
- Construir poco a poco. Para evitar las trampillas, tomé los temas más difíciles y los dividí en una serie de pasos más pequeños.
- Enfocarlo a la programación, no al lenguaje de programación. Incluí el subconjunto de Java mínimo imprescindible y excluí el resto.

Necesitaba un título, de modo que elegí caprichosamente *How to Think Like a Computer Scientist* (Cómo pensar como un informático).

Mi primera versión era tosca, pero funcionaba. Los estudiantes la leían, y comprendían lo suficiente como para que pudiera emplear el tiempo de clase en tratar los temas difíciles, los temas interesantes y (lo más importante) dejar a los estudiantes practicar.

Publiqué el libro bajo Licencia de Documentación Libre GNU (`GNU Free Documentation License`), que permite a los usuarios copiar, modificar y distribuir el libro.

Lo que sucedió después es la parte divertida. Jeff Elkner, un profesor de escuela secundaria de Virginia, adoptó mi libro y lo tradujo para Python. Me envió una copia de su traducción, y tuve la inusual experiencia de aprender Python leyendo mi propio libro.

Jeff y yo revisamos el libro, incorporamos un caso práctico realizado por Chriss Meyers, y en 2001 publicamos *How to Think Like a Computer Scientist: Learning with Python* (Cómo pensar como un informático: Aprendiendo con Python),

también bajo Licencia de Documentación Libre GNU (`GNU Free Documentation License`). Publiqué el libro como `Green Tea Press` y comencé a vender copias en papel a través de Amazon.com y librerías universitarias. Hay otros libros de `Green Tea Press` disponibles en greenteapress.com.

En 2003, comencé a impartir clases en el Olin College y tuve que enseñar Python por primera vez. El contraste con Java fue notable. Los estudiantes se tenían que esforzar menos, aprendían más, trabajaban en proyectos más interesantes, y en general se divertían mucho más.

Durante los últimos cinco años he continuado desarrollando el libro, corrigiendo errores, mejorando algunos de los ejemplos y añadiendo material, especialmente ejercicios. En 2008 empecé a trabajar en una revisión general—al mismo tiempo, se puso en contacto conmigo un editor de la Cambridge University Press interesado en publicar la siguiente edición. ¡Qué oportuno!

Espero que disfrutes con este libro, y que te ayude a aprender a programar y a pensar, al menos un poquito, como un informático.

Agradecimientos por "Think Python"

(Allen B. Downey)

Lo primero y más importante, mi agradecimiento a Jeff Elkner por haber traducido mi libro de Java a Python, ya que eso fue lo que hizo comenzar este proyecto y me introdujo en el que se ha convertido en mi lenguaje de programación favorito.

Quiero dar las gracias también a Chris Meyers, que ha contribuído en varias secciones de *How to Think Like a Computer Scientist*.

Y agradezco a la `Free Software Foundation` (Fundación de Software Libre) por haber desarrollado la `GNU Free Documentation License`, que ha ayudado a que mi colaboración con Jeff y Chris fuera posible.

Tambien quiero agradecer a los editores de Lulu que trabajaron en *How to Think Like a Computer Scientist*.

Doy las gracias a todos los estudiantes que trabajaron con las primeras versiones de este libro y a todos los colaboradores (listados en un Apéndice) que han enviado correcciones y sugerencias.

Y quiero dar las gracias a mi mujer, Lisa, por su trabajo en este libro, en `Green Tea Press`, y por todo lo demás, también.

Allen B. Downey
Needham MA
Allen Downey es un Profesor Asociado de Informática en el `Franklin W. Olin College of Engineering`.

A.5 Lista de colaboradores de "Think Python"

(Allen B. Downey)

Más de 100 lectores perspicaces y atentos me han enviado sugerencias y correcciones a lo largo de los últimos años. Su contribución y entusiasmo por este proyecto han resultado de gran ayuda.

Para conocer los detalles sobre la naturaleza de cada una de las contribuciones de estas personas, mira en el texto de "Think Python".

Lloyd Hugh Allen, Yvon Boulianne, Fred Bremmer, Jonah Cohen, Michael Conlon, Benoit Girard, Courtney Gleason and Katherine Smith, Lee Harr, James Kaylin, David Kershaw, Eddie Lam, Man-Yong Lee, David Mayo, Chris McAloon, Matthew J. Moelter, Simon Dicon Montford, John Ouzts, Kevin Parks, David Pool, Michael Schmitt, Robin Shaw, Paul Sleigh, Craig T. Snydal, Ian Thomas, Keith Verheyden, Peter Winstanley, Chris Wrobel, Moshe Zadka, Christoph Zwerschke, James Mayer, Hayden McAfee, Angel Arnal, Tauhidul Hoque and Lex Berezhny, Dr. Michele Alzetta, Andy Mitchell, Kalin Harvey, Christopher P. Smith, David Hutchins, Gregor Lingl, Julie Peters, Florin Oprina, D. J. Webre, Ken, Ivo Wever, Curtis Yanko, Ben Logan, Jason Armstrong, Louis Cordier, Brian Cain, Rob Black, Jean-Philippe Rey at Ecole Centrale Paris, Jason Mader at George Washington University made a number Jan Gundtofte-Bruun, Abel David and Alexis Dinno, Charles Thayer, Roger Sperberg, Sam Bull, Andrew Cheung, C. Corey Capel, Alessandra, Wim Champagne, Douglas Wright, Jared Spindor, Lin Peiheng, Ray Hagtvedt, Torsten Hübsch, Inga Petuhhov, Arne Babenhauserheide, Mark E. Casida, Scott Tyler, Gordon Shephard, Andrew Turner, Adam Hobart, Daryl Hammond and Sarah Zimmerman, George Sass, Brian Bingham, Leah Engelbert-Fenton, Joe Funke, Chao-chao Chen, Jeff Paine, Lubos Pintes, Gregg Lind and Abigail Heithoff, Max Hailperin, Chotipat Pornavalai, Stanislaw Antol, Eric Pashman, Miguel Azevedo, Jianhua Liu, Nick King, Martin Zuther, Adam Zimmerman, Ratnakar Tiwari, Anurag Goel, Kelli Kratzer, Mark Griffiths, Roydan Ongie, Patryk Wolowiec, Mark Chonofsky, Russell Coleman, Wei Huang, Karen Barber, Nam Nguyen, Stéphane Morin, Fernando Tardío, y Paul Stoop.

Appendix B

Detalles del Copyright

Hubiéramos preferido publicar el libro bajo la licencia CC-BY-SA, que es menos restrictiva. Pero, por desgracia, existen unas pocas organizaciones sin escrúpulos que buscan y encuentran libros con licencias libres y luego los publican y venden copias virtualmente idénticas de esos libros en un servició de impresión bajo demanda, como Lulu o CreateSpace. CreateSpace ha añadido (afortunadamente) una norma que da preferencia a los deseos del titular real del copyright sobre un titular sin derechos que pretenda publicar un trabajo con licencia libre. Por desgracia, existen muchos servicios de impresión bajo demanda y muy pocos tienen unas normas tan consideradas como CreateSpace.

Con pesar, he añadido el elemento NC a la licencia de este libro para poder recurrir en caso de que alguien intente clonar el libro y venderlo comercialmente. Por desgracia, al añadir NC se limitan otros usos de este material que sí me gustaría permitir. De modo que he añadido esta sección del documento para describir aquellas situaciones específicas de uso del material de este libro que algunos podrían considerar comerciales y para las cuales doy mi permiso por adelantado.

- Si imprimes un número limitado de copias de todo o parte de este libro para usar en un curso (es decir, como material para el curso), entonces tienes concedida licencia CC-BY para usar este material para ese propósito.
- Si eres profesor de una universidad, traduces este libro a un idioma distinto del inglés y lo utilizas para enseñar, puedes contactar conmigo y te concederé una licencia CC-BY-SA para estos materiales, con respecto a la publicación de tu traducción. En particular, tendrás permiso para vender comercialmente el libro traducido resultante.

Si estás interesado en traducir el libro, puedes ponerte en contacto conmigo para asegurarnos de que tienes todos los materiales relacionados con el curso, para que puedas traducirlos también.

Por supuesto, estás invitado a ponerte en contacto conmigo y pedirme permiso si estas clausulas no son suficientes. En cualquier caso, el permiso para reutilizar

y remezclar este material está concedido siempre que se produzca un claro valor añadido o beneficio para los estudiantes o profesores que se unan como resultado del nuevo trabajo.

Charles Severance
www.dr-chuck.com
Ann Arbor, MI, USA
9 de Septiembre de 2013

Index