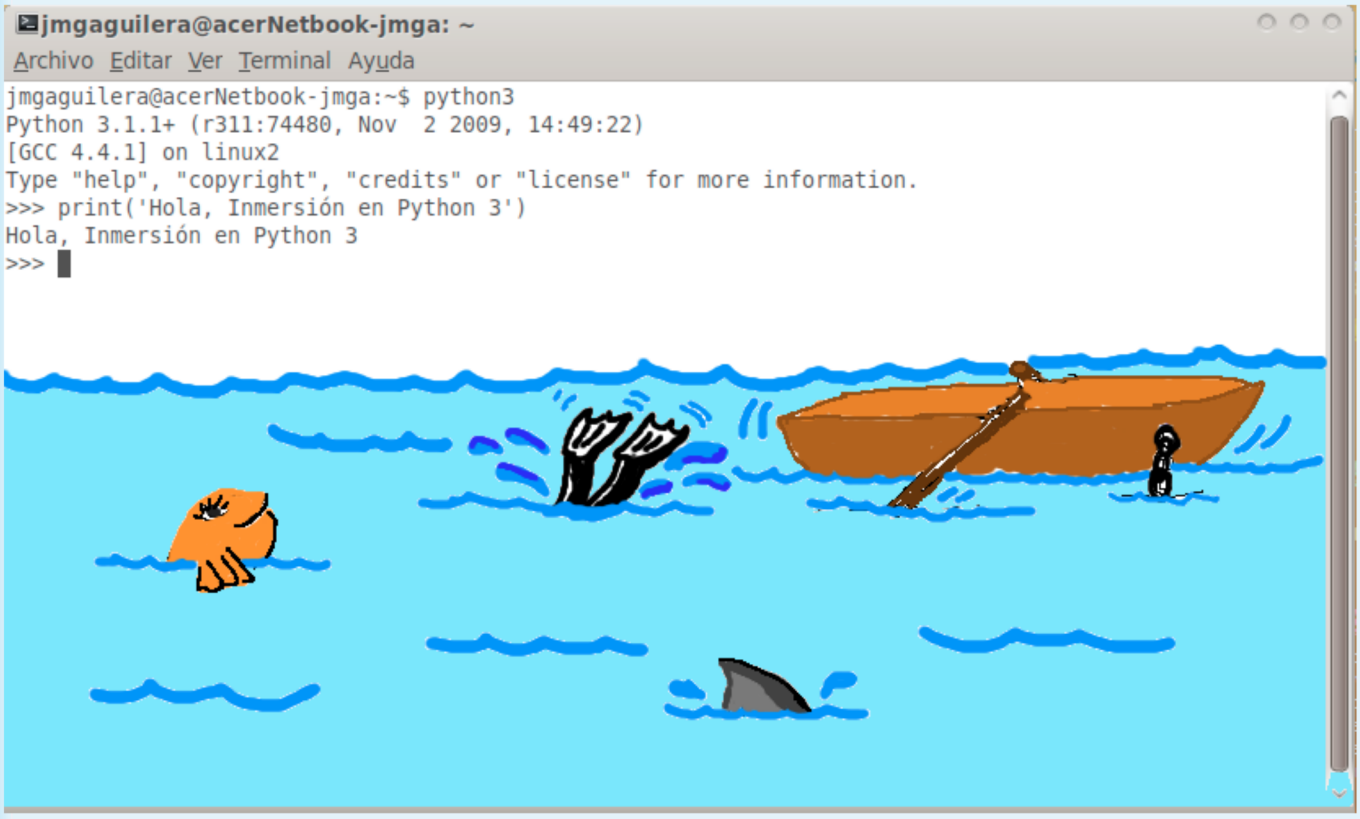
Inmersión en Python 3

escrito por Mark Pilgrim
traducido por José Miguel González Aguilera



español: http://inmersionenpython3.codegoogle.com
original inglés: http://www.diveintopython3.org

Inmersión en Python 3
por Mark Pilgrim

Traducción al español: José Miguel González Aguilera

Website de la traducción: http://code.google.com/p/inmersionenpython3

Agradecimientos del Traductor:
A Mark Pilgrim.
A Nieves, Alba y a Miguel.

Licencia:

Compartir bajo la misma licencia. Si altera o transforma esta obra, o genera una obra derivada, sólo puede distribuir la obra generada bajo una licencia idéntica a ésta.

Al reutilizar o distribuir la obra, tiene que dejar bien claro los términos de la licencia de esta obra.

Alguna de las condiciones puede no aplicarse si se obtiene el permiso del titular de los derechos de esta obra.

Nada en esta licencia menoscaba o restringe los derechos morales del autor.

Capítulo -1

Novedades de "Inmersión en Python 3"

"¿No es de aquí de donde venimos?"
—Pink Floyd, The Wall

-1.1. Alias "Bajo el nivel del mar"

Posiblemente hayas leído el libro original *Dive into Python* y puede que hasta lo hayas comprado. (Si es el caso: ¡gracias!) Ya conoces bastante el lenguaje Python. Estás preparado para dar el salto a Python 3. ... Si lo dicho es cierto, sigue leyendo. (Si no es así, tal vez sea mejor que comiences desde el principio en el capítulo **??**).

Python 3 viene con un script denominado 2to3. Aprende a usarlo y a quererlo. El apéndice **??** es una referencia sobre las cosas que la herramienta 2to3 puede arreglar automáticamente en la conversión del código de la versión 2 a la 3 de python. Puesto que muchas cosas son cambios de sintaxis, una buena forma de comenzar es aprender estas diferencias. Por ejemplo: print ahora es una función...

El caso de estudio del capítulo **??** documenta mi esfuerzo (¡al fin cumplido con éxito!) de convertir una librería real de Python 2 a Python 3. Puede servirte o no. Es un ejemplo complejo de entender puesto que en primer lugar tienes que comprender algo el funcionamiento de la librería, de forma que puedas entender lo que deja de funcionar y como lo arreglé. Mucho de lo que se rompió al pasar a la versión 3 de Python fue por causa de las cadenas. Por cierto, hablando de cadenas...

Cadenas. ¡Uff!. Por dónde podría empezar. Python 2 tenía "cadenas" y "cadenas unicode". Python 3 tiene "bytes" y "cadenas". Lo que significa que todas las

cadenas ahora son unicode, y si quieres trabajar con un puñado de bytes tienes que usar el tipo **bold** bytes.

Python 3 nunca convertirá implícitamente entre cadenas y bytes, por lo que si no estas seguro de lo que contiene una variable en un momento dado, el código seguro que fallará en algún momento. Lee el capítulo 4 sobre cadenas para conocer los detalles.

La división entre "bytes" y "cadenas" surgirá en diversas partes del libro:

1. En el capítulo 11 dedicado a los ficheros, aprenderás la diferencia entre leer ficheros en modo *binario* o en modo *texto*. La lectura (y escritura) de ficheros en modo *texto* requiere que se utilice el parámetro encoding. Existen métodos que cuentan los caracteres de un fichero y métodos que cuentan bytes. Si el código asume que un carácter es igual a un byte, no funcionará cuando el fichero contenga caracteres multibyte[1].

2. En el capítulo **??** dedicado a los servicios web n http, se muestra el módulo httplib2 que lee cabeceras y datos de HTTP. Las cabeceras se obtienen como cadenas, pero el contenido del cuerpo se obtiene como bytes.

3. En el capítulo **??** aprenderás el motivo por el que el módulo pickle de Python 3 define un formato de datos nuevo que es incompatible con Python 2 (Pista: Se debe a los bytes y cadenas). También afecta al módulo JSON, que no es capaz de manejar el tipo bytes. Te enseñaré como salvar este escollo.

4. En el capítulo **??** sobre la conversión de la librería chardet a Python 3 se verá que la mayor parte de los problemas de conversión provienen de los bytes y cadenas.

Incluso aunque no tengas interés en Unicode, ¡que tendrás!, querrás leer sobre el formateo de cadenas en Python 3 en el capítulo **??**, que es completamente diferente a Python 2.

Los iteradores están en todas partes en Python 3, y ahora los entiendo mucho mejor que hace cinco años cuando escribí "Inmersión en Python". Debes comprenderlos tú también, puesto que muchas funciones que anteriormente retornaban listas ahora, en Python 3, devuelven iteradores. Como mínimo, deberías leer la segunda parte del capítulo **??** dedicado a los iteradores y la segunda parte del capítulo **??** sobre el uso avanzado de los iteradores.

Por petición popular, he añadido el apéndice **??** sobre nombres de método especiales que guarda cierta similitud con el apartado similar de la documentación oficial de Python 3 pero con cierta ironía.

[1] En unicode muchos caracteres se representan utilizando más de un byte

Cuando estaba escribiendo "Inmersión en Python" todas las librerías de XML disponibles eran bastante malas. Entonces Fedrik Lundh escribió **bold** ElementTree, que es todo lo contrario a lo existente anteriormente. Los dioses de Python, actuando inteligentemente, incorporaron ElementTree a la librería estándar. Ahora esta librería es el fundamento del capítulo 12 sobre XML. Los viejos métodos para recorrer XML están aún disponibles, pero deberías evitarlos, ¡apestan!

Algo que es también nuevo —no en el lenguaje, pero sí en la comunidad— es la creación de repositorios de código como el índice de paquetes de python (PyPI). Python dispone de utilidades para empaquetar el código en formatos estándares y distribuirlos en PyPI. Lee el capítulo **??** sobre cómo empaquetar librerías en Python.

Capítulo 0

Instalación de Python

Nivel de dificultad:◆ ◇ ◇ ◇ ◇

"Tempora mutantur nos et mutamur in illis"
(Los tiempos cambian, y nosotros cambiamos con ellos)
—antiguo proverbio romano

0.1. Inmersión

Bienvenido a Python 3. ¡Vamos a mojarnos! En este capítulo, vas a instalar la versión de Python adecuada para ti.

0.2. ¿Cuál es la versión adecuada para ti?

Lo primero que necesitas hacer es instalar Python 3.

Si estás utilizando una sesión en un servidor remoto (posiblemente a través de Internet), el administrador del servidor puede que ya lo haya instalado por ti. Si estás utilizando Linux[1] en casa, puede que también lo tengas ya instalado, aunque actualmente[2] la mayor parte de las distribuciones de Linux vienen con Python 2 instalado (como verás en este capítulo, puedes tener simultáneamente más de una versión de Python en tu ordenador sin problemas). En los Mac OS X se incluye una versión de línea de comando de Python 2, pero no Python 3. Microsoft Windows no

[1]Nota del Traductor: El nombre correcto del sistema operativo Linux es GNU/Linux, no obstante, por comodidad, en este libro se utilizará únicamente Linux para mayor comodidad

[2]año 2009

trae ninguna versión de Python. Pero ¡no te preocupes! siempre puedes instalarlo tú mismo, tengas el sistema operativo que tengas.

La forma más sencilla para comprobar si tienes instalado Python 3 en tu sistema Linux o Mac OS X es abrir un terminal de línea de comandos. Para ello debes hacer lo siguiente:

- Si estás en Linux, busca en el menú de Aplicaciones un programa denominado terminal (puede estar en un submenú, posiblemente Accesorios o Sistema).
- Si estás en Mac OS X, existe una aplicación que se llama Terminal.app en la carpeta /Aplicaciones/Utilidades/.

Una vez te encuentres en la línea de comando[3], teclea python3 (en minúsculas y sin espacios) y observa lo que sucede. En mi sistema Linux, Python 3 ya está instalado, por lo que el resultado de ejecutar este comando hace que el terminal entre en la *consola*[4] *interactiva de Python*.

```
jmgaguilera@acerNetbook-jmga:~$ python3
Python 3.0.1+ (r301:69556, Apr 15 2009, 15:59:22)
[GCC 4.3.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

(Para salir de la consola interactiva de Python escribe exit() y pulsa la tecla INTRO.)

Al ejecutar esta misma sentencia python3 en un ordenador Linux que no tenga instalado Python 3 el mensaje que se obtendrá será parecido al siguiente:

```
jmgaguilera@acerNetbook-jmga:~$ python3
bash: python3: orden no encontrada
jmgaguilera@acerNetbook-jmga:~$ python3
```

Bueno, volviendo ahora a la pregunta sobre cuál es la versión de Python 3 apropiada para ti, queda claro que es aquella que se ejecute en el ordenador que tengas.

Para conocer cómo instalar Python 3, continúa leyendo en el apartado que corresponda a tu sistema operativo.

[3] También conocido como el "prompt"
[4] En inglés "shell"

0.3. Instalación en Microsoft Windows

Windows se ejecuta actualmente en dos plataformas diferentes: 32 y 64 bits. Asimismo, existen diferentes *versiones* de Windows —XP, Vista, Windows 7— y Python 3 funciona en todas ellas. Es más importante, con vistas a la instalación, la distinción que existe entre los dos tipos de arquitecturas. Si no sabes de qué tipo es la arquitectura de tu ordenador, lo más probable es que sea de 32 bits.

Visita python.org/download/ para descargar la aplicación de instalación de Python 3 que sea correcta para para la arquitectura de tu ordenador. Las posibilidades serán parecidas a:

- **Python 3.*.* x86 Windows installer** (Windows binary — does not include sources)
- **Python 3.*.* AMD64 Windows installer** (Windows AMD64 binary — does not include sources)

La descarga exacta varía en función de las actualizaciones. Por eso he puesto asteriscos en lugar del número de versión. Deberías instalar siempre la última versión disponible de Python 3.x a menos que tengas alguna razón importante para no hacerlo.

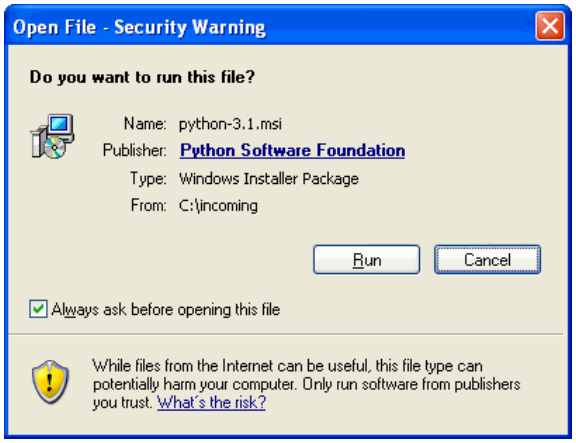


Figura 1: Advertencia al inicio

Cuando la descarga finalize, pulsa (doble click) sobre el fichero .msi que has descargado. Windows mostrará una alerta de seguridad (figura 1) para avisarte de que estás intentando ejecutar un fichero que instalará cosas en tu ordenador. El fichero instalador de Python está firmado electrónicamente por la *Python Software*

Foundation, que es la organización sin ánimo de lucro que supervisa el desarrollo de Python. ¡No aceptes imitaciones!

Pulsa el botón Run o Ejecutar[5] para que se inicie la ejecución del programa instalador de Python.

Figura 2: Tipo de instalación

Lo primero que pide el programa instalador (figura 2) es que le indiques si quieres instalar Python 3 para todos los usuarios del ordenadores o únicamente para ti. Por defecto aparece seleccionada la opción "Instalar para todos los usuarios", que es la mejor opción, a no ser que tengas una buena razón para no hacerlo[6].

Cuando hayas seleccionado la opción deseada, pulsa el botón Next o Siguiente para continuar con la instalación.

Lo siguiente que pedirá el instalador (figura 3) es que le digas el directorio de instalación. El valor por defecto para todas las versiones de Python 3.1.x es C:\Python31\, que es un valor adecuado para la mayoría de los usuarios. Salvo que tengas una razón específica para cambiarlo, como por ejemplo, que mantengas una unidad separada para la instalación de aplicaciones, puedes usar este directorio para instalar Python.

Para cambiar el directorio de instalación, puedes utilizar las opciones de pan-

[5] dependerá del idioma en el que se encuentre tu sistema operativo

[6] Una posible razón por la podrías querer instalarlo únicamente para tu usuario es que estuvieras instalando Python en el ordenador de la empresa y no tengas permisos de administrador en tu cuenta de usuario. Pero en ese caso, ¿qué haces instalando Python sin permiso del administrador de tu empresa? A mí no me metas en problemas, eso es cosa tuya.

talla o, simplemente, teclear el directorio deseado (con el path completo) en la caja de texto.

Figura 3: Directorio de instalación

Puedes instalar Python en el disco duro en el lugar que desees.

Cuando hayas finalizado, pulsa el botón Next o Siguiente para continuar.

Figura 4: Selección de elementos a instalar

La siguiente pantalla (figura 4) parece más compleja, pero en realidad no lo es. Como pasa con otros muchos instaladores, te ofrece la opción de que seleccio-

nes qué cosas concretas quieres instalar. Puedes instalar todos los componentes de Python 3, y si el espacio en disco es justo, puedes excluir ciertos componentes.

- **Registrar las extensiones**. Si seleccionas esta opción, el instalador modificará la configuración de Windows para que te permita ejecutar los scripts[7] de Python con solo hacer doble click sobre el fichero. Esta opción no necesita de espacio en disco, por lo que no tiene mucho sentido no marcarla.
- **Tcl\Tk** es la librería gráfica que utiliza la consola de Python. La usaremos a lo largo de todo el libro, por lo que es muy recomendable que la mantengas entre los componentes a instalar.
- **Documentación** instala un fichero de ayuda que contiene gran parte de la información que se encuentra en docs.python.org. Es recomendable instalar esta opción cuando es previsible que no dispongas de conexión permanente a Internet.
- **Scripts de utilidades**. Estos scripts incluyen diversas utilidades, entre ellas el script 2to3.py sobre el que hablaremos más adelante. Es necesaria si vas a migrar código de Python 2 a Python 3. Si no dispones de código para migrar puedes saltarte esta opción.
- **Suite de pruebas.** Es una colección de scripts que se utilizan para probar el buen funcionamiento del intérprete de Python. En este libro no lo vamos a usar, yo no lo he usado jamás en el largo tiempo que llevo programando en Python. Es totalmente opcional su instalación.

Si no estás seguro de cuando espacio en disco tienes libre, pulsa el botón Disk Usage. El instalador te mostrará las unidades de disco (figura 5) y el espacio libre disponible en cada una de ellas, así como el espacio que quedará después de la instalación.

Cuando termines la comprobación, pulsa el botón OK para volver a la pantalla anterior.

Si decides excluir alguna opción (figura 6), selecciona el botón desplegable que aparece a la izquierda del texto de la opción y selecciona Entire feature will be unavailable. Por ejemplo, si excluyes la suite de pruebas ahorrarás 7908 KBytes de espacio en disco.

Pulsa el botón Next para confirmar tu selección de opciones.

[7] ficheros que contienen sentencias de Python, que normalmente tienen la extensión .py

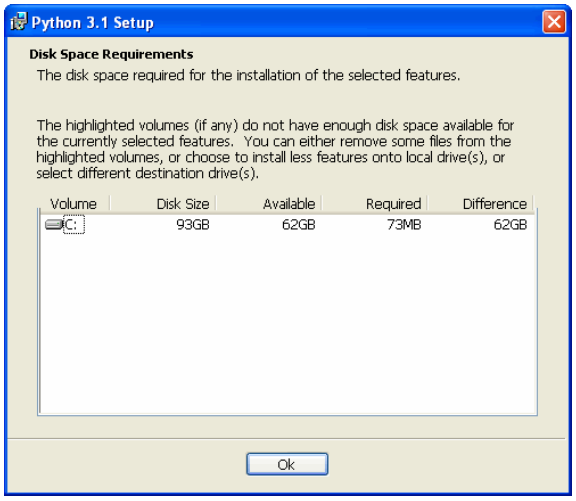


Figura 5: Espacio libre

Figura 6: Excluir una opción

El instalador copiará todos los ficheros (figura 7 al directorio de destino que hayas seleccionado (Suele ser tan rápido, que tuve que probarlo tres veces antes de conseguir sacar una "foto" de la pantalla mostrándolo).

Por último, pulsa el botón Finish para salir del instalador (figura 8).

Si ahora buscas en el menú de Inicio, deberías encontrar un nuevo elemento denominado Python 3.1. Dentro de esta nueva opción de menú encontrarás dos pro-

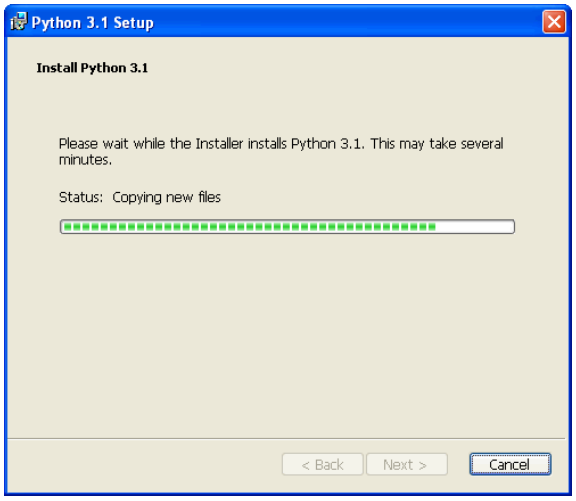


Figura 7: Instalación

Figura 8: Instalación completada

gramas denominados Python e IDLE. Selecciona uno de estos dos elementos para ejecutar la consola interactiva de Python (figura 9).

Continúa en el apartado 0.7

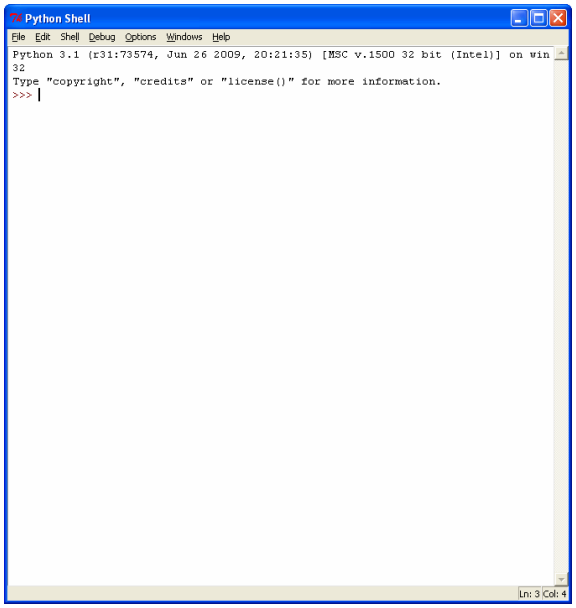


Figura 9: Instalación completada

0.4. Instalación en un Mac OS X

Todos los ordenadores Apple Macintosh modernos utilizan procesadores de Intel[8] Los Macintosh antiguos utilizaban procesadores Power PC. No es necesario que conozcas esta diferencia puesto que únicamente existe un instalador para todos los tipos de Macs.

Visita python.org/download/ para descargar la aplicación de instalación de Python 3 para Mac. Debes buscar un enlace cuyo nombre sea algo así como **Mac Installer Disk Image (3.*.***. El número de versión puede variar, pero asegúrate de descargar una versión de Python 3 y no de Python 2.

Tu navegador debería montar de forma automática esta imagen de disco y abrir una ventana de Finder para mostrarte el contenido de la imagen. Si no fuese así, deberás buscar la imagen de disco en el directorio de descargas y hacer doble click sobre ella para que se cargue. El nombre de la imagen de disco será algo así como python-3-1.dmg. Una vez tengas visible en pantalla el contenido de la imagen de disco (figura 10), podrás observar que contiene varios ficheros de texto (Build.txt, License.txt, ReadMe.txt), y el el fichero del paquete de instalación Python.mpkg.

[8]Como la mayoría de ordenadores con Windows

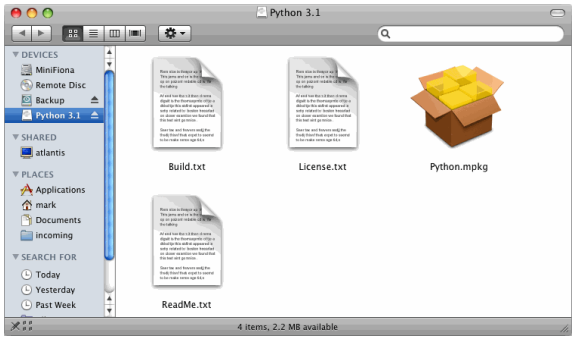


Figura 10: Finder: contenido de la imagen de disco

Haz doble click con el cursor sobre el fichero de instalación Python.mpkg para iniciar el instalador de Python para Mac.

Figura 11: Bienvenida a la instalación

La primera página (figura 11) que muestra el programa de instalación describe de forma concisa qué es Python, y remite al fichero ReadMe.txt (que seguramente no te leíste ¿verdad?) por si deseas conocer más detalles.

Pulsa el botón Continue para avanzar en la instalación.

La siguiente pantalla (figura 12) muestra información importante: Python necesita que tengas instalado Mac OS X 10.3 o superior. Si estás ejecutando una versión de Mac OS X 10.2 o anterior, deberías actualizar tu ordenador a última versión. Una de las razones más convincentes, es que Apple ya no proporciona actualizaciones de seguridad para tu versión del sistema operativo, por lo que tu ordenadores está en riesgo cada vez que está conectado a Internet. Otra razón, no menos convincente, es que no puedes ejecutar Python 3.

Figura 12: Información importante

Pulsa el botón Continue para avanzar en la instalación.

Figura 13: Licencia

Como todos los buenos instaladores, lo siguiente que el instalador de Python muestra es la pantalla de aceptación de la licencia (figura 13). Python es Open Source (software de fuentes abiertas) cuya licencia cuenta con la aprobación de la iniciativa de Código Abierto. Python cuenta con un cierto número de propietarios y patrocinadores a lo largo de su historia, cada uno de los cuales ha dejado su marca en la licencia. Pero el resultado final es este: Python es Código Abierto, y puedes usarlo en cualquier plataforma, para lo que desees, sin necesidad de pagar ningún canon, ni obligación, ni nada a cambio.

Pulsa el botón Continue de nuevo para avanzar en la instalación.

Debido a las peculiaridades del proceso de instalación estándar de Apple, es

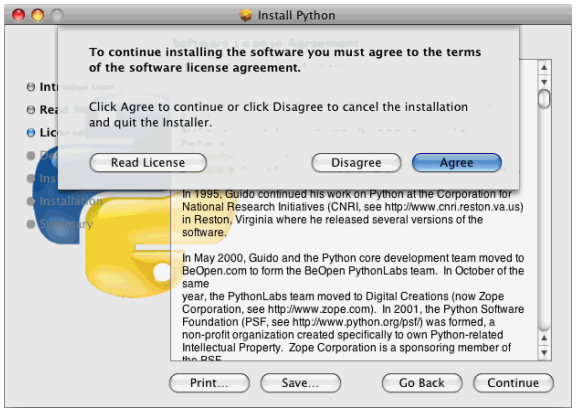


Figura 14: Aceptación de la Licencia

necesario que aceptes la licencia (figura 14) para que el instalador te permita continuar. Puesto que Python es Código Abierto, en realidad estás aceptando una licencia que te garantiza derechos adicionales, en lugar de quitártelos.

Pulsa el botón Agree para continuar.

La siguiente pantalla (figura 15) te permite modificar la ubicación en la que se efectuará la instalación. **Debes** instalar Python en el disco de arranque, pero debido a ciertas limitaciones en el instalador, éste no te obliga a ello, por lo que ¡ten cuidado!. En realidad, yo nunca he tenido la necesidad de cambiar la ubicación de instalación, por ello, salvo causa justificada, acepta la ubicación sugerida por el instalador.

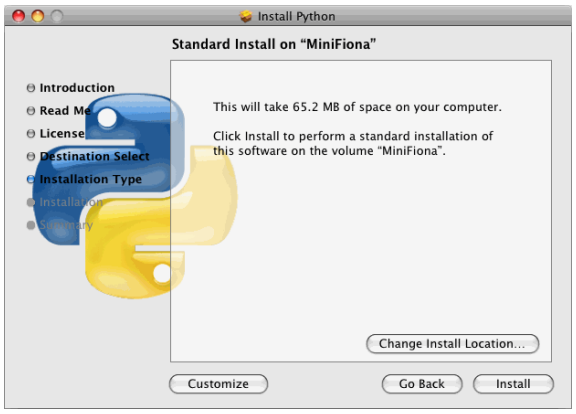


Figura 15: Selección de la ubicación

Desde esta pantalla también puedes modificar la instalación con el fin de que no

se instalen algunas funcionalidades. Si quieres hacer esto pulsa el botón Customize, en caso contrario pulsa el botón Instalar.

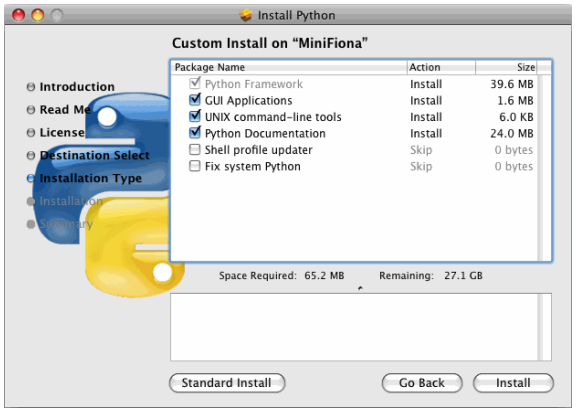


Figura 16: Personalización de la instalación

Si eliges una instalación personalizada (has pulsado el botón Customize), el instalador te muestra (figura 16) una pantalla con una lista de características:

- **Python Framework**. Es el núcleo de Python, por lo que está seleccionado y deshabilitado con el fin de que no puedas cambiarlo.
- **Aplicaciones GUI** incluye IDLE, la consola interactiva gráfica de Python que usaremos a lo largo de todo el libro. Te recomiendo encarecidamente que mantengas esta opción seleccionada.
- **Herramientas de línea de comandos**, que incluyen la aplicación python3. También te recomiendo que mantegas esta opción seleccionada.
- **Documentación de Python**, que contiene mucha de la información disponible en docs.python.org. Muy recomendables si tienes previsto estar desconectado de Internet.
- **Actualizador del perfil de la consola**, que controla si actualizas tu perfil de consola (utilizado por la aplicación Terminal.app) con el fin de que la versión de Python que estás instalando se encuentre en el camino de búsqueda de la consola. Para los propósitos de este libro, esta opción no es necesario que la instales.
- **Actualizar la versión de Python del sistema**. Esta opción no debería modificarse. Le dice a tu ordenador Mac que utilice Python 3 como versión

por defecto para todos los scripts, incluido aquellos que vienen con el sistema operativo. Seleccionar esta opción podría producir efectos muy negativos en tu sistema, puesto que la mayor parte de los scripts del sistema operativo están escritos para Python 2, y pueden fallar en Python 3.

Pulsa el botón Install para continuar.

Figura 17: Solicitando derechos administrativos

Debido a que el instalador copia archivos binarios en /usr/local/bin/, antes de iniciar dicha copia se solicitan permisos de administrador mediante una pantalla (figura 17) en la que hay que teclear la clave del administrador del sistema. No es posible instalar Python en Mac sin disponer de las credenciales de administrador.

Pulsa el botón OK para comenzar la instalación.

El instalador mostrará una barra de progreso (figura 18) mientras se instalan las funcionalidades que hayas seleccionado.

Si todo va bien, el instalador mostrará en pantalla (figura 19) una marca verde para indicar que la instalación de ha completado satisfactoriamente.

Pulsa el botón Close para salir del instalador.

Si no has cambiado la ubicación de la instalación, Python 3.1.* se habrá instalado en una carpeta denominada Python 3.1 (figura 20) dentro de la carpeta /Aplications. El elemento más importante en ella es IDLE, que es la consola gráfica interactiva de Python.

Haz doble click con el cursor sobre IDLE para ejecutar la consola de Python.

La mayor parte del tiempo la pasarás explorando Python mediante el uso de

Figura 18: Instalación

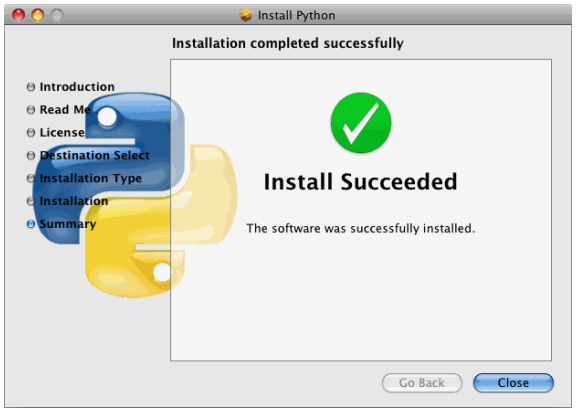


Figura 19: Fin de la instalación

esta consola (figura 21). Los ejemplos de este libro asumen que eres capaz de ejecutar esta consola en todo momento.

Continúa en el apartado 0.7

0.5. Instalación en Ubuntu Linux

Las diferentes distribuciones existentes hoy día de Linux suelen disponer de vastos repositorios de aplicaciones listas para instalar de forma sencilla. Los detalles exactos varían en función de la distribución de Linux. En Ubuntu Linux, la forma más sencilla de instalar Python 3 consiste en usar la opción Añadir y quitar... del menú de Aplicaciones (figura 22).

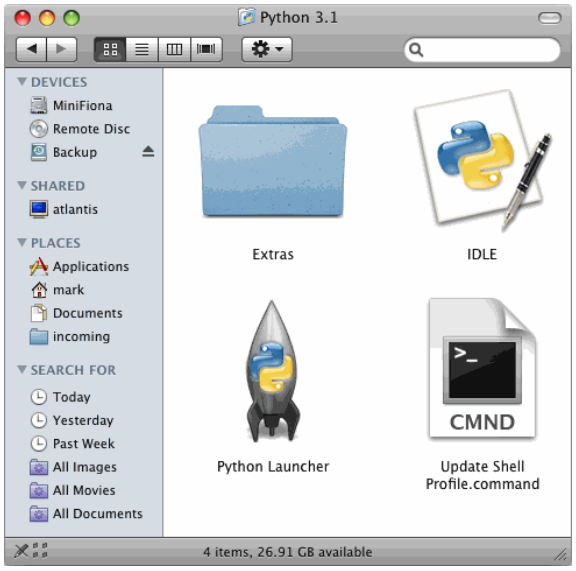


Figura 20: Carpeta Python

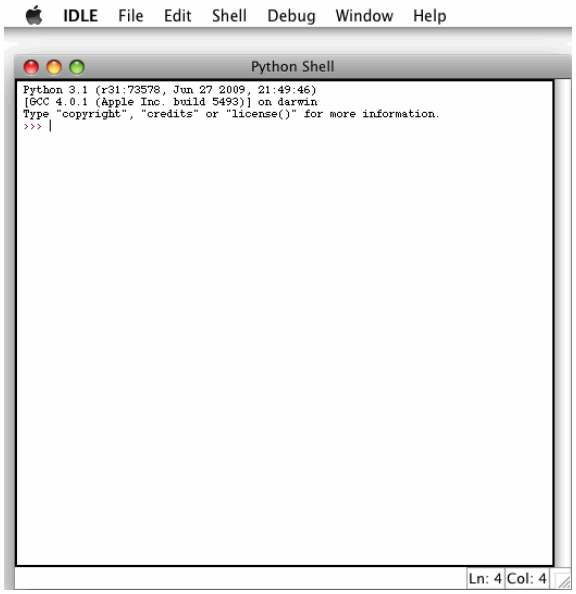


Figura 21: Consola gráfica

Cuando ejecutas por primera vez el programa para Añadir/Quitar aplicaciones, se muestra una lista de aplicaciones preseleccionadas en diferentes categorías. Algunas ya se encuentran instaladas en tu ordenador, pero la mayoría no. Puesto

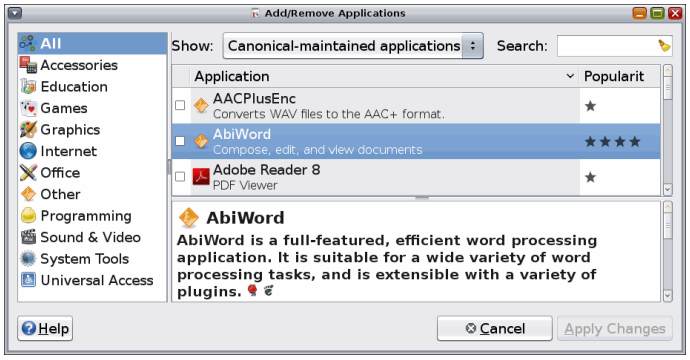


Figura 22: Añadir/Quitar aplicaciones

que este repositorio consta de más de 10.000 aplicaciones, encontrar la que se desea puede ser difícil, para facilitar la labor es posible aplicar diferentes filtros que limitan las aplicaciones que se muestran en la lista de pantalla. El filtro por defecto es "aplicaciones mantenidas por Canonical" que es el pequeño subconjunto formado por aquellas apliicaciones que se mantienen oficialmente por parte de Canonical, la compañía que distribuye y mantiene Ubuntu Linux.

Como Python 3 no está en este subconjunto de aplicaciones, el primer paso es desplegar los filtros (Mostrar:) y seleccionar Todas las aplicaciones libres (figura 23).

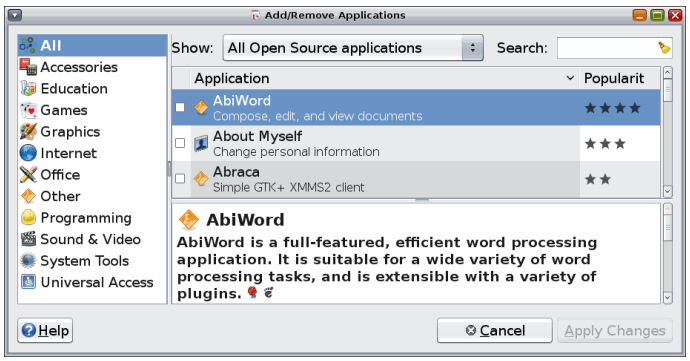


Figura 23: Todas las aplicaciones libres

Después puedes filtrar aún más utilizando la caja de texto de búsqueda con el fin de buscar el texto Python 3 (figura 24).

Ahora la lista de aplicaciones que se muestran se limita a aquellas que, de algún modo, incluyen la cadena Python 3. Ahora debes marcar dos paquetes. El primero es Python (v3.0). Que contiene el intérprete de Python 3.

Figura 24: Búsqueda de aplicaciones relacionadas con Python 3

Figura 25: Selección del paquete Python 3

El segundo paquete que hay que marcar se encuentra inmediatamente delante, IDLE (usando Python 3.0), que es la consola gráfica que usaremos a lo largo de todo el libro (figura 26).

Una vez hayas seleccionado los dos paquetes, pulsa el botón Aplicar cambios para continuar.

El gestor de paquetes solicitará que confirmes que quieres instalar tanto IDLE (usando Python 3.0) como Python (3.0) (figura 27).

Pulsa el botón Aplicar para continuar.

El gestor de paquetes te pedirá que te identifiques con la clave de usuario para acceder a los privilegios administrativos que permiten instalar aplicaciones. Una vez hecho esto, el gestor de paquetes mostrará una pantalla (figura 28) con el grado de avance de la instalación mientras se descargan los paquetes seleccionados del repositorio de Internet de Ubuntu Linux.

Figura 26: Selección del paquete IDLE

Figura 27: Confirmación

Cuando los paquetes se hayan descargado, el instalador iniciará automáticamente el proceso de instalación en tu ordenador (figura 29).

Si todo va bien, el gestor de paquetes confirmará que ambos paquetes se instalaron satisfactoriamente (figura 30). Desde esta pantalla puedes ejecutar directamente IDLE haciendo doble click sobre él. O puedes pulsar el botón Cerrar para finalizar el gestor de paquetes.

En cualquier caso, puedes lanzar la consola gráfica de Python siempre que quieras seleccionando IDLE en el submenú Programación del menú de Aplicaciones.

Es en la consola de Python (figura 31) donde pasarás la mayor parte del tiempo explorando Python. Los ejemplos de este libro asumen que eres capaz de ejecutar la consola de Python siempre que sea necesario.

Continúa en el apartado 0.7

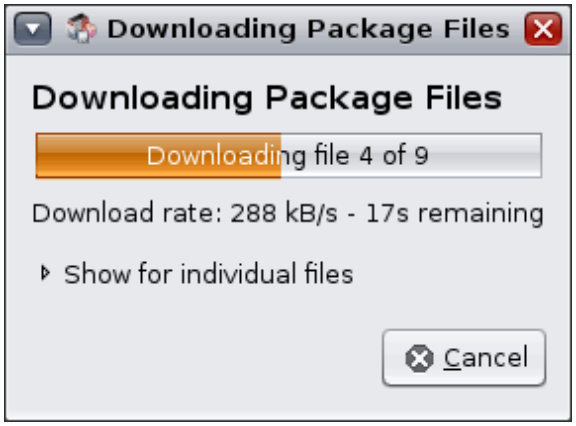


Figura 28: Descarga de paquetes

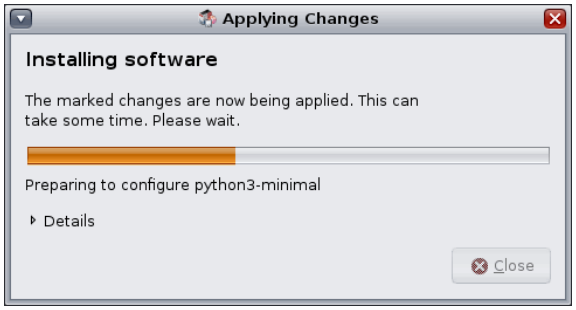


Figura 29: Descarga de paquetes

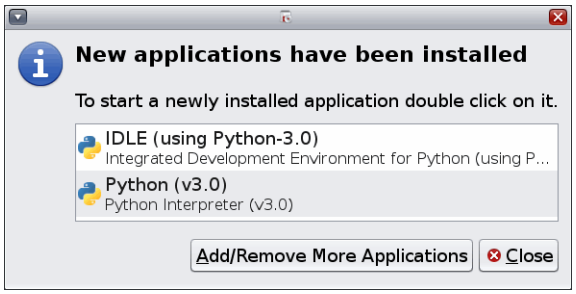


Figura 30: Instalación finalizada

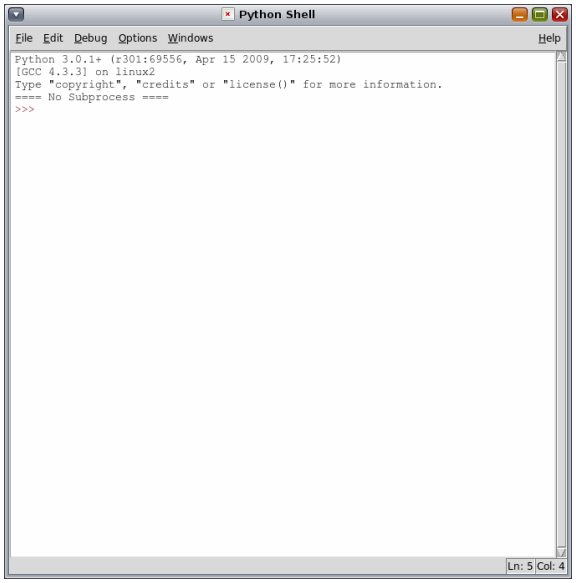


Figura 31: Consola de Python en Ubuntu Linux

0.6. Instalación en otras plataformas

Python 3 está disponible en otras muchas plataformas. En particular, está disponible prácticamente en todas las distribuciones Linux, BSD y Sun Solaris. Por ejemplo, RedHat Linux utiliza el gestor de paquetes yum; FreeBSD tiene su propia colección de paquetes, y Solaris tiene el gestor de paquetes pkgadd y otros. Una rápida búsqueda en Internet de los términos Python 3 + emphtu sistema operativo te mostrará si existe un paquete de Python 3 disponible para tu sistema, y cómo instalarlo.

0.7. Uso de la consola interactiva de Python

En la consola interactiva de Python puedes explorar la sintaxis del lenguaje, solicitar ayuda interactiva sobre las sentencias del lenguaje, y depurar programas cortos.

La consola gráfica (denominada IDLE) también proporciona un editor de textos bastante decente que resalta mediante colores la sintaxis del lenguaje Python. Si no tienes aún un editor de textos de tu elección, puedes darle una oportunidad a IDLE.

¡Vamos a comenzar! La shell de Python es un estupendo lugar para comenzar

a *jugar* con el lenguaje Python de forma interactiva. A lo largo de este libro verás un montón de ejemplos como este:

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

Los tres símbolos de *mayor que*, >>>, representan el *prompt*[9] de Python. No teclees nunca estos tres caracteres. Se muestran para que sepas que este ejemplo se debe teclear en la consola de Python.

Lo que tienes que teclear es 1 + 1. En la consola puedes teclear cualquier expresión o sentencia válida del lenguaje. ¡No seas tímido, no muerde! Lo peor que puede pasarte es que Python muestre un mensaje de error, si tecleas algo que no entiende. Las sentencias se ejecutan inmediatamente (después de que pulses la tecla INTRO); las expresiones se calculan en el momento, y la consola imprime en pantalla el resultado.

2 es el resultado de la expresión. Como 1 + 1 es una expresión válida en el lenguaje Python, al pulsar la tecla INTRO Python evalúa la expresióne imprime el resultado, que en este caso es 2.

Vamos a probar otro ejemplo.

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

Muy sencillo, ¿no?

Pero hay muchas otras cosas que puedes hacer en la consola de Python. Si en algún momento te bloqueas —no recuerdas una sentencia, o no recuerdas los argumentos que debes pasar a una función determinada— puedes obtener ayuda en la propia consola. Simplemente teclea help y pulsa ENTER.

```
>>> help
Type help() for interactive help, or help(object) for help about object.
```

Exiten dos modos de ayuda:

- Puedes solicitar ayuda de un objeto concreto, lo que muestra la documentación del mismo y vuelve al prompt de la consola de Python.

[9]Nota del Traductor: El prompt es el indicador que usa una consola, en este caso la consola de Python, para que el usuario sepa que puede teclear alguna sentencia. Como el uso de la palabra *prompt* está tan extendido para este concepto, y no existe uno en español de amplio uso, en este libro se utilizará sin traducir.

- También puedes entrar en el *modo ayuda*, en el que en lugar de evaluar expresiones de Python, puedes teclear palabras reservadas del lenguaje o nombres de sentencias y la consola imprime lo que sepa sobre ellas.

  Para entrar en el modo interactivo de ayuda teclea help() y pulsa INTRO.

```
>>>help()

Welcome to Python 3.0!  This is the online help utility.

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules",
"keywords", or "topics".  Each module also comes with a one-line summary
of what it does; to list the modules whose summaries contain a given word
such as "spam", type "modules spam".

help>
```

Observa que ahora el prompt cambia de >>> a help>. Este cambio sirve para recordarte que te encuentras en el modo de ayuda interactiva. Ahora puedes teclear cualquier palabra reservada, sentencia, nombre de módulo, nombre de función —casi cualquier cosa que Python entienda— y leer la documentación que haya disponible sobre el tema tecleado.

```
help> print

Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file: a file-like object (stream); defaults to the current sys.stdout.
    sep:  string inserted between values, default a space.
    end:  string appended after the last value, default a newline.
help> Papaya
no Python documentation found for 'Papaya'
```

```
help> quit

You are now leaving help and returning to the Python interpreter.
If you want to ask for help on a particular object directly from the
interpreter, you can type "help(object)".  Executing "help('string')"
has the same effect as typing a particular string at the help> prompt.

>>>
```

En el ejemplo anterior se obtiene en primer lugar la documentación sobre la función print. Para ello se ha tecleado en el modo ayuda la palabra print y luego se ha pulsado INTRO. Como resultado se obtiene un texto en el que se muestra el nombre de la función, un breve resumen de la misma, los argumentos de la función y sus valores por defecto. Si la documentación te parece demasiado opaca, no te asustes. Aprenderás lo necesario sobre todos estos conceptos en los próximos capítulos de este libro.

Evidentemente el modo de ayuda no lo sabe todo. Si tecleas algo que no sea una sentencia, módulo, función u otra palabra reservada de Python,el modo de ayuda interactiva mostrará un mensaje indicando que no encuentra documentación alguna para el concepto que hayas tecleado.

Por último, para salir del modo de ayuda únicamente tienes que teclear quit y pulsar INTRO.

El prompt vuelve de nuevo a >>> para indicar que has abandonado el modo de ayuda interactiva y que de nuevo te encuentras en la consola de Python.

IDLE, además de servir como consola gráfica de Python, incluye también un editor de textos que conoce el lenguaje Python. Verás cómo usarlo en la sección siguiente.

0.8. Editores de texto e IDEs para Python

IDLE no es el único entorno existente para escribir programas en Python. Aunque es muy útil para comenzar a aprender el lenguaje, muchos desarrolladores prefieren utilizar otros editores de texto o *Entornos Integrados de Desarrollo*[10]. No los voy a abarcar aquí, únicamente comentaré que la comunidad de Python mantiene una lista de editores para el lenguaje Python sobre diversas plataformas y licencias de software.

[10]En inglés se suele hablar de IDE, para referirse a los *Integrated Development Environment*, que son aplicaciones que permiten desarrollar de forma rápida al incluir un editor de textos, compilador, depurador e incluso herramientas de diseño de aplicaciones avanzadas.

También puede ser de interés para ti la lista de Entornos Integrados de Desarrollo para Python, aunque aún son pocos los que sirven para Python 3. Uno de ellos es PyDev, un plugin para Eclipse que convierte a Eclipse en un completo Entorno Integrado de Desarrollo para Python. Ambos, Eclipse y PyDev, son multiplataforma y de código abierto.

Por la parte comercial, existe un entorno de desarrollo denominado Komodo IDE. Tiene una licencia que se paga por cada usuario, pero también ofrece descuento para estudiantes, y una versión con licencia de prueba limitada.

Llevo programando en Python nueve años, yo, para editar los programas, utilizo GNU Emacs y los depuro en la shell de línea de comando[11]. No existe un modo correcto de desarrollar en Python. ¡Encuentra lo que mejor se adapte a ti!

[11] Nota del Traductor:En mi caso uso GVim y el depurador de consola pudb

Capítulo 1

Tu primer programa en Python

Nivel de dificultad:◆ ◇ ◇ ◇ ◇

"No entierres tu carga en un santo silencio.
¿Tienes un problema? Estupendo. Alégrate,
sumérgete en él e investiga."
—Ven. Henepola Gunarata

1.1. Inmersión

Los libros sobre programación suelen comenzar con varios capítulos sobre los fundamentos y van, poco a poco, avanzando hasta llegar a hacer programas útiles. Vamos a saltarnos todo eso. Lo primero que vamos a ver es un programa Python completo. Probablemente no tenga ningún sentido para ti. No te preocupes por eso, vamos a diseccionarlo línea por línea. Primero léelo y trata de interpretarlo.

```
1  # parahumanos.py
2
3  SUFIJOS = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],
4             1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB',
5                    'YiB']}
6
7  def tamanyo_aproximado(tamanyo, un_kilobyte_es_1024_bytes=True):
8      '''Convierte un tamaño de fichero en formato legible por personas
9
10     Argumentos/parámetros:
11     tamanyo -- tamaño de fichero en bytes
12     un_kilobyte_es_1024_bytes -- si True (por defecto),
13                                  usa múltiplos de 1024
14                                  si False, usa múltiplos de 1000
```

```
15
16      retorna: string
17
18      '''
19      if tamanyo < 0:
20          raise ValueError('el número debe ser no negativo')
21
22      multiplo = 1024 if un_kilobyte_es_1024_bytes else 1000
23      for sufijo in SUFIJOS[multiplo]:
24          tamanyo /= multiplo
25          if tamanyo < multiplo:
26              return '{0:.1f} {1}'.format(tamanyo, sufijo)
27
28      raise ValueError('número demasiado grande')
29
30  if __name__ == '__main__':
31      print(tamanyo_aproximado(1000000000000, False))
32      print(tamanyo_aproximado(1000000000000))
```

Antes de analizarlo paso a paso vamos a ejecutar el programa en la línea de comandos. En Linux o en Mac debes teclear: python3 parahumanos.py[1]. El resultado será parecido a lo siguiente:

```
1  tu_usuario@tu_ordenador:~/inmersionEnPython3$ python3 parahumanos.py
2  1.0 TB
3  931.3 GiB
```

En Windows debes teclear lo mismo: python3 parahumanos.py, únicamente varía la forma del prompt de la consola. El resultado será parecido a:

```
1  C:\\inmersionenpython3:> python3 parahumanos.py
2  1.0 TB
3  931.3 GiB
```

¿Qué ha pasado? Acabas de ejecutar tu primer programa Python. Has ejecutado el intérprete de Python en la línea de comandos (python3), y le has pasado como parámetro el nombre del fichero de script (parahumanos.py) que querías ejecutar.

El fichero de script, a su vez, define una única función de python, la función tamnyo_aproximado, que toma como parámetros un tamaño de fichero con una precisión de bytes y calcula el tamaño en una unidad mayor en la que el valor quede más *bonito*, a cambio, el resultado es aproximado. (El funcionamiento del Explorador de Windows; del Finder de Mac OS X, o de Nautilus, Dolphin o Thunar de Linux es muy parecido. Si muestras en cualquiera de ellos una carpeta de documentos en modo detalle, de forma que se vean en diferentes columnas, el icono del documento,

[1] Para que funcione correctamente debes moverte al directorio en el que esté grabado el fichero parahumanos.py.

nombre, tamaño, tipo, fecha de última modificación, etc. Observarás que si un documento determinado ocupa 1093 bytes, en la columna de tamaño no dirá eso, sino que dirá algo así como 1 KB. Esto es lo que hace la función tamanyo_aproximado)

Las líneas de código print(tamanyo_aproximado(*argumentos*)) del final del script, líneas 31 y 32, son dos llamadas a funciones —primero se llama a la función tamanyo_aproximado() pasándole unos parámetros (también llamados argumentos), esta función se ejecuta y devuelve un resultado que, posteriormente, se pasa como parámetro a la función print(). Todo ello en la misma línea.

La función print() es interna del lenguaje Python[2]; nunca verás una declaración explícita de ella. La puedes usar cuando quieras, en cualquier parte de un programa Python[3].

¿Porqué la ejecución del script en la línea de comandos retorna siempre la misma respuesta? Lo veremos más adelante. Primero vamos a ver el funcionamiento de la función tamanyo_aproximado().

1.2. Declaración de funciones

Python dispone de funciones como la mayoría de los lenguajes, pero no tiene ficheros de cabecera como c++ o secciones de interface/implementation como en Pascal. En Python únicamente hay que declarar la función, como en el siguiente ejemplo:

```
1 def tamanyo_aproximado(tamanyo, un_kilobyte_es_1024_bytes=True):
```

La palabra reservada def inicia la declaración de la función, seguida del nombre que le quieres dar a la misma, seguida de los parámetros de la función entre paréntesis. Separándolos por comas en caso de que sean varios parámetros.

Observa también que, en Python, las funciones no definen un tipo de datos de retorno. No se especifica el tipo de datos del valor que retornan las funciones. Es más, ni siquiera se especifica si se retorna o no un valor.

> En Python cuando necesitas una función, solamente tienes que declararla.

En realidad, todas las funciones de Python tienen un valor de retorno; si dentro del código de la función se ejecuta una sentencia return, el valor que acompaña a la sentencia será el valor de retorno, en caso contrario se retorna el valor None, que es la forma de expresar el vacío (null) en Python.

[2] En inglés built-in.

[3] Existen montones de funciones internas del lenguaje, y muchas más que están separadas en *módulos*. Lo veremos poco a poco, ten paciencia, pequeño saltamontes.

> En algunos lenguajes, las funciones que retornan un valor se declaran con la palabra function, y las subrutinas que no retornan un valor con la palabra sub. En Python no existen las subrutinas. Todas son funciones, todas las funciones devuelven un valor (None si tú no devuelves algo expresamente con la palabra reservada return) y todas las funciones comienzan con la palabra def.

La función tamanyo_aproximado() recibe dos parámetros o argumentos, —tamanyo y un_kilobyte_es_1024_bytes— pero ninguno de ellos especifica un tipo de datos. En Python, las variables nunca se tipifican explícitamente, Python deduce y mantiene el tipo de datos de la variable de forma interna según el valor que tenga asignado la misma.

> En Java y otros lenguajes con tipificación estática, debes especificar el tipo de datos de los parámetros y valor de retorno de cada función. En Python nunca especificas el tipo de datos de nada de forma explícita. Python mantiene el rastro de los tipos de datos de forma interna basándose en los valores que asignes a las variables.

1.2.1. Parámetros opcionales y con nombre

Python permite que los parámetros de una función tengan valores por defecto; si la función se llama (para ejecutarla) si indicar el parámetro Python usará el valor por defecto para asignarlo al parámetro que no se ha especificado en la llamada a la función. Asimismo, los parámetros se pueden pasar en la llamada en cualquier orden si se utilizan parámetros con nombre.

Veamos de nuevo la declaración de la función tamanyo_aproximado().

```
1 def tamanyo_aproximado(tamanyo, un_kilobyte_es_1024_bytes=True):
```

El segundo parámetro un_kilobyte_es_1024_bytes, especifica un valor por defecto igual a True. Como consecuencia, este parámetro pasa a ser *opcional*; puedes llamar a la función sin pasarlo en los paréntesis. Python se comportará como si lo hubieras llamado con el valor True como segundo parámetro.

Veamos el final del script[4]:

```
1 if __name__ == '__main__':
2     print(tamanyo_aproximado(1000000000000, False))
3     print(tamanyo_aproximado(1000000000000))
```

[4]En Python se les suele llamar también *script* a los ficheros con el código fuente de los programas.

1. La primera llamada a la función (línea 2) utiliza dos parámetros. Durante la ejecución de la función tamanyo_aproximado un_kilobyte_es_1024_bytes tendrá el valor False, que es lo que se pasa como segundo parámetro en la llamada a la función.

2. La segunda llamada a la función (línea 3) utiliza un único parámetro. Pero Python no se queja ya que el segundo es opcional. Como no se especifica, el segundo parámetro utiliza su valor por defecto True, de acuerdo a lo que se definió en la declaración de la función.

También puedes pasar los valores a una función utilizando nombres. Prueba lo siguiente en la consola:

```
1  >>> from parahumanos import tamanyo_aproximado
2  >>> tamanyo_aproximado(4000, un_kilobyte_es_1024_bytes=False)
3  '4.0 KB'
4  >>> tamanyo_aproximado(tamanyo=4000, un_kilobyte_es_1024_bytes=False)
5  '4.0 KB'
6  >>> tamanyo_aproximado(un_kilobyte_es_1024_bytes=False, tamanyo=4000)
7  '4.0 KB'
8  >>> tamanyo_aproximado(un_kilobyte_es_1024_bytes=False, 4000)
9  SyntaxError: non-keyword arg after keyword arg (<pyshell#4>, line 1)
10 >>> tamanyo_aproximado(tamanyo=4000, False)
11 SyntaxError: non-keyword arg after keyword arg (<pyshell#5>, line 1)
12 >>>
```

1. *Línea 2:* Llama a la función tamnyo_aproximado() pasándole 4000 al primer parámetro (tamanyo) y el valor False en el denominado un_kilobyte_es_1204_bytes (En este caso coincide que el parámetro con nombre se está pasando en la segunda posición y también está declarado en la función como segundo parámetro, pero esto es simplemente una coincidencia).

2. *Línea 4:* Llama a la función tamanyo_aproximado() pasándole 4000 al parámetro denominado tamanyo y False al parámetro denominado un_kilobyte_es_1024_bytes (Estos parámetros coinciden en orden con los de la declaración de la función, pero vuelve a ser una simple coincidencia).

3. *Línea 6:* Llama a a la función tamanyo_aproximado() paándole False al parámetro denominado un_kilobyte_es_1024_bytes y 4000 al parámetro denominado tamanyo (Esta es la utilidad de usar nombres en las llamadas a una función, poder pasarlos en cualquier orden, e incluso no pasar alguno de los existentes para que tomen valores por defecto mientras sí que pasas uno de los últimos parámetros de la función).

4. *Línea 8:* Esta llamada a la función falla porque se usa un parámetro con nombre seguido de uno sin nombre (por posición). Esta forma de llamar a la función siempre falla. Python lee la lista de parámetros de izquierda a derecha, en cuanto aparece un parámetro con nombre, el resto de parámetros debe también proporcionarse por nombre. Los primeros pueden ser por posición.

5. *Línea 10:* Esta llamada también falla por la misma razón que la anterior. ¿Te sorprende? Después de todo, el primer parámetro se ha denominado tamanyo y recibe el valor 4000, es *obvio* que el valor False debería asignarse al parámetro un_kilobyte_es_1024_bytes. Pero Python no funciona de esa forma. Tan pronto como lee un parámetro con nombre, todos los parámetros siguientes (a la derecha) tienen que llevar el nombre del parámetro.

1.3. Cómo escribir código legible

No te voy a aburrir con una larga charla sobre la importancia de documentar el código. Solamente decir que el código se escribe una vez pero se lee muchas veces, y que quien más lo va a leer eres tú, seis meses después de haberlo escrito (por ejemplo: cuando ya no te acuerdes de nada pero necesites corregir o añadir algo). Python hace fácil escribir código legible, aprovéchate de ello. Me lo agradecerás dentro de seis meses.

1.3.1. Cadenas de texto de documentación

Puedes documentar una función proporcionándole una cadena de documentación (abreviando se suele hablar de docstring). En este programa, la función tamanyo_aproximado() tiene una cadena de documentación (docstring):

```
1  def tamanyo_aproximado(tamanyo, un_kilobyte_es_1024_bytes=True):
2      '''Convierte un tamaño de fichero en formato legible por personas
3
4      Argumentos/parámetros:
5      tamanyo -- tamaño de fichero en bytes
6      un_kilobyte_es_1024_bytes -- si True (por defecto),
7                                   usa múltiplos de 1024
8                                   si False, usa múltiplos de 1000
9
10     retorna: string
11
12     '''
```

La comillas triples sirven para escribir cadenas de texto que ocupen más de

una línea. Todo lo que se escribe entre las comillas triples forma parte de una única cadena de texto, incluídos los espacios en blanco, retornos de carro, saltos de línea y otras comillas *sueltas*. Este tipo de cadenas de texto lo puedes utilizar donde quieras dentro del código Python, pero normalmente se utilizan para definir docstring (cadenas de texto de documentación).

> Las comillas triples son la manera más simple de escribir cadenas de texto que incluyan, a su vez, comillas simples y/o dobles, como cuando en Perl 5 se utiliza q/.../

> Todas las funciones se merecen un *docstring* que las explique

En este ejemplo, todo lo que se encuentra entre las comillas triples es el docstring de la función, que sirve para documentar lo que hace la función. Un docstring, si existe, debe ser lo primero que aparece definido en una función (es decir, se debe encontrar en la primera línea que aparece después de la declaración de la función). Técnicamente no necesitas escribir un docstring para cada función, pero deberías. Sé que lo has escuchado en las clases que programación a las que hayas asistido, pero Python te da un incentivo mayor para que lo hagas: los docstring están disponibles en tiempo de ejecución como un atributo de la función.

> Muchos entornos integrados de programación (IDEs) utilizan los docstring para proporcionar ayuda y documentación sensible al contexto, de forma que cuando teclees el nombre de una función, aparece su docstring como pista sobre el significado de la función y de sus parámetros. Esto puede ser muy útil, tan útil como explicativos sean los docstring que escribas.

1.4. El camino de búsqueda para import

Antes de continuar, quiero mencionar brevemente el camino[5] de búsqueda de las librerías. Cuando importas un módulo, Python busca en varios lugares hasta encontrarlo. En concreto, busca en todos los directorios que se encuentren definidos en la variable sys.path. Como se trata de una lista, puedes verla fácilmente o modificarla con los métodos estándares de manipulación de listas. (Aprenderás a trabajar con listas en el capítulo **??** sobre Tipos de Dato Nativos).

[5] En español se usa también ruta de búsqueda. En inglés se usa la palabra path para referirse a este concepto

```
1  >>> import sys
2  >>> sys.path
3  ['',
4   '/usr/lib/python3.0',
5   '/usr/lib/python3.0/plat-linux2',
6   '/usr/lib/python3.0/lib-dynload',
7   '/usr/lib/python3.0/dist-packages',
8   '/usr/local/lib/python3.0/dist-packages']
9  >>> sys
10 <module 'sys' (built-in)>
11 >>> sys.path.insert(0, '/home/jmgaguilera/inmersionenpython3/ejemplos')
12 >>> sys.path
13 ['/home/jmgaguilera/inmersionenpython3/ejemplos',
14  '',
15  '/usr/lib/python3.0',
16  '/usr/lib/python3.0/plat-linux2',
17  '/usr/lib/python3.0/lib-dynload',
18  '/usr/lib/python3.0/dist-packages',
19  '/usr/local/lib/python3.0/dist-packages']
20 >>>
```

1. *Línea 1:* Al importar el paquete sys de esta forma, todas sus funciones y atributos quedan a disposición del programador para su uso.

2. *Líneas 2-8:* sys.path es una variable (path) del paquete sys que contiene una lista de los directorios que constituyen el camino de búsqueda (El tuyo será diferente, ya que depende del sistema operativo, de la versión de Python que tengas instalada, y del lugar en el que está instalada). Siempre que se haga un import en el código, Python buscará en estos directorios (por orden), hasta encontrar un fichero cuyo nombre coincida con el valor que se usa en la sentencia import cmás la extensión .py.

3. *Líneas 9-10:* En realidad te he mentido un poco, la realidad es un poco más compleja, no todos los módulos se almacenan en ficheros con extensión .py. Algunos de ellos, como el módulo sys son módulos internos (built-in); no existen en ficheros, están construidos internamente en el propio lenguaje. En la práctica funcionan exactamente igual que los módulos que están en ficheros, la única diferencia es que no existe el código fuente, ¡Porque no están escritos en Python! (El módulo sys está escrito en lenguaje c).

4. *Línea 11:* En *tiempo de ejecución* puedes añadir un nuevo directorio al camino de búsqueda de Python añadiendo un directorio a la variable sys.path, así Python también buscará en él cada vez que intentes importar un módulo. El efecto de este cambio dura mientras se mantenga en ejecución Python. Al

finalizar, y volver a entrar en Python, el camino (la variable sys.path) volverá a tener los valores iniciales.

5. *Líneas 12-19*: Al ejecutar sys.path.insert(0, path) se nsertó un nuevo directorio en la primera posición (en Python la primera posición se numera con el cero) de la lista de sys.path. Casi siempre, será esto lo que quieras hacer. En casos en los que exista algún conflicto de nombres (por ejemplo, si Python tiene su propia versión de una librería y es de la versión 2, pero quieres utilizar otra que sea de la versión 3), así te aseguras que tus módulos se encuentran antes y ejecutan en lugar de los originales.

1.5. En Python todo es un Objeto

En caso de te lo hayas perdido, acabo de decir que las funciones de Python tienen atributos, y que esos atributos están disponibles en tiempo de ejecución. Una función, como todo lo demás en Python, es un objeto.

Abre la consola interactiva de Python y ejecuta lo siguiente:

```
1  >>> import parahumanos
2  >>> print(parahumanos.tamanyo_aproximado(4096, True))
3  4.0 KiB
4  >>> print(parahumanos.tamanyo_aproximado.__doc__)
5  Convierte un tamaño de fichero en un formato legible por personas
6
7      Argumentos/parámetros:
8      tamanyo -- tamaño de fichero en bytes
9      un_kilobyte_es_1024_bytes -- si True (por defecto),
10                                  usa múltiplos de 1024
11                                  si False, usa múltiplos de 1000
12
13     retorna: string
14
15
16 >>>
```

1. *Línea 1:* Importa (carga en memoria) el programa parahumanos como un módulo —un trozo de código que puedes utilizar de forma interactiva o desde un programa Python mayor. Una vez se ha importado el módulo, puedes utilizar (referenciar) cualquiera de sus funciones públicas, clases o atributos. Si desde un módulo se desea utilizar la funcionalidad de otro, basta con hacer exactamente lo mismo que en esta línea de la consola interactiva.

2. *Línea 2:* Cuando quieres utilizar las funciones que estén definidas en los módulos importados, tienes que añadir el nombre del módulo. No es posible utilizar simplemente tamanyo_aproximado, debes utilizar parahumanos.tamanyo_aproximado. Si has utilizado Java, esta forma de utilizar una función debería sonarte.

3. *Línea 4:* En este caso, en lugar de llamar a la función como podrías esperar, se consulta uno de los atributos de la función, __doc__.

> En Python import es equivalente al require de Perl. Cuando importas (import) un módulo de Python puedes acceder a todas sus funciones con la sintaxis módulo.función. En Perl, cuando se requiere (require) un módulo puedes acceder a todas sus funciones con la sintaxis módulo::función

1.5.1. ¿Qué es un objeto?

En Python todo es un objeto, y todos los objetos pueden tener atributos y métodos. Todas las funciones son objetos, y tienen el atributo __doc__, que retorna el docstring que se haya definido en el código fuente. El módullo sys es también un objeto que tiene (entre otras cosas) un atributo denominado path. Como se ha dicho: todo lo que se defina en Python es un objeto y puede tener atributos y métodos.

Sin embargo, no hemos contestado aún a la pregunta fundamental: ¿Qué es un objeto? Los diferentes lenguajes de programación definen *objeto* de diferente forma. En algunos, significa que *todos* los objetos *deben* tener atributos y métodos; en otros, significa que todos los objetos pueden tener subclases. En Python la definición es más *relajada*. Algunos objetos no tienen ni atributos ni métodos, *pero podrían*. No todos los objetos pueden tener subclases. Pero todo es un objeto en el sentido de que pueden asignarse a variables y pasarse como parámetro de una función.

Puede que hayas oído en otro contexto de programación el término *objeto de primera clase*. En Python, las funciones son *objetos de primera clase*. Puedes pasar una función como parámetro de otra función. Los módulos también son *objetos de primera clase*. Puedes pasar un módulo completo como parámetro de una función. Las clases son *objetos de primera clase*, y las instancias de las clases también lo son.

Esto es importante, por lo que lo voy a repetir en caso de que se te escapara las primeras veces: *en Python, todo es un objeto*. Las cadenas son objetos, las listas son objetos. Las funciones son objetos. Las clases son objetos. Las instancias de las clases son objetos. E incluso los módulos son objetos.

1.6. Indentar código

Las funciones de Python no tienen begin o end, y tampoco existen llaves que marquen donde comienza y acaba el código de una función. El único delimitador es el símbolo de los dos puntos (:) y el propio indentado del código.

```
1  def tamanyo_aproximado(tamanyo, un_kilobyte_es_1024_bytes=True):
2      if tamanyo < 0:
3          raise ValueError('El número debe ser no negativo')
4
5      multiplo = 1024 if un_kilobyte_es_1024_bytes else 1000
6      for sufijo in SUFIJO[multiplo]:
7          tamanyo /= multiplo
8          if tamanyo < multiplo:
9              return '{0:.1f} {1}'.format(tamanyo, sufijo)
10
11     raise ValueError('número demasiado grande')
```

1. *Línea 1:* Los bloques de código se definen por su indentado. Por "bloque de código" se entiende lo siguiente: funciones, sentencias if, bucles for, bucles while y similar. Al indentar se inicia el bloque y al desindentar se finaliza. No existen llaves, corchetes o palabras clave para iniciar y finalizar un bloque de forma explícita. Esto implica que los espacios en blanco son significativos, y deben ser consistentes. En este ejemplo, el código de la función está indentado con cuatro espacios. No es necesario que sean cuatro, pero sí que sea consistente y siempre sean los mismos. La primera línea que no esté indentada deliminta el final de la función.

2. *Línea 2:* En Python, la sentencia if debe contener un bloque de código. Si la expresión que sigue al if es verdadera[6] se ejecuta el bloque indentado que contiene el if, en caso contrario lo que se ejecuta es el bloque contenido en el else (si existe). Observa la ausencia de paréntesis alrededor de la expresión.

3. *Línea 3:* Esta línea se encuentra *dentro* del bloque de código del if. La sentencia raise elevará una excepción (del tipo ValueError, pero únicamente si tamanyo ¡0.

4. *Línea 4:* Esta línea *no* marca el final de la función. Las líneas que estan completamente en blanco no cuentan. Únicamente sirven para hacer más legible el código, pero no cuentan como delimitadores de código. La función continúa en la línea siguiente.

[6] Si el resultado de evaluarla es **True**.

5. *Línea 6:* El bucle for también marca el comienzo de un bloque de código. Los bloques pueden contener múltiples líneas, siempre que estén indentadas con el mismo número de espacios. Este bucle for contiene tres líneas de código en él. No existe ninguna otra sintáxis especial para los bloques de varias líneas. Basta con indentar y... ¡seguir adelante con el trabajo!

Después de algunas protestas iniciales e insidiosas analogías con Fortran, seguro que harás las paces con esta forma de marcar los bloques de código y comenzarás a apreciar sus beneficios. Uno de los mayores beneficios es que todos los programas Python tienen un formato similar, al ser la indentación un requisito del lenguaje y no un elemento de estilo. La consecuencia inmediata es que los programas Python son más fáciles de leer y comprender por parte de una persona diferente de su programador[7].

Python utiliza los saltos de línea para separar las sentencias y los dos puntos y la indentación para separar los bloques de código. C++ y Java utilizan puntos y coma para separar sentencias y llaves para separar los bloques de código.

1.7. Excepciones

Las excepciones están en todas partes en Python. Prácticamente todos los módulos de la librería estándar las utilizan, y el propio lenguaje las lanza en muchas circunstancias. Las verás una y otra vez a lo largo del libro.

¿Qué es una excepción? Normalmente es un error, una indicación de que algo fue mal (No todas las excepciones son errores, pero no te preocupes por eso ahora). Algunos lenguajes de programación fomentan que se retornen códigos de error en las funciones, que los programadores tendrán que *chequear*. Python fomenta el uso de las excepciones, que los programadores tienen que *capturar* y *manejar*.

Cuando sucede un error se muestra en la consola de Python algunos detalles de la excepción y cómo se produjo. A esto se le llama *excepción sin capturar*. Cuando la excepción se generó, Python no encontró un trozo de código que estuviera previsto que la capturase y respondiera en consecuencia, por eso la excepción se fue *elevando* hasta llegar al nivel más alto en la consola, la cual muestra alguna información útil para la

Al contrario que Java, las funciones de Python no declaran las excepciones que podrían elevar. Te corresponde a ti determinar las excepciones que pueden suceder y necesitas capturar.

[7] ¡o por el propio programador después de unos meses!

depuración del código y finaliza. Si esto sucede
en la consola no es excesivamente preocupante, pero si le sucede a tu programa en plena ejecución, el programa finalizaría de forma incontrolada y no se capturase la excepción. Puede que sea lo que quieras, pero puede que no.

El hecho de que suceda una excepción no implica necesariamente que el programa tenga que fallar. Las excepciones se pueden *manejar*. Algunas veces una excepcion sucede porque tienes un error en el código (como por ejemplo, acceder al valor de una variable que no existe), pero en otras ocasiones puedes anticiparlo. Si vas a abrir un fichero, puede que no exista. Si vas a importar un módulo, puede que no esté instalado. Si te vas a conectar a una base de datos, puede que no esté disponible, o puede que no tengas las credenciales necesarias para acceder a ella. Si sabes que una línea de código puede *elevar* una excepción, deberías *manejar* la excepción utilizando el bloque try...except.

> Python utiliza bloques try...except para manejar excepciones, y la sentencia raise para generarlas. Java y C++ utilizan bloques try...catch para manejarlas y la sentencia throw para generarlas.

La función tamanyo_aproximado() eleva excepciones por dos causas: el tamaño que se pasa es mayor que el previsto, o si es menor que cero.

```
1     if tamanyo < 0:
2         raise ValueError('El número debe ser no negativo')
```

La sintaxis para elevar una excepción es muy simple. Utiliza la sentencia raise, seguida del nombre de la excepción y, opcionalmente, se le pasa como parámetro una cadena de texto que sirve para propósitos de depuración. La sintaxis es parecida a la de llamar a una función [8].

> No necesitas manejar las excepciones en la función que las eleva. Si no se manejan en la función que las eleva, las excepciones pasan a la función que la llamó, luego a la que llamó a esa, y así sucesivamente a través de toda la "pila de llamadas". Si una excepción no se manejase en ninguna función, el programa fallará y finalizará, Python imprimirá una *traza* del error, y punto. Puede que fuese lo que querías o no, depende de lo que pretendieras, ¡que para eso eres el programador!

[8] Las excepciones son objetos, como todo en Python ¿recuerdas?. Para implementarlas se utilizan clases (class) de objetos. Al ejecutar en este caso la sentencia raise, en realidad se está creando una instancia de la clase ValueError y pasándole la cadena "El número debe ser no negativo" al método de inicialización. ¡Pero nos estamos adelantando!

1.7.1. Capturar errores al importar

Una de las excepciones internas de Python es ImportError, que se eleva cuando intentas importar un módulo y falla. Esto puede suceder por diversas causas, pero la más simple es que el módulo no exista en tu camino de búsqueda. Puedes utilizar esta excepción para incluir características opcionales a tu programa. Por ejemplo, la librería chardet que aparece en el capítulo ?? autodetecta la codificación de caracteres. Posiblemente tu programa quiera utilizar esta librería si está instalada, pero continuar funcionando si no lo está. Para ello puedes utilizar un bloque try...except.

```
1 try:
2     import chardet
3 except ImportError:
4     chardet = None
```

Posteriormente, en el código, puedes consultar la presencia de la librería con una simple sentencia if:

```
1 if chardet:
2     # hacer algo
3 else:
4     # seguir de todos modos
```

Otro uso habitual de la excepcion ImportError es cuando dos módulos implementan una API[9] común, pero existe preferencia por uno de ellos por alguna causa (tal vez sea más rápida, o use menos memoria). Puedes probar a importar un módulo y si falla cargar el otro. Por ejemplo, en el capítulo 12 sobre XML se habla de dos módulos que implementan una API común, denominada ElementTree API. El primero lxml.etree, es un módulo desarrollado por terceros que requiere descargarlo e instalarlo tú mismo. El segundo, xml.etree.ElementTree, es más lento pero forma parte de la librería estándar de Python 3.

```
1 try:
2     from lxml import etree
3 except ImportError:
4     import xml.etree.ElementTree as etree
```

Al final de este bloque try...except, has importando *algún* modulo y lo has llamado etree. Puesto que ambos módulos implementan una API común, el resto del código no se tiene que preocupar de qué módulo se ha cargado[10]. Asimismo, como el módulo que se haya importado termina llamándose etree, el resto del código no

[9] Application Programming Interface. Interfaz de programación de aplicaciones.

[10] Nota del Traductor:Al implementar la misma API ambos módulos se *comportan* igual por lo que son indistingubles en cuanto a funcionamiento. Así, el resto del código puede funcionar sin conocer qué módulo se ha importado realmente.

tiene que estar repleto de sentencias if para llamar a diferentes módulos con diferente nombre.

1.8. Variables sin declarar

Échale otro vistazo a la siguiente línea de código de la función tamanyo_aproximado:

```
1 |     multiplo = 1024 if un_kilobyte_es_1024_bytes else 1000
```

La variable multiplo no se ha declarado en ningún sitio, simplemente se le asigna un valor. En Python es correcto. Lo que no te dejará hacer nunca Python es referenciar a una variable a la que nunca le has asignado un valor. Si intentas hacerlo se elevará la excepción NameError.

```
1 >>> x
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4 NameError: name 'x' is not defined
5 >>> x = 1
6 >>> x
7 1
```

¡Le darás las gracias frecuentemente a Python por avisarte!

1.9. Mayúsculas y minúsculas

En Python, todos los nombres distinguen mayúsculas y minúsculas: los nombres de variables, de funciones, de módulos, de excepciones. De forma que no es el mismo nombre si cambia alguna letra de mayúscula a minúscula o viceversa.

```
>>> un_entero = 1
>>> un_entero
1
>>> UN_ENTERO
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'UN_ENTERO' is not defined
>>> Un_Entero
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'Un_Entero' is not defined
>>> un_enteRo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'un_enteRo' is not defined
```

Y así siempre si pruebas todas las combinaciones posibles.

1.10. Ejecución de scripts

Los módulos de Python son objetos, por lo que tienen propiedades muy útiles. Puedes utilizar alguna de ellas para probar tus módulos de una forma sencilla. Para ello puedes incluir un bloque de código especial que se ejecute cuando arrancas el fichero desde la línea de comando. Observa las últimas líneas de parahumanos.py:

> Todo lo que existe en Python es un Objeto

```
if __name__ == '__main__':
    print(tamanyo_aproximado(1000000000000, False))
    print(tamanyo_aproximado(1000000000000))
```

> Como en C, Python utiliza == para las comparaciones y = para las asignaciones. Al contrario que C, Python no permite la asignación "en línea", por lo que no es posible asignar un valor por accidente cuando tu intención fuese comparar.

¿Qué es lo que hace este if tan especial? Como los módulos son objetos, tienen propiedades, y una de las propiedades de los módulos es __name__. El valor de la propiedad __name__ depende de la forma en la que estés utilizando el módulo. Si importas el módulo con la sentencia import el valor que contiene __name__ es el nombre del fichero del módulo sin la extensión.

```
>>> import parahumanos
>>> parahumanos.__name__
'parahumanos'
```

Pero también puedes ejecutar directamente el módulo como un programa autónomo, en cuyo caso __name__ contiene el valor especial __main__. En el ejemplo, Python evaluará la sentencia if, la expresión será verdadera y ejecutará el bloque de código contenido en el if. En este caso, imprimir dos valores:

```
jmgaguilera@acerNetbook:~/inmersionEnPython3/src$ python3 parahumanos.py
1.0 TB
931.3 GiB
```

¡Y así queda explicado tu primer programa Python!

1.11. Lecturas complementarias

- PEP 257: Docstring Conventions "Convenciones para escribir docstring". Explica lo que distingue un buen docstring de un gran docstring.
- Tutorial de Python: Cadenas de texto para documentación también aborda la materia.
- PEP 8: Guía de estilo para codificación en Python comenta cual es el estilo recomendado de indentación.
- Manual de referencia de Python explica lo que significa decir que todo en Python es un objeto, porque algunas personas son algo pedantes y les gusta discutir largo y tendido sobre ese tipo de cosas.

Capítulo 2

Tipos de dato nativos

Nivel de dificultad:◆ ◆ ◇ ◇ ◇

"La curiosidad es la base de toda la filosofía,
las preguntas alimentan su progreso,
la ignorancia su fin."
—Michel de Montaigne

2.1. Inmersión

Aparta tu primer programa en Python durante unos minutos, y vamos a hablar sobre tipos de dato. En Python cada valor que exista, tiene un tipo de dato, pero no es necesario declarar el tipo de las variables. ¿Como funciona? Basado en cada asignación a la variable, Python deduce el tipo que es y lo conserva internamente.

Python proporciona muchos tipos de dato nativos. A continuación se muestran los más importantes:

1. **Booleanos:** Su valor es True o False.
2. **Números:** Pueden ser enteros (1, 2, 3,...), flotantes (1.1, 1.2, 1.3,...)[1], fracciones (1/2, 1/3, 2/3,...), o incluso números complejos ($i = \sqrt{-1}$).
3. **Cadenas:** Son secuencias de caracteres Unicode, por ejemplo, un documento HTML.

[1]Nota del traductor: los números decimales se representan utilizando *punto decimal*. Aunque en español utilizamos la *coma decimal* en este libro usamos el punto decimal por ser el formato que se requiere en Python.

4. **Bytes y arrays de bytes:** por ejemplo, un fichero de imágenes JPEG.
5. **Listas:** Son secuencias ordenadas de valores.
6. **Tuplas:** Son secuencias ordenadas e inmutables de valores.
7. **Conjuntos:** Son "bolsas" de valores sin ordenar.
8. **Diccionarios:** Son "bolsas" de sin ordenar de parejas clave-valor. Es posible buscar directamente por clave.

Aparte de estos, hay bastantes más tipos. Todo es un objeto en Python, por lo que existen tipos *module, function, class, method, file*, e incluso *compiled code*[2]. Ya has visto alguno de ellos en el capítulo anterior. En el capítulo **??** aprenderás las clases, y en el capítulo 11 los ficheros (también llamados archivos).

Las cadenas y bytes son suficientemente importantes —y complejas— como para merecer un capítulo aparte. Vamos a ver los otros tipos de dato en primer lugar.

2.2. Booleanos

El tipo de datos booleano solamente tiene dos valores posibles: verdadero o fals. Python dispone de dos constantes denominadas True y False, que se pueden utilizar para asignar valores booleanos directamente. Las expresiones también se pueden evaluar a un valor booleano. En algunos lugares (como las sentencias if, Python espera una expresión que se pueda evaluar a un valor booleano. Estos sitios se denominan *contextos booleanos*. Puedes utilizar casi cualquier expresión en un contexto booleano, Python intentará determinar si el resultado puede ser verdadero o falso. Cada tipo de datos tiene sus propias reglas para identificar qué valores equivalen a verdadero y falso en un contexto booleano (Esto comenzará a tener un sentido más claro para ti cuando veas algunos ejemplos concretos).

En la práctica, puedes utilizar casi cualquier expresión en un contexto booleano.

Por ejemplo:

```
    if tamanyo < 0:
        raise ValueError('el número debe ser no negativo')
```

[2]Nota del traductor: Son tipos de dato del lenguaje Python que representan a: módulos, funciones, clases, métodos, ficheros y código compilado.

La variable tamanyo contiene un valor entero, 0 es un entero, y ¡ es un operador numérico. El resultado de la expresión es siempre un valor de tipo booleano. Puedes comprobarlo en la consola interactiva de Python:

```
>>> tamanyo = 1
>>> tamanyo < 0
False
>>> tamanyo = 0
>>> tamanyo < 0
False
>>> tamanyo = -1
>>> tamanyo < 0
True
```

Debido a la herencia que se conserva de Python 2, los booleanos se pueden tratar como si fuesen números. True es 1 y False es 0.

```
>>> True + True
2
>>> True - False
1
>>> True * False
0
>>> True / False
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: int division or modulo by zero
```

Aún así, no hagas esto. ¡Olvida incluso que lo he mencionado![3]

2.3. Números

Los números son maravillosos. Tienes un montón donde elegir. Python proporciona enteros y números de coma flotante, pero no existen declaraciones de tipo para distinguirlos. Python los distingue por la existencia o no del punto decimal[4].

[3]Nota del traductor: se trata de una práctica heredada de Python 2 pero que no puede considerarse buena práctica de programación.

[4]Nota del traductor: En español se dice "coma decimal", como vamos a mostrar puntos decimales en todo el libro, por coherencia se utilizará también el término "punto decimal".

```
1  >>> type(1)
2  <class 'int'>
3  >>> isinstance(1, int)
4  True
5  >>> 1 + 1
6  2
7  >>> 1 + 1.0
8  2.0
9  >>> type(2.0)
10 <class 'float'>
```

1. *Línea 1:* la función type() permite consultar el tipo de cualquier valor o variable. Como era de esperar 1 es un valor de tipo int.

2. *Línea 3:* la función isinstance() permite chequear si un valor o variable es de un tipo determinado.

3. *Línea 5:* La suma de dos valores de tipo int da como resultado otro valor de tipo int.

4. *Línea 7:* La suma de un valor int con otro de tipo float da como resultado un valor de tipo float. Python transforma el valor entero en un valor de tipo float antes de hacer la suma. El valor que se devuelve es de tipo float.

2.3.1. Convertir enteros en flotantes y viceversa

Como acabas de ver, algunos operadores (como la suma) convierten los números enteros en flotantes si es necesario. También puedes convertirlos tú mismo.

```
1  >>> float(2)
2  2.0
3  >>> int(2.0)
4  2
5  >>> int(2.5)
6  2
7  >>> int(-2.5)
8  -2
9  >>> 1.12345678901234567890
10 1.1234567890123457
11 >>> type(1000000000000000)
12 <class 'int'>
```

1. *Línea 1:* Utilizando la función float() puedes convertir explícitamente un valor de tipo int en float.

2. *Línea 3:* Como era de prever, la conversión inversa se hace utilizando la función int().

3. *Línea 5:* La función int() trunca el valor flotante, no lo redondea.

4. *Línea 7:* La función int() trunca los valores negativos hacia el 0. Es una verdadera función de truncado, no es una función de suelo[5].

5. *Línea 9:* En Python, la precisión de los números de punto flotante alcanza 15 posiciones decimales.

6. *Línea 11:* En Python, la longitud de los números enteros no está limitada. Pueden tener tantos dígitos como se requieran.

   Python 2 tenía dos tipos separados int y long. El tipo int estaba limitado por el sistema sys.maxint, siendo diferente según la plataforma, pero usualmente era $2^{32} - 1$. Python 3 tiene un único tipo entero que, en su mayor parte, equivale al tipo long de Python 2. Para conocer más detalles consulta PEP 237.

2.3.2. Operaciones numéricas habituales

Puedes hacer muchos tipos de cálculos con números.

```
1  >>> 11 / 2
2  5.5
3  >>> 11 // 2
4  5
5  >>> -11 // 2
6  -6
7  >>> 11.0 // 2
8  5.0
9  >>> 11 ** 2
10 121
11 >>> 11 % 2
12 1
```

1. *Línea 1:* El operador / efectúa una división en punto flotante. El resultado siempre es de tipo float, incluso aunque ambos operadores (dividendo y divisor) sean int.

[5] Nota del traductor: en inglés "floor function", que redondea siempre al entero menor, por lo que el número -2.5 sería convertido a -3 en el caso de aplicarle una función de *suelo*

2. *Línea 3:* El operador // efectúa una división entera algo extraña. Cuando el resultado es positivo, el resultado es int truncado sin decimales (no redondeado).

3. *Línea 5:* Cuando el operador // se usa para dividir un número negativo el resultado se redondea hacia abajo al entero más próximo (en este caso el resultado de la división sería -5.5, que redondeado es -5).

4. *Línea 7:* El operador ** significa "elevado a la potencia de". 11^2 es 121.

5. *Línea 9:* El operador % devuelve el resto de la división entera. 11 dividido entre 2 es 5 con un resto de 1, por lo que el resultado en este caso es 1.

> En Python 2, el operador / se usaba para representar a la división entera, aunque mediante una directiva de Python 2, podías hacer que se comportase como una división de punto flotante. En Python 3, el operador / siempre es una división de punto flotante. Para consultar más detalles puedes mirar PEP 238.

2.3.3. Fracciones

Python no está limitado a números enteros y de punto flotante. También puede aplicar toda esa matemática que aprendiste en el instituto y que luego rápidamente olvidaste.

```
1  >>> import fractions
2  >>> x = fractions.Fraction(1, 3)
3  >>> x
4  Fraction(1, 3)
5  >>> x * 2
6  Fraction(2, 3)
7  >>> fractions.Fraction(6, 4)
8  Fraction(3, 2)
9  >>> fractions.Fraction(0, 0)
10 Traceback (most recent call last):
11   File "<stdin>", line 1, in <module>
12   File "fractions.py", line 96, in __new__
13     raise ZeroDivisionError('Fraction(%s, 0)' % numerator)
14 ZeroDivisionError: Fraction(0, 0)
```

1. *Línea 1:* Para comenzar a utilizar fracciones hay que importar el módulo fractions.

2. *Línea 2:* Para definir una fracción crea un objeto Fraction y pásale el numerador y denominador.

3. *Línea 5:* Con las fracciones puedes efectuar los cálculos matemáticos habituales. Estas operaciones devuelven un objeto Fraction, 2*(1/2)=(2/3).

4. *Línea 7:* El objeto Fraction reduce automáticamente las fracciones: (6/4) = (3/2).

5. *Línea 9:* Python tiene el buen sentido de no crear una fracción con el denominador a cero.

2.3.4. Trigonometría

También puedes hacer cálculos trigonométricos en Python.

```
1 >>> import math
2 >>> math.pi
3 3.1415926535897931
4 >>> math.sin(math.pi / 2)
5 1.0
6 >>> math.tan(math.pi / 4)
7 0.99999999999999989
```

1. *Línea 2:* El módulo math tiene definida una constante que almacena el valor del número π, la razón de la circunferencia de un círculo respecto de su diámetro.

2. *Línea 4:* En el módulo math se encuentran todas las funciones trigonométricas básicas, incluidas sin(), cos(), tan() y variantes como asin().

3. *Línea 6:* De todos modos ten en cuenta que Python no tiene precisión infinita, tan(π/4) debería devolver 1.0, no 0.99999999999999989.

2.3.5. Números en un contexto booleano

> El valor cero es equivalente a falso y los valores distintos de cero son equivalentes a verdadero.

Los números se pueden utilizar en contextos booleanos, como en la sentencia if. El valor cero es equivalente a falso y los valores distintos de cero son equivalentes a verdadero.

```
1  >>> def es_true(anything):
2  ...     if anything:
3  ...       print("sí, es true")
4  ...     else:
5  ...       print("no, es false")
6  ...
7  >>> es_true(1)
8  sí, es true
9  >>> es_true(-1)
10 sí, es true
11 >>> es_true(0)
12 no, es false
13 >>> es_true(0.1)
14 sí, es true
15 >>> es_true(0.0)
16 no, es false
17 >>> import fractions
18 >>> es_true(fractions.Fraction(1, 2))
19 sí, es true
20 >>> es_true(fractions.Fraction(0, 1))
21 no, es false
```

1. *Línea 1:* ¿Sabías que puedes definir tus propias funciones en la consola interactiva de Python? Simplemente pulsa INTRO al final de cada línea, y termina pulsando un último INTRO en una línea en planco para finalizar la definición de la función.

2. *Línea 7:* En un contexto booleano, como el de la sentencia if, los números enteros distintos de cero se evalúan a True; el número cero se evalúa a False.

3. *Línea 13:* Los números en punto flotante distintos de cero son True; 0.0 se evalúa a False. ¡Ten cuidado con este caso! Al más mínimo fallo de redondeo (que no es imposible, como has visto en el apartado anterior) Python se encontraría comprobando el número 0.0000000000001 en lugar del 0 y retornaría True.

4. *Línea 18:* Las fracciones también se pueden utilizar en un contexto booleano. Fraction(0, n) se evalúa a False para cualquier valor de n. Todas las otras fracciones se evalúan a True.

2.4. Listas

El tipo de datos List es el más utilizado en Python. Cuando digo "lista", puede que pienses en un "array[6], cuyo tamaño he declarado anteriormente a su uso, que únicamente puede contener elementos del mismo tipo". No pienses eso, las listas son mucho más *guays*.

> Una lista de Python es como un array de Perl 5. En Perl 5 las variables que almacenan arrays siempre comienzan con el carácter @. En Python las variables se pueden nombrar como se quiera, ya que Python mantiene el tipo de datos internamente.

> Una lista de Python es mucho más que un array de Java (aunque puede utilizarse como si lo fuese si eso es lo que quieres). Una analogía mejor sería pensar en la clase ArrayList de Java, que puede almacenar un número arbitrario de objetos y expandir su tamaño dinámicamente al añadir nuevos elementos.

2.4.1. Crear una lista

Crear una lista es fácil: utiliza unos corchetes para para delimitar una lista de valores separados por coma.

```
1  >>> lista = ['a', 'b', 'jmgaguilera', 'z', 'ejemplo']
2  >>> lista
3  ['a', 'b', 'jmgaguilera', 'z', 'ejemplo']
4  >>> lista[0]
5  'a'
6  >>> lista[4]
7  'ejemplo'
8  >>> lista[-1]
9  'ejemplo'
10 >>> lista[-3]
11 'jmgaguilera'
```

1. *Líneas 1 a 3:* Primero definimos una lista de cinco elementos. Observa que mantiene el orden original. No es por casualidad. Una lista es un conjunto ordenado de elementos.

[6] matriz de una o más dimensiones

2. *Línea 4:* Se puede acceder a los elementos de la lista como en el caso de los arrays de Java, teniendo en cuenta que el primer elemento se numera como cero. El primer elemento de cualquier lista no vacía es lista[0].

3. *Línea 6:* El último elemento de esta lista de cinco elementos es lista[4], puesto que los elementos se indexan contando desde cero.

4. *Línea 8:* Si se usan índices con valor negativo se accede a los elementos de la lista contando desde el final. El último elemento de una lista siempre se puede indexar utilizando lista[-1].

5. *Línea 10:* Si los números negativos en los índices te resultan confusos, puedes pensar de esta forma: lista[-n] == lista[len(lista)-n]. Por eso, en esta lista, lista[-3] == lista[5 - 3] == lista[2].

2.4.2. Partición de listas

> lista[0] es el primer elemento de la lista.

Una vez has definido una lista, puedes obtener cualquier parte de ella como una nueva lista. A esto se le llama *particionado*[7] de la lista.

```
1  >>> lista
2  ['a', 'b', 'jmgaguilera', 'z', 'ejemplo']
3  >>> lista[1:3]
4  ['b', 'jmgaguilera']
5  >>> lista[1:-1]
6  ['b', 'jmgaguilera', 'z']
7  >>> lista[0:3]
8  ['a', 'b', 'jmgaguilera']
9  >>> lista[:3]
10 ['a', 'b', 'jmgaguilera']
11 >>> lista[3:]
12 ['z', 'ejemplo']
13 >>> lista[:]
14 ['a', 'b', 'jmgaguilera', 'z', 'ejemplo']
```

1. *Línea 3:* Puedes obtener una parte de una lista especificando dos índices. El valor de retorno es una nueva lista que contiene los elementos de la lista original, en orden, comenzando en el elemento que estaba en la posición del primer índice (en este caso lista[1]), hasta el elemento anterior al indicado por el segundo índice (en este caso lista[3]).

[7] En inglés: slicing.

2. *Línea 5:* El particionado de listas también funciona si uno o ambos índices son negativos. Si te sirve de ayuda puedes imaginártelo así: leyendo la lista de izquierda a derecha, el primer índice siempre especifica el primer elemento que quieres obtener y el segundo índice el primer elemento que no quieres obtener. El valor de retorno es una lista con todos los elementos que están entre ambos índices.

3. *Línea 7:* Los índices de las listas comienzan a contar en cero, por eso un particionado de lista[0:3] devuelve los primeros tres elementos de la lista, comenzando en lista[0], pero sin incluir lista[3].

4. *Línea 9:* Si el primer índice es cero, puedes omitirlo. Python lo deducirá. Por eso lista[:3] es lo mismo que lista[0:3].

5. *Línea 11:* De igual forma, si el segundo índice es la longitud de la cadena, puedes omitirlo. Por eso, en este caso, lista[3:] es lo mismo que lista[3:5], al tener esta lista cinco elementos. Existe una elegante simetría aquí. En esta lista de cinco elementos lista[:3] devuelve los 3 primeros elementos y lista[3:] devuelve una lista con los restantes. De hecho, lista[:n] siempre retornará los n primeros elementos y lista[n:] los restantes, sea cual sea el tamaño de la lista.

6. *Línea 13:* Si se omiten ambos índices, se obtiene una nueva lista con todos los elementos de la lista original. Es una forma rápida de hacer una copia de una lista.

2.4.3. Añadir elementos a una lista

Existen cuatro maneras de añadir elementos a una lista.

```
1  >>> lista = ['a']
2  >>> lista = lista + [2.0, 3]
3  >>> lista
4  ['a', 2.0, 3]
5  >>> lista.append(True)
6  >>> lista
7  ['a', 2.0, 3, True]
8  >>> lista.extend(['cuatro', 'omega'])
9  >>> lista
10 ['a', 2.0, 3, True, 'cuatro', 'omega']
11 >>> lista.insert(0, 'omega')
12 >>> lista
13 ['omega', 'a', 2.0, 3, True, 'cuatro', 'omega']
```

1. *Línea 2:* El operador + crea una nueva lista a partir de la concatenación de otras dos. Una lista puede contener cualquier número de elementos, no hay

límite de tamaño (salvo el que imponga la memoria disponible). Si embargo, si la memoria es importante, debes tener en cuenta que la concatenación de listas crea una tercera lista en memoria. En este caso, la nueva lista se asigna inmediatamente a la variable lista. Por eso, esta línea de código se efectúa en dos pasos —concatenación y luego asignación— que puede (temporalmente) consumir mucha memoria cuando las listas son largas.

2. *Línea 3:* Una lista puede contener elementos de cualquier tipo, y cada elemento puede ser de un tipo diferente. Aquí tenemos una lista que contiene una cadena de texto, un número en punto flotante y un número entero.

3. *Línea 5:* El método append() añade un nuevo elemento, único, al final de la lista (¡Ahora ya tenemos *cuatro* tipos de dato diferentes en la lista!).

4. *Línea 8:* Las listas son clases. "Crear" una lista realmente consiste en instanciar una clase. Por eso las listas tienen métodos que sirven para operar con ellas. El método extend() toma un parámetro, una lista, y añade cada uno de sus elementos a la lista original.

5. *Línea 11:* El método insert() inserta un único elemento a la lista original. El primer parámetro es el índice del primer elemento de la lista original que se desplazará de su posición para añadir los nuevos. Los elementos no tienen que ser únicos; por ejemplo, ahora hay dos elementos separados en la lista cuyo valor es "omega": el primer elemento lista[0] y el último elemento lista[6].

   La llamada al método lista.insert(0,valor) es equivalente a la función unshift() de Perl. Añade un elemento al comienzo de la lista, y los restantes elementos se desplazan para hacer sitio.

   Vamos a ver más de cerca la diferencia entre append() y extend().

```
1  >>> lista = ['a', 'b', 'c']
2  >>> lista.extend(['d', 'e', 'f'])
3  >>> lista
4  ['a', 'b', 'c', 'd', 'e', 'f']
5  >>> len(lista)
6  6
7  >>> lista[-1]
8  'f'
9  >>> lista.append(['g', 'h', 'i'])
10 >>> lista
11 ['a', 'b', 'c', 'd', 'e', 'f', ['g', 'h', 'i']]
12 >>> len(lista)
13 7
14 >>> lista[-1]
15 ['g', 'h', 'i']
```

1. *Línea 2:* El método extend() recibe un único parámetro, que siempre es una lista, y añade cada uno de sus elementos al final de la lista original lista.

2. *Línea 5:* Si una lista de tres elementos se extiende con una lista de otros tres elementos, la lista resultante tiene seis elementos.

3. *Línea 9:* Por otra parte, el método append() recibe un único parámetro, que puede ser de cualquier tipo. En este caso estamos ejecutando el método pasándole una lista de tres elementos.

4. *Línea 12:* Si partes de una lista de seis elementos y añades otra lista a ella, finalizas con una lista de... siete elementos. ¿Porqué siete? Porque el último elemento (que acabas de añadir) *es en sí mismo una lista*. Una lista puede contener datos de cualquier tipo, incluídas otras listas. Puede que sea lo que quieras o puede que no. Pero es lo que le has pedido a Python al ejecutar append() con una lista como parámetro.

2.4.4. Búsqueda de valores en una lista

```
1  >>> lista = ['a', 'b', 'nuevo', 'mpilgrim', 'nuevo']
2  >>> lista.count('nuevo')
3  2
4  >>> 'nuevo' in lista
5  True
6  >>> 'c' in lista
7  False
8  >>> lista.index('mpilgrim')
9  3
10 >>> lista.index('nuevo')
11 2
12 >>> lista.index('c')
13 Traceback (innermost last):
14   File "<interactive input>", line 1, in ?
15 ValueError: list.index(x): x not in list
```

1. *Línea 2:* Como te puedes imaginar, el método count() devuelve el número de veces que aparece un valor específico —el parámetro— en la lista.

2. *Línea 4:* Si lo único que quieres saber es si un valor se encuentra o no en la lista, el operador in es ligeramente más rápido que el método count(). El operador in devuelve True o False, no indica en qué lugar de la lista se encuentra el elemento, ni el número de veces que aparece.

3. *Línea 8:* Si necesitas conocer el lugar exacto en el que se encuentra un valor dentro de la lista debes utilizar el método index(). Por defecto, este método buscará en toda la lista, aunque es posible especificar un segundo parámetro para indicar el lugar de comienzo (0 es el primer índice), e incluso, un tercer elemento para indicar el índice en el que parar la búsqueda.

4. *Línea 10:* El método index() encuentra la *primera* ocurrencia del valor en la lista. En este caso el valor "nuevo" aparece dos veces, en la posición 2 y en la 4, el método devuelve la posición de la primera ocurrencia: 2.

5. *Línea 12:* Puede que no esperases, pero el método index() eleva una excepción ValueError cuando no es capaz de encontrar el elemento en la lista.

¡Espera un momento! ¿Qué significa eso? Pues lo que he dicho: el método index() eleva una excepción si no es capaz de encontrar el valor en la lista. Esto es diferente de la mayoría de los lenguajes de programación que suelen devolver algún índice no válido, como por ejemplo, -1. Aunque al principio te pueda parecer algo desconcertante, creo que con el tiempo llegarás a apreciarlo. Significa que tu programa fallará en la fuente del problema, en lugar de fallar más tarde en algún otro lugar por no haber contemplado la posibilidad de que un elemento no se encontrara en la lista. Recuerda que -1 es un valor de índice válido. Si el método index() devolviera -1... ¡las sesiones de depuración serían bastante complicadas!

2.4.5. Eliminar elementos de una lista

> Las listas nunca tienen huecos

Las listas se expanden y contraen de forma automática. Ya has visto como expandirlas. Existen varios modos de eliminar elementos de una lista.

```
1 >>> lista = ['a', 'b', 'nuevo', 'mpilgrim', 'nuevo']
2 >>> lista[1]
3 'b'
4 >>> del lista[1]
5 >>> lista
6 ['a', 'nuevo', 'mpilgrim', 'nuevo']
7 >>> lista[1]
8 'nuevo'
```

1. *Línea 4:* Para eliminar un elemento de una lista puedes utilizar la sentencia del.

2. *Línea 7:* Si intentas acceder al elemento en la posición 1 después de borrar el índice 1 *no* da error. Después de borrar un elemento, todos los elementos que iban detrás de él se desplazan a la izquierda para "rellenar el vacío" que dejó el elemento eliminado.

¿Y si no conoces la posición del elemento? No hay problema, en vez de la posición, puedes utilizar el valor del elemento para eliminarlo.

```
1 >>> lista.remove('nuevo')
2 >>> lista
3 ['a', 'mpilgrim', 'nuevo']
4 >>> lista.remove('nuevo')
5 >>> lista
6 ['a', 'mpilgrim']
7 >>> lista.remove('nuevo')
8 Traceback (most recent call last):
9   File "<stdin>", line 1, in <module>
10 ValueError: list.remove(x): x not in list
```

1. *Línea 1:* Puedes eliminar un elemento de una lista utilizando el método remove(). Este método recibe como parámetro un valor y elimina la primera ocurrencia de ese valor en la lista. Como antes, todos los elementos a la derecha del eliminado, se desplazan a la izquierda para "rellenar el vacío", puesto que las listas nunca tienen huecos.

2. *Línea 4:* Puedes llamar al método remove() tantas veces como sea necesario. Pero si se intenta eliminar un valor que no se encuentre en la lista, el método elevará una excepción ValueError.

2.4.6. Eliminar elementos de una lista: ronda extra

Otro método de interés que tiene las listas es pop(), que permite eliminar elementos de una lista de un modo especial.

```
1  >>> lista = ['a', 'b', 'nuevo', 'mpilgrim']
2  >>> lista.pop()
3  'mpilgrim'
4  >>> lista
5  ['a', 'b', 'nuevo']
6  >>> lista.pop(1)
7  'b'
8  >>> lista
9  ['a', 'nuevo']
10 >>> lista.pop()
11 'nuevo'
12 >>> lista.pop()
13 'a'
14 >>> lista.pop()
15 Traceback (most recent call last):
16   File "<stdin>", line 1, in <module>
17 IndexError: pop from empty list
```

1. *Línea 2:* Cuando se llama sin parámetros, el método pop() elimina el último valor de la lista y *devuelve el valor eliminado.*

2. *Línea 6:* Es posible extraer cualquier elemento de una lista. Para ello hay que pasar el índice deseado al método pop(). Se eliminará el elemento indicado, los siguientes se moverán a la izquierda "rellenar el vacío" y se devuelve el valor recién eliminado.

3. *Línea 14:* Si llamas al método pop() con una lista vacía se eleva una excepción.

   El método pop() sin argumentos se comporta igual que la función pop() de Perl. Elimina el último valor de la lista y lo devuelve. Perl dispone de otra función, shift(), que eliminar el primer elemento y devuelve su valor; en Python es equivalente a lista.pop(0).

2.4.7. Listas en contextos booleanos

> Las listas vacías equivalen a falso, todas las demás a verdadero.

Puedes utilizar las listas en contextos booleanos, como en la sentencia if.

```
1  >>> def es_true(anything):
2  ...     if anything:
3  ...       print("sí, es true")
4  ...     else:
5  ...       print("no, es false")
6  ...
7  >>> es_true([])
8  no, es false
9  >>> es_true(['a'])
10 sí, es true
11 >>> es_true([False])
12 sí, es true
```

1. *Línea 7:* En un contexto booleano una lista vacía vale False.
2. *Línea 9:* Cualquier lista con al menos un elemento vale True.
3. *Línea 11:* Cualquier lista con al menos un elemento vale True. El valor de los elementos de la lista es irrelevante.

2.5. Tuplas

Una tupla es una lista inmutable. Una tupla no se puede modificar después de haberla creado.

```
1  >>> tupla = ("a", "b", "mpilgrim", "z", "ejemplo")
2  >>> tupla
3  ('a', 'b', 'mpilgrim', 'z', 'ejemplo')
4  >>> tupla[0]
5  'a'
6  >>> tupla[-1]
7  'ejemplo'
8  >>> tupla[1:3]
9  ('b', 'mpilgrim')
```

1. *Línea 1:* Las tuplas se definen de la misma forma que las listas. La única diferencia es que los elementos se cierran entre paréntesis en lugar de corchetes.
2. *Línea 4:* Los elementos de una tupla están ordenados como los de una lista. Los índices también comienzan a contar en cero, por lo que el primer elemento de una tupla siempre es tupla[0].
3. *Línea 6:* Los índices negativos cuentan desde el final de la tupla como en las listas.

4. *Línea 8:* El particionado también funciona como en las listas. Una partición de una tupla es una nueva tupla con los elementos seleccionados.

Lo que diferencia a las tuplas de las listas es que las primeras no se pueden modificar. En términos técnicos se dice que son inmutables. En términos prácticos esto significa que no tienen métodos que te permitan modificarlas. Las listas tienen métodos como append(), extend(), insert(), remove() y pop(). Las tuplas no tienen ninguno de estos métodos. Puedes particionar una tupla porque en realidad se crea una nueva tupla, y puedes consultar si contienen un valor determinado, y... eso es todo.

```
1  # continuación del ejemplo anterior
2  >>> tupla
3  ('a', 'b', 'mpilgrim', 'z', 'ejemplo')
4  >>> tupla.append("nuevo")
5  Traceback (innermost last):
6    File "<interactive input>", line 1, in ?
7  AttributeError: 'tupla' object has no attribute 'append'
8  >>> tupla.remove("z")
9  Traceback (innermost last):
10   File "<interactive input>", line 1, in ?
11 AttributeError: 'tupla' object has no attribute 'remove'
12 >>> tupla.index("ejemplo")
13 4
14 >>> "z" in tupla
15 True
```

1. *Línea 4:* No puedes añadir elementos a una tupla. No existen los métodos append() o extend().
2. *Línea 8:* No puedes eliminar elementos de una tupla. No existen los métodos remove() o pop().
3. *Línea 12:* Sí puedes buscar elementos en una tupla puesto que consultar no cambia la tupla.
4. *Línea 14:* También puedes utilizar el operador in para chequear si existe un elemento en la tupla.

¿Para qué valen las tuplas?

- Las tuplas son más rápidas que las listas. Si lo que defines es un conjunto estático de valores y todo lo que vas a hacer es iterar a través de ellos, lo mejor es que uses una tupla en lugar de una lista.

- Es más seguro, puesto que proteges contra escritura los datos que no necesitas modificar.
- Algunas tuplas se pueden utilizar como claves de diccionarios como veremos más adelante en el capítulo. Las listas nunca se pueden utilizar como claves de diccionarios.

Las tuplas se pueden convertir en listas y viceversa. La función interna tuple() puede recibir como parámetro una lista y devuelve una tupla con los mismos elementos que tenga la lista, y la función list() toma como parámetro una tupla y retorna una lista. En la práctica la función tuple() "congela" una lista, y la función list() "descongela" una tupla.

2.5.1. Tuplas en un contexto booleano

Las tuplas también se pueden utilizar en un contexto booleano:

```
1  >>> def es_true(anything):
2  ...     if anything:
3  ...       print("sí, es true")
4  ...     else:
5  ...       print("no, es false")
6  ...
7  >>> es_true(())
8  no, es false
9  >>> es_true(('a', 'b'))
10 sí, es true
11 >>> es_true((False,))
12 sí, es true
13 >>> type((False))
14 <class 'bool'>
15 >>> type((False,))
16 <class 'tuple'>
```

1. *Línea 7:* Una tupla vacía siempre vale false en un contexto booleano.
2. *Línea 9:* Una tupla con al menos un valor vale true.
3. *Línea 11:* Una tupla con al menos un valor vale true. El valor de los elementos es irrelevante. Pero ¿qué hace esa coma ahí?
4. *Línea 13:* Para crear una tupla con un único elemento, necesitas poner una coma después del valor. Sin la coma Python asume que lo que estás haciendo es poner un par de paréntesis a una expresión, por lo que no se crea una tupla.

2.5.2. Asignar varios valores a la vez

A continuación se observa una forma muy interesante de programar múltiples asignaciones en Python. Para ello utilizamos las tuplas:

```
1 >>> v = ('a', 2, True)
2 >>> (x, y, z) = v
3 >>> x
4 'a'
5 >>> y
6 2
7 >>> z
8 True
```

1. *Línea 2:* v es una tupla con tres elementos y (x, y, z) es una tupla con tres variables. Al asignar una tupla a la otra, lo que sucede es que cada una de las variables recoge el valor del elemento de la otra tupla que corresponde con su posición.

Esto tiene toda clase de usos. Supón que quieres asignar nombres a un rango de valores, puedes combinar la función range() con la asignación múltiple para hacerlo de una forma rápida:

```
1 >>> (LUNES, MARTES, MIERCOLES, JUEVES,
2 ... VIERNES, SABADO, DOMINGO) = range(7)
3 >>> LUNES
4 0
5 >>> MARTES
6 1
7 >>> DOMINGO
8 6
```

1. *Línea 1:* La función interna range() genera una secuencia de números enteros[8]. Las variables que vas a definir son LUNES, MARTES, etc)[9]. El módulo calendar define unas constantes enteras para cada día de la semana).

2. *Línea 3:* Ahora cada variable tiene un valor: LUNES vale 0, MARTES vale 1, etc.

También puedes utilizar la asignación múltiple para construir funciones que devuelvan varios valores a la vez. Simplemente devolviendo una tupla con los valores.

[8] Técnicamente construye un **iterador**, no una lista o tupla. Lo veremos más adelante.

[9] Este ejemplo procede del módulo calendar, que es un pequeño módulo que imprime un calendario, como el programa de UNIX cal

Desde el código que llama a la función se puede tratar el valor de retorno como una tupla o se puede asignar los valores individuales a unas variables. Muchas librerías estándares de Python hacen esto, incluido el módulo os, que utilizaremos en el siguiente capítulo.

2.6. Conjuntos

Un conjunto es una "bolsa" sin ordenar de valores únicos. Un conjunto puede contener simultáneamente valores de cualquier tipo de datos. Con dos conjuntos se pueden efectuar las típicas operaciones de unión, intersección y diferencia de conjuntos.

2.6.1. Creación de conjuntos

Comencemos por el principio, crear un conjunto es fácil.

```
1 >>> un_conjunto = {1}
2 >>> un_conjunto
3 {1}
4 >>> type(un_conjunto)
5 <class 'set'>
6 >>> un_conjunto = {1, 2}
7 >>> un_conjunto
8 {1, 2}
```

1. *Línea 1:* Para crear un conjunto con un valor basta con poner el valor entre llaves ().
2. *Línea 4:* Los conjuntos son clases, pero no te preocupes por ahora de esto.
3. *Línea 6:* Para crear un conjunto con varios elementos basta con separarlos con comas y encerrarlos entre llaves.

También es posible crear un conjunto a partir de una lista:

```
1 >>> una_lista = ['a', 'b', 'mpilgrim', True, False, 42]
2 >>> un_conjunto = set(una_lista)
3 >>> un_conjunto
4 {'a', False, 'b', True, 'mpilgrim', 42}
5 >>> una_lista
6 ['a', 'b', 'mpilgrim', True, False, 42]
```

1. *Línea 2:* Para crear un conjunto de una lista utiliza la función set()[10].

2. *Línea 3:* Como comenté anteriormente, un conjunto puede contener valores de cualquier tipo y está *desordenado*. En este ejemplo, el conjunto no recuerda el orden en el que estaba la lista que sirvió para crearlo. Si añadieras algún elemento nuevo no recordaría el orden en el que lo añadiste.

3. *Línea 5:* La lista original no se ha modificado.

¿Tienes un conjunto vacío? Sin problemas. Puedes crearlo y más tarde añadir elementos.

```
1  >>> un_conjunto = set()
2  >>> un_conjunto
3  set()
4  >>> type(un_conjunto)
5  <class 'set'>
6  >>> len(un_conjunto)
7  0
8  >>> no_seguro = {}
9  >>> type(no_seguro)
10 <class 'dict'>
```

1. *Línea 1:* Para crear un conjunto vacío debes utilizar la función set() sin parámetros.

2. *Línea 2:* La representación impresa de un conjunto vacío parece algo extraña. ¿Tal vez estabas esperando {}? Esa expresión se utiliza para representar un diccionario vacío, no un conjunto vacío. Aprenderás a usar los diccionarios más adelante en este capítulo.

3. *Línea 4:* A pesar de la extraña representación impresa se trata de un conjunto.

4. *Línea 6:* ...y este conjunto no tiene elementos.

5. *Línea 8:* Debido a razones históricas procedentes de Python 2. No puedes utilizar las llaves para crear un conjunto vacío, puesto que lo que se crea es un diccionario vacío, no un conjunto vacío.

[10] Aquellos que conocen cómo están implementados los conjuntos apuntarán que realmente no se trata de una llamada a una función, sino de la instanciación de una clase. Te *prometo* que en este libro aprenderás la diferencia. Pero por ahora basta con que sepas que set() se comporta como una función que devuelve como resultado un conjunto.

2.6.2. Modificación de conjuntos

Hay dos maneras de añadir valores a un conjunto: el método add() y el método update().

```
1  >>> un_conjunto = {1, 2}
2  >>> un_conjunto.add(4)
3  >>> un_conjunto
4  {1, 2, 4}
5  >>> len(un_conjunto)
6  3
7  >>> un_conjunto.add(1)
8  >>> un_conjunto
9  {1, 2, 4}
10 >>> len(un_conjunto)
11 3
```

1. *Línea 2:* El método add() recibe un parámetro, que puede ser de cualquier tipo, cuyo resultado es añadir el parámetro al conjunto.

2. *Línea 5:* Este conjunto tiene ahora cuatro elementos.

3. *Línea 7:* Los conjuntos son "bolsas" de valores *únicos*. Por eso, si intentas añadir un valor que ya exista en el conjunto no hará nada. Tampoco elevará un error. Simplemente no se hace nada.

4. *Línea 10:* Por eso, el conjunto *aún* tiene tres elementos.

```
1  >>> un_conjunto = {1, 2, 3}
2  >>> un_conjunto
3  {1, 2, 3}
4  >>> un_conjunto.update({2, 4, 6})
5  >>> un_conjunto
6  {1, 2, 3, 4, 6}
7  >>> un_conjunto.update({3, 6, 9}, {1, 2, 3, 5, 8, 13})
8  >>> un_conjunto
9  {1, 2, 3, 4, 5, 6, 8, 9, 13}
10 >>> un_conjunto.update([10, 20, 30])
11 >>> un_conjunto
12 {1, 2, 3, 4, 5, 6, 8, 9, 10, 13, 20, 30}
```

1. *Línea 4:* El método update() toma un parámetro, un conjunto, y añade todos sus elementos al conjunto original. Funciona como si llamaras al método add() con cada uno de los elementos del conjunto que pasas como parámetro.

2. *Línea 5:* Los elementos duplicados se ignoran puesto que los conjuntos no pueden contener duplicados.

3. *Línea 7:* Puedes llamar al método update() con cualquier número de parámetros. Cuando lo llamas con dos conjuntos, el método añade todos los elementos de cada conjunto al conjunto original (sin incluir duplicados).

4. *Línea 10:* El método update() puede recibir como parámetro elementos de diferentes tipos de dato, incluidas las listas. Cuando se llama pasándole una lista, el método update() añade todos los elementos de la lista al conjunto original.

2.6.3. Eliminar elementos de un conjunto

Existen tres formas de eliminar elementos individuales de un conjunto: Las dos primeras discard() y remove(), se diferencian de forma sutil:

```
1  >>> un_conjunto = {1, 3, 6, 10, 15, 21, 28, 36, 45}
2  >>> un_conjunto
3  {1, 3, 36, 6, 10, 45, 15, 21, 28}
4  >>> un_conjunto.discard(10)
5  >>> un_conjunto
6  {1, 3, 36, 6, 45, 15, 21, 28}
7  >>> un_conjunto.discard(10)
8  >>> un_conjunto
9  {1, 3, 36, 6, 45, 15, 21, 28}
10 >>> un_conjunto.remove(21)
11 >>> un_conjunto
12 {1, 3, 36, 6, 45, 15, 28}
13 >>> un_conjunto.remove(21)
14 Traceback (most recent call last):
15   File "<stdin>", line 1, in <module>
16 KeyError: 21
```

1. *Línea 4:* El método discard() toma un único parámetro y elminia el elemento del conjunto.

2. *Línea 7:* Si llamas al método discard() con un valor que no exista en el conjunto no se produce ningún error. Simplemente no se hace nada.

3. *Línea 10:* El método remove() también recibe un único parámetro y también elimina el elemento del conjunto.

4. *Línea 13:* Aquí esta la diferencia: si el valor no existe en el conjunto, el método remove() eleva la excepción KeyError.

Como pasa con las listas, los conjuntos también tienen el método pop():

```
1  >>> un_conjunto = {1, 3, 6, 10, 15, 21, 28, 36, 45}
2  >>> un_conjunto.pop()
3  1
4  >>> un_conjunto.pop()
5  3
6  >>> un_conjunto.pop()
7  36
8  >>> un_conjunto
9  {6, 10, 45, 15, 21, 28}
10 >>> un_conjunto.clear()
11 >>> un_conjunto
12 set()
13 >>> un_conjunto.pop()
14 Traceback (most recent call last):
15   File "<stdin>", line 1, in <module>
16 KeyError: 'pop from an empty set'
```

1. *Línea 2:* El método pop() elimina un único valor del conjunto y retorna el valor. Sin embargo, como los conjuntos no están ordenados, no hay un "último" elemento, por lo que no hay forma de controlar qué elemento es el que se extrae. Es aleatorio.

2. *Línea 10:* El método clear() elimina todos los valores del conjunto dejándolo vacío. Es equivalente a un_conjunto = set(), que crearía un nuevo conjunto vacío y lo asignaría a la variable, eliminando el conjunto anterior.

3. *Línea 13:* Si se intenta extraer un valor de un conjunto vacío se eleva la excepción KeyError.

2.6.4. Operaciones típicas de conjuntos

Los conjuntos de Python permiten las operaciones habituales de este tipo de datos:

```
1  >>> un_conjunto = {2, 4, 5, 9, 12, 21, 30, 51, 76, 127, 195}
2  >>> 30 in un_conjunto
3  True
4  >>> 31 in un_conjunto
5  False
6  >>> otro_conjunto = {1, 2, 3, 5, 6, 8, 9, 12, 15, 17, 18, 21}
7  >>> un_conjunto.union(otro_conjunto)
8  {1, 2, 195, 4, 5, 6, 8, 12, 76, 15, 17, 18, 3, 21, 30, 51, 9, 127}
9  >>> un_conjunto.intersection(otro_conjunto)
10 {9, 2, 12, 5, 21}
11 >>> un_conjunto.difference(otro_conjunto)
12 {195, 4, 76, 51, 30, 127}
13 >>> un_conjunto.symmetric_difference(otro_conjunto)
14 {1, 3, 4, 6, 8, 76, 15, 17, 18, 195, 127, 30, 51}
```

1. *Línea 2:* Para comprobar si un valor está contenido en un conjunto se puede utilizar el operador in. Funciona igual que en las listas.
2. *Línea 7:* El método union() retorna un nuevo conjunto que contiene todos los elementos que están en *alguno* de los conjuntos originales.
3. *Línea 9:* El método intersection() retorna un nuevo conjunto con los elementos que están en *ambos* conjuntos originales.
4. *Línea 11:* El método difference() retorna un nuevo conjunto que contiene los elementos que están en un_conjunto pero no en otro_conjunto.
5. *Línea 13:* El método symmetric_difference() retorna un nuevo conjunto que contiene todos los elementos que están únicamente en uno de los conjuntos originales.

Tres de estos métodos son simétricos:

```
1  # continuación del ejemplo anterior
2  >>> otro_conjunto.symmetric_difference(un_conjunto)
3  {3, 1, 195, 4, 6, 8, 76, 15, 17, 18, 51, 30, 127}
4  >>> otro_conjunto.symmetric_difference(un_conjunto) == \
5  ... un_conjunto.symmetric_difference(otro_conjunto)
6  True
7  >>> otro_conjunto.union(un_conjunto) == \
8  ... un_conjunto.union(otro_conjunto)
9  True
10 >>> otro_conjunto.intersection(un_conjunto) == \
11 ... un_conjunto.intersection(otro_conjunto)
12 True
13 >>> otro_conjunto.difference(un_conjunto) == \
14 ... un_conjunto.difference(otro_conjunto)
15 False
```

1. *Línea 2:* Aunque el resultado de la diferencia simétrica de un_conjunto y otro_conjunto parezca diferente de la diferencia simétrica de otro_conjunto y un_conjunto, recuerda que los conjuntos están desordenados. Dos conjuntos con los mismos valores se consideran iguales.

2. *Línea 4:* Y eso es lo que sucede aquí. No te despistes por la representación impresa de los conjuntos. Como contienen los mismos valores, son iguales.

3. *Línea 7:* La unión de dos conjuntos también es simétrica.

4. *Línea 10:* La intersección de dos conjuntos también es simétrica.

5. *Línea 13:* La diferencia de dos conjuntos no es simétrica, lo que tiene sentido, es análogo a la resta de dos números; importa el orden de los operandos.

Finalmente veamos algunas consultas que se pueden hacer a los conjuntos:

```
1 >>> un_conjunto = {1, 2, 3}
2 >>> otro_conjunto = {1, 2, 3, 4}
3 >>> un_conjunto.issubset(otro_conjunto)
4 True
5 >>> otro_conjunto.issuperset(un_conjunto)
6 True
7 >>> un_conjunto.add(5)
8 >>> un_conjunto.issubset(otro_conjunto)
9 False
10 >>> otro_conjunto.issuperset(un_conjunto)
11 False
```

1. *Línea 3:* un_conjunto es un subconjunto de otro_conjunto —Todos los miembros de un_conjunto forman parte de otro_conjunto.

2. *Línea 5:* Pregunta lo mismo pero al revés. otro_conjunto es un superconjunto de un_conjunto —Todos los miembros de un_conjunto forman parte de otro_conjunto.

3. *Línea 7:* Tan pronto como añadas un valor a un_conjunto que no se encuentre en otro_conjunto ambas consultas devuelven False.

2.6.5. Los conjuntos en contextos booleanos

Puedes utilizar conjuntos en contextos booleanos, como en una sentencia if.

```
1  >>> def es_true(algo):
2  ...     if algo:
3  ...         print("sí, es true")
4  ...     else:
5  ...         print("no, es false")
6  ...
7  >>> es_true(set())
8  no, es false
9  >>> es_true({'a'})
10 sí, es true
11 >>> es_true({False})
12 sí, es true
```

1. *Línea 7:* En un contexto booleano los conjuntos vacíos valen False.
2. *Línea 9:* Cualquier conjunto con al menos un elemento vale True.
3. *Línea 11:* Cualquier conjunto con al menos un elemento vale True. El valor de los elementos es irrelevante.

2.7. Diccionarios

Un diccionario es un conjunto desordenado de parejas clave-valor. Cuando añades una clave a un diccionario, tienes que añadir también un valor para esa clave[11]. Los diccionarios de Python están optimizados para recuperar fácilmente el valor cuando conoces la clave, no al revés[12].

> Un diccionario de Python, es como un hash de Perl 5. En Perl 5 las variables que almacenan "hashes" siempre comienzan por el carácter %. En Python, las variables pueden tener el nombre que se quiera porque el tipo de datos se mantiene internamente.

2.7.1. Creación de diccionarios

Crear diccionarios es sencillo. La sintaxis es similar a la de los conjuntos, pero en lugar de valores, tienes que poner parejas clave-valor. Una vez has creado el diccionario, puedes buscar los valores mediante el uso de su clave.

[11] Más tarde puedes cambiar el valor asignado a la clave si lo deseas.

[12] Nota del Traductor: en otros lenguajes se habla de arrays asociativos o tablas hash para representar este mismo concepto

```
1  >>> un_dic = {'servidor': 'db.diveintopython3.org', 'basedatos': 'mysql'}
2  >>> un_dic
3  {'servidor': 'db.diveintopython3.org', 'basedatos': 'mysql'}
4  >>> un_dic['servidor']
5  'db.diveintopython3.org'
6  >>> un_dic['basedatos']
7  'mysql'
8  >>> un_dic['db.diveintopython3.org']
9  Traceback (most recent call last):
10   File "<stdin>", line 1, in <module>
11 KeyError: 'db.diveintopython3.org'
```

1. *Línea 1:* En el ejemplo creamos primero un diccionario con dos elementos y lo asignamos a la variable un_dic. Cada elemento es una pareja clave-valor. El conjunto completo de elementos se encierra entre llaves.

2. *Línea 4:* "servidor" es una clave, y su valor asociado se obtiene mediante la referencia un_dic["servidor"] cuyo valor es "db.diveintopython3.org".

3. *Línea 6:* "basedatos" es una clave y su valor asociado se obtiene mediante la referencia un_dic["basedatos"] cuyo valor es "mysql".

4. *Línea 8:* Puedes recuperar los valores mediante la clave, pero no puedes recuperar las claves mediante el uso de su valor. Por eso un_dic["servidor"] vale "db.diveintopython3.org" pero un_dic["db.diveintopython3.org"] eleva una excepción de tipo KeyError al no ser una clave del diccionario.

2.7.2. Modificación de un diccionario

```
1  >>> un_dic
2  {'servidor': 'db.diveintopython3.org', 'basedatos': 'mysql'}
3  >>> un_dic['basedatos'] = 'blog'
4  >>> un_dic
5  {'servidor': 'db.diveintopython3.org', 'basedatos': 'blog'}
6  >>> un_dic['usuario'] = 'mark'
7  >>> un_dic
8  {'servidor': 'db.diveintopython3.org', 'usuario': 'mark',
9   'basedatos': 'blog'}
10 >>> un_dic['usuario'] = 'dora'
11 >>> un_dic
12 {'servidor': 'db.diveintopython3.org', 'usuario': 'dora',
13  'basedatos': 'blog'}
14 >>> un_dic['Usuario'] = 'mark'
15 >>> un_dic
16 {'Usuario': 'mark', 'servidor': 'db.diveintopython3.org',
17  'usuario': 'dora', 'basedatos': 'blog'}
```

1. *Línea 3:* No puedes tener claves duplicadas en un diccionario. Al asignar un valor a una clave existente el valor anterior se pierde.

2. *Línea 6:* Puedes añadir nuevas parejas clave-valor en cualquier momento. La sintaxis es idéntica a la que se utiliza para modificar valores.

3. *Línea 8:* El elemento nuevo del diccionario (clave "usuario", valor "mark") aparece en la mitad. Esto es una mera coincidencia, los elementos de un diccionario no están ordenados.

4. *Línea 10:* Al asignar un valor a una clave existente, simplemente se sustituye el valor anterior por el nuevo.

5. *Línea 14:* Esta sentencia ¿cambia el valor de la clave "usuario" para volverle asignar "mark"? ¡No! Si lo observas atentamente verás que la "U" está en mayúsculas. Las claves de los diccionarios distinguen las mayúsculas y minúsculas, por eso esta sentencia crea una nueva pareja clave-valor, no sobreescribe la anterior. Puede parecerte casi lo mismo, pero en lo que a Python respecta, es totalmente diferente.

2.7.3. Diccionarios con valores mixtos

Los diccionarios no se usan únicamente con cadenas de texto. Los valores de un diccionario pueden ser de cualquier tipo, incluidos enteros, booleanos, cualquier objeto o incluso otros diccionarios. Y en un mismo diccionario, no es necesario que todos los valores sean del mismo tipo, puedes mezclarlos según lo necesites. Los tipos de datos que pueden ser claves de un diccionario están más limitados, pero pueden ser cadenas de texto, enteros, y algunos tipos más. También es factible mezclar diferentes tipos de clave en un mismo diccionario.

De hecho, ya hemos visto un diccionario con valores diferentes a cadenas de texto.

```
1 SUFIJOS = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],
2            1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}
```

Vamos a descomponerlo en la consola interactiva de Python.

```
1  >>> SUFIJOS = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],
2  ...         1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}
3  >>> len(SUFIJOS)
4  2
5  >>> 1000 in SUFIJOS
6  True
7  >>> SUFIJOS[1000]
8  ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB']
9  >>> SUFIJOS[1024]
10 ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']
11 >>> SUFIJOS[1000][3]
12 'TB'
```

1. *Línea 3:* Como sucede con las listas y conjuntos, la función len() devuelve el número de claves que tiene un diccionario.

2. *Línea 5:* También como pasa con las listas y conjuntos puedes utilizar el operador in para comprobar si una clave determinada está en el diccionario.

3. *Línea 7:* 1000 es una clave del diccionario SUFIJOS; su valor es una lista de ocho elementos (ocho cadenas de texto, por ser más precisos).

4. *Línea 9:* De igual manera, 1024 es una clave del diccionario SUFIJOS; su valor también es una lista de ocho elmentos.

5. *Línea 11:* Puesto que SUFIJOS[1000] es una lista, puedes utilizar los corchetes para acceder a los elementos individuales. Recuerda que los índices en Python comienzan a contar en cero.

2.7.4. Diccionarios en un contexto booleano

También puedes utilizar un diccionario en un contexto booleano, como en la sentencia if.

> Todo diccionario vacío equivale a False y todos los demás equivalen a True.

```
>>> def es_true(algo):
...     if algo:
...         print("sí, es true")
...     else:
...         print("no, es false")
...
>>> es_true({})
no, es false
>>> es_true({'a' : 1})
sí, es true
```

1. *Línea 7:* En un contexto booleano un diccionario vacío equivale a False.
2. *Línea 9:* Cualquier diccionario con, al menos, una pareja clave-valor equivale a True.

2.8. None

None es una constante especial de Python. Representa al valor nulo. None no es lo mismo que False. None tampoco es 0. None tampoco es la cadena vacía. Cualquier comparación de None con otra cosa diferente de él mismo se evalúa al valor False.

None es el único valor nulo. Tiene su propio tipo de dato (NoneType). Puedes asignar None a cualquier variable, pero no puedes crear nuevos objetos del tipo NoneType. Todas las variables cuyo valor es None son iguales entre sí.

```
>>> type(None)
<class 'NoneType'>
>>> None == False
False
>>> None == 0
False
>>> None == ''
False
>>> None == None
True
>>> x = None
>>> x == None
True
>>> y = None
>>> x == y
True
```

2.8.1. None en un contexto booleano

En un contexto booleano None vale False y not None vale True.

```
1
2  >>> def es_true(algo):
3  ...     if algo:
4  ...       print("sí, es true")
5  ...     else:
6  ...       print("no, es false")
7  ...
8  >>> es_true(None)
9  no, es false
10 >>> es_true({not None)
11 sí, es true
```

2.9. Lecturas complementarias

- Operaciones booleanas
- Tipos numéricos
- Tipos secuencia
- Tipos conjunto
- Tipos mapa
- módulo fractions
- módulo math
- PEP 237: Unificación de enteros largos y enteros
- PEP 238: Modificación del operador de división

Capítulo 3

Compresiones

Nivel de dificultad:◆ ◆ ◇ ◇ ◇

"Nuestra imaginación está desplegada a más no poder, no como en la ficción, para imaginar las cosas que no están realmente ahí, sino para entender aquellas que sí lo están."
—Rychard Feynman

3.1. Inmersión

Este capítulo te explicará las listas por comprensión, diccionarios por comprensión y conjuntos por comprensión: tres conceptos centrados alrededor de una técnica muy potente. Pero antes vamos a dar un pequeño paseo alrededor de dos módulos que te van a servir para navegar por tu sistema de ficheros.

3.2. Trabajar con ficheros y directorios

Python 3 posee un módulo denominado os que es la contracción de "operating system"[1]. El módulo os contiene un gran número de funciones para recuperar —y en algunos casos, modificar— información sobre directorios, ficheros, procesos y variables del entorno local. Python hace un gran esfuerzo por ofrecer una API unificada en todos los sistemas operativos que soporta, por lo que tus programas pueden funcionar en casi cualquier ordenador con el mínimo de código específico posible.

[1] Sistema Operativo.

3.2.1. El directorio de trabajo actual

Cuando te inicias en Python, pasas mucho tiempo en la consola interactiva. A lo largo del libro verás muchos ejemplos que siguen el siguiente patrón:

1. Se importa uno de los módulos de la carpeta de ejemplos.
2. Se llama a una función del módulo.
3. Se explica el resultado.

Si no sabes cuál es el directorio actual de trabajo, el primer paso probablemente elevará la excepción ImportError. ¿Porqué? Porque Python buscará el módulo en el camino de búsqueda actual (ver capítulo **??**), pero no lo encontrará porque la carpeta ejemplos no está incluida en él. Para superar este problema hay dos soluciones posibles:

1. Añadir el directorio de ejemplos al camino de búsqueda de importación.
2. Cambiar el directorio de trabajo actual a la carpeta de ejemplos.

El directorio de trabajo actual es una propiedad invisible que Python mantiene en memoria. Siempre existe un directorio de trabajo actual: en la consola interactiva de Python; durante la ejecución de un programa desde la línea de comando, o durante la ejecución de un programa Python como un CGI de algún servidor web.

El módulo os contiene dos funciones que te permiten gestionar el directorio de trabajo actual.

```
1 >>> import os
2 >>> print(os.getcwd())
3 /home/jmgaguilera
4 >>> os.chdir('/home/jmgaguilera/inmersionenpython3/ejemplos')
5 >>> print(os.getcwd())
6 /home/jmgaguilera/inmersionenpython3/ejemplos
```

1. *Línea 1:* El módulo os viene instalado con Python. Puedes importarlo siempre que lo necesites.
2. *Línea 2:* Utiliza la función os.getcwd() para recuperar el directorio de trabajo actual. Cuando ejecutas la consola interactiva, Python toma como directorio de trabajo actual aquél en el que te encontrases en el sistema operativo antes de entrar en la consola; si ejecutas la consola desde una opción de menú del

sistema operativo, el directorio de trabajo será aquél en el que se encuentre el programa ejecutable de Python o tu directorio de trabajo por defecto[2].

3. *Línea 4:* Utiliza la función os.chdir() para cambiar de directorio. Conviene utilizar la convención de escribir los separadores en el estilo de Linux (con las barras inclinadas adelantadas) puesto que este sistema es universal y funciona también en Windows. Este es uno de los lugares en los que Python intenta ocultar las diferencias entre sistemas operativos.

3.2.2. Trabajar con nombres de ficheros y directorios

Aprovechando que estamos viendo los directorios, quiero presentarte el módulo os.path, que contiene funciones para manipular nombres de ficheros y directorios.

```
1  >>> import os
2  >>> print(os.path.join('/home/jmgaguilera/inmersionenpython3/ejemplos/',
3                         'parahumanos.py'))
4  /home/jmgaguilera/inmersionenpython3/ejemplos/parahumanos.py
5  >>> print(os.path.join('/home/jmgaguilera/inmersionenpython3/ejemplos',
6                         'parahumanos.py'))
7  /home/jmgaguilera/inmersionenpython3/ejemplos/parahumanos.py
8  >>> print(os.path.expanduser('~'))
9  /home/jmgaguilera
10 >>> print(os.path.join(os.path.expanduser('~'),
11                         'inmersionenpython3', 'examples',
12                         'humansize.py'))
13 /home/jmgaguilera/inmersionenpython3/ejemplos\parahumanos.py
```

1. *Línea 2:* La función os.path.join() construye un nombre completo de fichero o directorio (nombre de path) a partir de uno o más partes. En este caso únicamente tiene que concatenar las cadenas.

2. *Línea 5:* Este caso es menos trivial. La función añade una barra inclinada antes de concatenar. Dependiendo de que el ejemplo se construya en Windows o en una versión de Linux o Unix, la barra inclinada será invertida o no. Python será capaz de encontrar el fichero o directorio independientemente del sentido en el que aparezcan las barras inclinadas. En este caso, como el ejemplo lo construí en Linux, la barra inclinada es la típica de Linux.

3. *Línea 8:* La función os.path.expanduser() obtendrá un camino completo al directorio que se exprese y que incluye como indicador el directorio raíz del

[2]Esto depende del sistema operativo: windows, linux, ...

usuario conectado. Esto funcionará en todos los sistemas operativos que tengan el concepto de "directorio raíz del usuario", lo que incluye OS X, Linux, Unix y Windows. El camino que se retorna no lleva la barra inclinada al final, pero, como hemos visto, a la función os.path.join() no le afecta.

4. *Línea 10:* Si combinamos estas técnicas podemos construir fácilmente caminos completos desde el directorio raíz del usuario. La función os.path.join() puede recibir cualquier número de parámetros. Yo me alegré mucho al descubrir esto puesto que la función anyadirBarra() es una de las típicas que siempre tengo que escribir cuando aprendo un lenguaje de programación nuevo. *No escribas* esta estúpida función en Python, personas inteligentes se ha ocupado de ello por ti.

El módulo os.path también contiene funciones para trocear caminos completos, nombres de directorios y nombres de fichero en sus partes constituyentes.

```
1  >>> nombrepath = '/home/jmgaguilera/inmersionenpython3/parahumanos.py'
2  >>> os.path.split(nombrepath)
3  ('/home/jmgaguilera/inmersionenpython3/ejemplos', 'parahumanos.py')
4  >>> (nombredir, nombrefich) = os.path.split(nombrepath)
5  >>> nombredir
6  '/home/jmgaguilera/inmersionenpython3/ejemplos'
7  >>> nombrefich
8  'parahumanos.py'
9  >>> (nombrecorto, extension) = os.path.splitext(nombrefich)
10 >>> nombrecorto
11 'parahumanos'
12 >>> extension
13 '.py'
```

1. *Línea 2:* La función split() divide un camino completo en dos partes que contienen el camino y el nombre de fichero. Los retorna en una tupla.

2. *Línea 4:* ¿Recuerdas cuando dije que podías utilizar la asignación múltiple para devolver varios valores desde una función? os.path.split() hace exactamente eso. Puedes asignar los valores de la tupla que retorna a dos variables. Cada variable recibe el valor que le corresponde al elemento de la tupla.

3. *Línea 5:* La primera variable, nombredir, recibe el valor del primer elemento de la tupla que retorna os.path.split(), el camino al fichero.

4. *Línea 7:* La segunda variable, nombrefich, recibe el valor del segundo elemento de la tupla que retorna os.path.split(), el nombre del fichero.

5. *Línea 9:* os.path también posee la función os.path.splitext() que divide el nombre de un fichero en una tupla que contiene el nombre y la extensión separados en dos elementos. Puedes utilizar la misma técnica que antes para asignarlos a dos variables separadas.

3.2.3. Listar directorios

El módulo glob es otra herramienta incluida en la librería estándar de Python. Proporciona una forma sencilla de acceder al contenido de un directorio desde un programa. Utiliza los caracteres *comodín* que suelen usarse en una consola de línea de comandos.

```
1  >>> os.chdir('/home/jmgaguilera/inmersionenpython3/')
2  >>> import glob
3  >>> glob.glob('ejemplos/*.xml')
4  ['ejemplos\\feed-broken.xml',
5   'ejemplos\\feed-ns0.xml',
6   'ejemplos\\feed.xml']
7  >>> os.chdir('ejemplos/')
8  >>> glob.glob('*test*.py')
9  ['alphameticstest.py',
10  'pluraltest1.py',
11  'pluraltest2.py',
12  'pluraltest3.py',
13  'pluraltest4.py',
14  'pluraltest5.py',
15  'pluraltest6.py',
16  'romantest1.py',
17  'romantest10.py',
18  'romantest2.py',
19  'romantest3.py',
20  'romantest4.py',
21  'romantest5.py',
22  'romantest6.py',
23  'romantest7.py',
24  'romantest8.py',
25  'romantest9.py']
```

1. *Línea 3:* El módulo glob utiliza comodines y devuelve el camino de todos los ficheros y directorios que coindicen con la búsqueda. En este ejemplo se busca un directorio que contenga ficheros terminados en "*.xml", lo que encontrará todos los ficheros xml que se encuentren en el directorio de ejemplos.

2. *Línea 7:* Ahora cambio el directorio de trabajo al subdirectorio ejemplos. La

función os.chdir() puede recibir como parámetro un camino relativo a la posición actual.

3. *Línea 8:* Puedes incluir varios comodines de búsqueda. El ejemplo encuentra todos los ficheros del directorio actual de trabajo que incluyan la palabra test en alguna parte del nombre y que, además, terminen con la cadena .py.

3.2.4. Obtener metadatos de ficheros

Todo sistema de ficheros moderno almacena metadatos sobre cada fichero: fecha de creación, fecha de la última modificación, tamaño, etc. Python proporciona una API unificada para acceder a estos metadatos. No necesitas abrir el fichero, únicamente necesitas su nombre.

```
1  >>> import os
2  >>> print(os.getcwd())
3  /home/jmgaguilera/inmersionenpython3/ejemplos
4  >>> metadata = os.stat('feed.xml')
5  >>> metadata.st_mtime
6  1247520344.9537716
7  >>> import time
8  >>> time.localtime(metadata.st_mtime)
9  time.struct_time(tm_year=2009, tm_mon=7, tm_mday=13, tm_hour=17,
10   tm_min=25, tm_sec=44, tm_wday=0, tm_yday=194, tm_isdst=1)
```

1. *Línea 2:* El directorio de trabajo actual es ejemplos.

2. *Línea 4:* feed.xml es un fichero que se encuentra en el directorio ejemplos. La función os.stat() devuelve un objeto que contiene diversos metadatos sobre el fichero.

3. *Línea 5:* st_mtime contiene la fecha y hora de modificación, pero en un formato que no es muy útil (Técnicamente es el número de segundos desde el inicio de la Época, que está definida como el primer segundo del 1 de enero de 1970 ¡En serio!).

4. *Línea 7:* El módulo time forma parte de la librería estándar de Python. Contiene funciones para convertir entre diferentes representaciones del tiempo, formatear valores de tiempo en cadenas y manipular las referencias a los husos horarios.

5. *Línea 8:* La función time.localtime() convierte un valor de segundos desde el inicio de la época (que procede la propiedad anterior) en una estructura más

útil que contiene año, mes, día, hora, minuto, segundo, etc. Este fichero se modificó por última vez el 13 de julio de 2009 a las 5:25 de la tarde.

```
1 # continuación del ejemplo anterior
2 >>> metadata.st_size
3 3070
4 >>> import parahumanos
5 >>> parahumanos.tamnyo_aproximado(metadata.st_size)
6 '3.0 KiB'
```

1. *Línea 2:* La función os.stat() también devuelve el tamaño de un fichero en la propiedad st_size. El fichero feed.xml ocupa 3070 bytes.

2. *Línea 5:* Aprovecho la función tamanyo_aproximado() para verlo de forma más clara.

3.2.5. Construcción de caminos absolutos

En el apartado anterior, se observó cómo la función glob.glob() devolvía una lista de nombres relativa. El primer ejemplo mostraba caminos como "ejemplos/feed.xml", y el segundo ejemplo incluso tenía nombres más cortos como "romantest1.py". Mientras permanezcas en el mismo directorio de trabajo los path relativos funcionarán sin problemas para recuperar información de los ficheros. No obstante, si quieres construir un camino absoluto —Uno que contenga todos los directorios hasta el raíz del sistema de archivos— lo que necesitas es la función os.path.realpath().

```
1 >>> import os
2 >>> print(os.getcwd())
3 /home/jmgaguilera/inmersionenpython3/ejemplos
4 >>> print(os.path.realpath('feed.xml'))
5 /home/jmgaguilera/inmersionenpython3/ejemplos/feed.xml
```

3.3. Listas por comprensión

La creacion de listas por comprensión proporciona una forma compacta de crear una lista a partir de otra mediante la realización de una operación a cada uno de los elementos de la lista original.

```
1 >>> una_lista = [1, 9, 8, 4]
2 >>> [elem * 2 for elem in una_lista]
3 [2, 18, 16, 8]
4 >>> una_lista
5 [1, 9, 8, 4]
6 >>> una_lista = [elem * 2 for elem in una_lista]
7 >>> una_lista
8 [2, 18, 16, 8]
```

1. *Línea 2:* Para explicar esto es mejor leerlo de derecha a izquierda. una_lista es la lista origen que se va a recorrer para generar la nueva lista. El intérprete de Python recorre cada uno de los elementos de una_lista, asignando temporalmente el valor de cada elemento a la variable elem. Después Python aplica la operación que se haya indicado, en este caso elem * 2, y el resultado lo añade a la nueva lista.

2. *Línea 4:* Como se observa, la lista original no cambia.

3. *Línea 6:* No pasa nada por asignar el resultado a la variable que tenía la lista original. Python primero construye la nueva lista en memoria y luego asigna el resultado a la variable.

Para crear una lista de esta forma, puedes utilizar cualquier expresión válida de Python, como por ejemplo las funciones del módulo os para manipular ficheros y directorios.

```
1 >>> import os, glob
2 >>> glob.glob('*.xml')
3 ['feed-broken.xml', 'feed-ns0.xml', 'feed.xml']
4 >>> [os.path.realpath(f) for f in glob.glob('*.xml')]
5 ['/home/jmgaguilera/inmersionenpython3/ejemplos/feed-broken.xml',
6  '/home/jmgaguilera/inmersionenpython3/ejemplos/feed-ns0.xml',
7  '/home/jmgaguilera/inmersionenpython3/ejemplos/feed.xml']
```

1. *Línea 2:* Esta llamada retorna una lista con todos los ficheros terminados en .xml del directorio de trabajo.

2. *Línea 4:* Esta lista generada por comprensión toma la lista original y la transforma en una nueva lista con los nombres completos de ruta.

Las listas por comprensión también permiten filtrar elementos, generando una lista cuyo tamaño sea menor que el original.

```
1 >>> import os, glob
2 >>> [f for f in glob.glob('*.py') if os.stat(f).st_size > 6000]
3 ['pluraltest6.py',
4  'romantest10.py',
5  'romantest6.py',
6  'romantest7.py',
7  'romantest8.py',
8  'romantest9.py']
```

1. *Línea 2:* Para filtrar una lista puedes incluir la claúsula if al final de la comprensión. Esta expresión se evalúa para cada elemento de la lista original. Si el resultado es verdadero, el elemento será calculado e incluido en el resultado. En este caso se seleccionan todos los ficheros que terminan en .py que se encuentren en el directorio de trabajo, se comprueba si son de tamaño mayor a 6000 bytes. Seis de ellos cumplen este requisito, por lo que son los que aparecen en el resultado final.

Hasta el momento, todos los ejemplos de generación de listas por comprensión han utilizado expresiones muy sencillas —multiplicar un número por una constante, llamada a una función o simplemente devolver el elemento original de la lista— pero no existe límite en cuanto a la complejidad de la expresión.

```
1  >>> import os, glob
2  >>> [(os.stat(f).st_size, os.path.realpath(f)) for f in glob.glob('*.xml')]
3  [(3074, 'c:/home/jmgaguilera/inmersionenpython3/ejemplos/feed-broken.xml'),
4   (3386, 'c:/home/jmgaguilera/inmersionenpython3/ejemplos/feed-ns0.xml'),
5   (3070, 'c:/home/jmgaguilera/inmersionenpython3/ejemplos/feed.xml')]
6  >>> import parahumanos
7  >>> [(parahumanos.tamanyo_aproximado(os.stat(f).st_size), f)
8       for f in glob.glob('*.xml')]
9  [('3.0 KiB', 'feed-broken.xml'),
10  ('3.3 KiB', 'feed-ns0.xml'),
11  ('3.0 KiB', 'feed.xml')]
```

1. *Línea 2:* En este caso se buscan los ficheros que finalizan en .xml en el directorio de trabajo actual, se recupera su tamaño (mediante una llamada a la función os.stat()) y se construye una tupla con el tamaño del fichero y su ruta completa (mediante una llamada a os.path.realpath().

2. *Línea 7:* En este caso se aprovecha la lista anterior para generar una nueva con el tamaño aproximado de cada fichero.

3.4. Diccionarios por comprensión

Es similar al apartado anterior pero genera un diccionario en lugar de una lista.

```
1  >>> import os, glob
2  >>> metadata = [(f, os.stat(f)) for f in glob.glob('*test*.py')]
3  >>> metadata[0]
4  ('alphameticstest.py', nt.stat_result(st_mode=33206, st_ino=0, st_dev=0,
5    st_nlink=0, st_uid=0, st_gid=0, st_size=2509, st_atime=1247520344,
6    st_mtime=1247520344, st_ctime=1247520344))
7  >>> metadata_dict = {f:os.stat(f) for f in glob.glob('*test*.py')}
8  >>> type(metadata_dict)
9  <class 'dict'>
10 >>> list(metadata_dict.keys())
11 ['romantest8.py', 'pluraltest1.py', 'pluraltest2.py', 'pluraltest5.py',
12   'pluraltest6.py', 'romantest7.py', 'romantest10.py', 'romantest4.py',
13   'romantest9.py', 'pluraltest3.py', 'romantest1.py', 'romantest2.py',
14   'romantest3.py', 'romantest5.py', 'romantest6.py', 'alphameticstest.py',
15   'pluraltest4.py']
16 >>> metadata_dict['alphameticstest.py'].st_size
17 2509
```

1. *Línea 2:* Esto no genera un diccionario por comprensión, genera una lista por comprensión. Encuentra todos los ficheros terminados en .py con el texto test en el nombre y luego construye una tupla con el nombre y los metadatos del fichero (llamando a la función os.stat()).

2. *Línea 3:* Cada elemento de la lista resultante es una tupla.

3. *Línea 7:* Esto sí es una generación de un diccionario por comprensión. La sintaxis es similar a la de la generación de listas, con dos diferencias: primero, se encierra entre llaves en lugar de corchetes; segundo, en lugar de una única expresión para cada elemento, contiene dos expresiones separadas por dos puntos. La expresión que va delante de los dos puntos es la clave del diccionario y la expresión que va detrás es el valor (os.stat(f) en este ejemplo).

4. *Línea 8:* El resultado es un diccionario.

5. *Línea 10:* La claves de este caso particular son los nombres de los ficheros.

6. *Línea 16:* El valor asociado a cada clave es el valor que retornó la función os.stat(). Esto significa que podemos utilizar este diccionario para buscar los metadatos de un fichero a partir de su nombre. Uno de los elementos de estos metadatos es st_size, el tamaño de fichero. Para el fichero alphameticstest.py el valor es 2509 bytes.

Como con las listas, puedes incluir la claúsula if para filtrar los elementos de entrada mediante una expresión que se evalúa para cada uno de los elementos.

```
1 >>> import os, glob, parahumanos
2 >>> dict = {os.path.splitext(f)[0]:parahumanos.tamanyo_aproximado(
3                os.stat(f).st_size) \
4 ...            for f in glob.glob('*') if os.stat(f).st_size > 6000}
5 >>> list(dict.keys())
6 ['romantest9', 'romantest8', 'romantest7', 'romantest6',
7  'romantest10', 'pluraltest6']
8 >>> dict['romantest9']
9 '6.5 KiB'
```

1. *Línea 4:* Este ejemplo construye una lista con todos los ficheros del directorio de trabajo actual (glob.glob('*')), filtra la lista para incluir únicamente aquellos ficheros mayores de 6000 bytes (if os.stat(f).s_size > 6000) y utiliza la lista filtrada para construir un diccionario cuyas claves son los nombres de fichero menos la extensión (os.path.splitext(f)[0]) y los valores el tamaño de cada uno de ellos (parahumanos.tamanyo_aproximado(os.stat(f).st_size)).

2. *Línea 5:* Como viste en el ejemplo anterior son seis ficheros, por lo que hay seis elementos en el diccionario.

3. *Línea 7:* El valor de cada elemento es la cadena que retorna la función tamanyo_aproximado().

3.4.1. Trucos que se pueden hacer

Te presento un truco que puede serte de utilidad: intercambiar las claves y valores de un diccionario.

```
1 >>> dict = {'a': 1, 'b': 2, 'c': 3}
2 >>> {value:key for key, value in a_dict.items()}
3 {1: 'a', 2: 'b', 3: 'c'}
```

3.5. Conjuntos por comprensión

Por último mostraré la sintaxis para generar conjuntos por comprensión. Es muy similiar a la de los diccionarios, con la única diferencia de que únicamente se incluyen valores en lugar de parejas clave-valor.

```
1 >>> conjunto = set(range(10))
2 >>> conjunto
3 {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
4 >>> {x ** 2 for x in conjunto}
5 {0, 1, 4, 81, 64, 9, 16, 49, 25, 36}
6 >>> {x for x in conjunto if x %2 == 0}
7 {0, 8, 2, 4, 6}
8 >>> {2**x for x in range(10)}
9 {32, 1, 2, 4, 8, 64, 128, 256, 16, 512}
```

1. *Línea 4:* Los conjuntos generados por comprensión pueden partir de otro conjunto en lugar de una lista. En este ejemlo se calcula el cuadrado de cada uno de los elementos (los números del 0 al 9).
2. *Línea 6:* Como en el caso de las listas y diccionarios, puedes incluir una claúsula if para filtrar elementos antes de calcularlos e incluirlos en el resultado.
3. *Línea 8:* Los conjuntos por comprensión no necesitan tomar un conjunto como entrada, pueden partir de cualquier tipo de secuencia.

3.6. Lecturas complementarias

- módulo os
- os — Portabilidad en el acceso a características específicas del sistema operativo
- módulo os.path
- Manipulación de los nombres de fichero independiente de la plataforma — os.path
- módulo glob
- Patrones de búsqueda de ficheros — glob
- módulo time
- Funciones para manipulación de hora – time
- Listas por comprensión
- Comprensiones anidadas
- Técnicas para hacer bucles

Capítulo 4

Cadenas de texto

Nivel de dificultad:◆ ◆ ◆ ◇ ◇

"Te digo esto porque eres uno de mis amigos,
¡Mi vocabulario comienza donde el tuyo termina!"
—Dr. Seuss, ¡On beyond Zebra!

4.1. Temas aburridos que debes conocer antes de la inmersión

¿Sabías que la gente de Bougainville tiene el alfabeto más pequeño del mundo? El alfabeto Rotokas está compuesto únicamente por 12 letras: A, E, G, I, K, O, P, R, S, T, U y V. En el otro lado del espectro los lenguajes como el Chino, Japonés y Koreano tienen miles de caracteres. El inglés, desde luego, tiene 26 letras —52 si cuentas las mayúsculas y minúsculas de forma separada— más un puñado de símbolos de puntuación *!@#$ %&?*.

Cuando las personas hablan sobre "texto" piensan en "caracteres y símbolos en la pantalla del ordenador". Pero los ordenadores no conocen ni símbolos ni caracteres, conocen bits y bytes. Cada elemento textual que ves en la pantalla está almacenado con una codificación de caracteres particular. Explicándolo de manera informal, la codificación de caracteres proporciona una conversión entre lo que ves en la pantalla y lo que el ordenador realmente almacena en memoria o en disco. Existen muchas codificaciones de caracteres diferentes, algunas optimizadas para determinados lenguajes como el ruso, el chino o el inglés, y otras que se pueden utilizar para diferentes lenguajes.

En realidad es más complicado. Muchos caracteres son comunes a diferentes codificaciones, pero cada codificación puede utilizar una secuencia de bytes diferente para almacenar esos caracteres en memoria o disco. Puedes imaginarte que una codificación de caracteres es como una especia de clave de desencriptado. Cuando tengas una secuencia de bytes —un fichero, una página web o cualquier otra cosa— y se considere que esos bytes representan "texto", necesitas conocer en qué codificación de caracteres se encuentra para poder decodificar los bytes y conocer a qué caracteres representan. Si tienes una clave de decodificación equivocada o no dispones de ninguna, la decodificación no será posible o será errónea (si se usa una decodificación equivocada), y el resultado será un texto sin sentido.

Todo lo que que pensabas que sabías sobre las cadenas de texto es erróneo.

Seguramente habrás visto a veces paginas web con extraños caracteres de interrogación o similar, en donde esperabas algún caracter como el apóstrofo o vocales acentuadas. Esto suele indicar que el autor de la página no declaró correctamente la codificación de caracteres que utilizó por lo que tu navegador la tiene que adivinar y el resultado es una mezcla de caracteres esperados e inesperados. En inglés esto es simplemente desconcertante, pero en otros lenguajes el resultado puede ser completamente ilegible.

Existen tablas de codificación de caracteres para cada uno de los lenguajes importantes del mundo. Puesto que cada lenguaje es diferente, y la memoria y el espacio en disco ha sido caro históricamente, cada tabla de codificación de caracteres está optimizada para un lenguaje en particular. Lo que quiero decir con esto es que cada una de las codificaciones usa los mismos números (0 - 255) para representar los caracteres de un lenguaje determinado. Por ejemplo, posiblemente estés familiarizado con la codificación ASCII, que almacena los caracteres del inglés como números que van del 0 al 127 (65 es la "A", 97 es la "a", etc). El inglés es un alfabeto muy simple, por lo que puede expresarse con menos de 128 números. Para aquellos que sepan contar en base 2, eso significa 7 bits de los 8 de un byte.

Algunos lenguajes de Europa como el francés, español y alemán necesitan más letras que el inglés. O, para ser más precisos, tienen letras que se combinan con diversas marcas diacríticas, como el carácter ñ del español. La tabla de codificación de caracteres más común para estos lenguajes es CP-1252, que también es conocida como windows-1252 porque se utiliza ampliamente en el sistema operativo Microsoft Windows. La codificación CP-1252 comparte con ASCII los primeros 128 caracteres (0-127), pero luego se extiende en el rango de 128 a 255 para los caracteres restantes (241 es la "ñ", 252 es la "ü", etc). Continúa siendo una tabla de codificación de un único byte. El valor mayor, 255, aún *cabe* en un byte.

Además existen lenguajes como el chino, japonés y coreano, que tienen tantos

caracteres que requieren tablas de codificación de caracteres multibyte. Esto significa que cada "carácter" se representa como un número de dos bytes lo que abarca del 0 al 65535. Pero las codificaciones multibyte también tienen el mismo problema que las diferentes codificaciones de un único byte: cada una de ellas puede utilizar el mismo número para expresar un carácter diferente. La única diferencia entre ellas es que el rango de caracteres disponible es mayor en las codificaciones multibyte.

Esto no suponía demasiado problema en un mundo desconectado, en donde "texto" era algo que tecleabas para tí y ocasionalmente imprimías. No existía mucho "texto plano". El código fuente era ASCII y todo el mundo usaba procesadores de textos que definían su propo formato que tenían en cuenta la información de codificación de caracteres junto con la información de estilo, etc. La gente leía estos documentos con el mismo programa procesador de texto que el autor original, por lo que todo funcionaba, más o menos.

Ahora piensa en la aparición de las redes globales y en el correo y la web. Mucho "texto plano" anda suelto por el mundo, se crea en un ordenador, se transmite a un segundo y se muestra en un tercero. Los ordenadores únicamente distinguen números, pero los números pueden significar cosas diferentes. ¡Oh no! ¿Qué hacer? Bien, los sistemas tuvieron que diseñarse para transportar la información de codificación junto con el "texto plano". Recuerda, se trata de las claves de descodificación que mapean los números entendidos por el ordenador a caracteres legibles por personas. Una clave de descodificación perdida da lugar a texto ilegible.

Ahora piensa en intentar almacenar diversos documentos de texto en el mismo lugar, como en una tabla de una misma base de datos que almacena todo el correo electrónico que hayas recibido. Aún necesitas almacenar la codificación de caracteres junto con cada correo electrónico para que se pueda leer apropiadamente. ¿Parece difícil? Prueba a buscar en tu base de datos de correos, eso significa convertir entre múltiples tablas de codificación de caracteres sobre la marcha. ¿No suena divertido?.

Piensa ahora en la posibilidad de documentos multilíngues, en donde aparecen caracteres en diferentes lenguajes[1]. Y, por supuesto, querrás buscar el contenido de *esos* documentos.

Ahora llora un rato, porque todo lo que creías conocer sobre las cadenas de texto es erróneo, y no existe algo así como el "texto plano".

[1]Pista: los programas que han intentando hacer esto utilizan habitualmente códigos de escape para conmutar entre "modos". Si estás en modo ruso koi8-r el código 241 significa Я. Si cambias a modo Griego el código 241 significa ω.

4.2. Unicode

Entra en Unicode.

Unicode es un sistema diseñado para representar *todos* los caracteres de *todos* los lenguajes. Representa cada letra, carácter o ideograma como un número de cuatro bytes. Cada número representa a un único carácter que se use en al menos uno de los lenguajes del mundo (No se usan todos los números, pero se usan más de 65535 por lo que no es suficiente utilizar dos bytes). Los caracteres que se utilizan en diferentes lenguajes tienen el mismo número generalmente, a menos que exista una buena razón etimológica para que no sea así. De todos modos hay exactamente un número por cada carácter y un carácter por número. De esta forma, cada número siempre significa una única cosa. No existen "modos" que rastrear, U+0041 siempre corresponde a "A", incluso si tu lenguaje no usa tal símbolo.

A primera vista parece una gran idea, una tabla de codificación de caracteres para gobernarlos a todos. Múltiples lenguajes por documento, no más "cambios de modo" para conmutar entre tablas de codificación en medio de un documento. Pero existe una pregunta obvia. ¿Cuatro bytes? ¿Para cada carácter? Parece un gasto inútil la mayor parte de las ocasiones, especialmente para idiomas como el inglés o el español, que necesitan menos de un byte (256 números) para expresar cada uno de los caracteres posibles. De hecho, es un desperdicio de espacio incluso para los lenguajes basados en ideogramas como el chino, que nunca necesitan más de dos caracteres para cada carácter.

Existe una tabla de codificación Unicode que utiliza cuatro bytes por cada carácter. Se denomina UTF-32 porque 32 bits es igual a 4 bytes. UTF-32 es una codificación directa; toma cada carácter Unicode (un número de 4 bytes) y representa al carácter con ese mismo número. Esto tiene varias ventajas siendo la más importante que puedes encontrar el "enésimo" carácter de una cadena en un tiempo constante ya que se encuentra en a partir del byte $4 * n$. También tiene varios inconvenientes, siendo el más obvio que necesita cuatro bytes para almacenar cada carácter.

Incluso aunque existen muchos caracteres Unicode, resulta que la mayoría de la gente nunca usará nada más allá de los primeros 65535. Por eso existe otra codificación Unicode denominada UTF-16 (16 bits son 2 bytes) que codifica cada uno de los caracteres de 0 a 65535 como dos bytes. Además utiliza algunos "trucos sucios" por si necesitas representar aquellos caracteres que se usan raramente y que están más allá del 65535. La ventaja más obvia es que esta tabla de codificación de caracteres es el doble de eficiente que la de UTF-32 puesto que cada carácter requiere únicamente dos bytes para almacenarse, en lugar de cuatro bytes. Aún se puede encontrar fácilmente el enésimo carácter de una cadena en un tiempo constante, siempre que

se asuma que no existen caracteres especiales de los que están por encima de 65535. Lo que suele ser una buena asunción... ¡hasta el momento en que no lo es!

También existen algunos inconvenientes no tan obvios tanto en UFT-32 y UTF-8. Los ordenadores de sistemas diferentes suelen almacenar los bytes de diferentes formas. Esto significa que el carácter U+4E2D podría almacenarse en UTF-16 bien como 4E 2D o como 2D 4E, dependiendo de que el sistema sea "big-endian" o "little-endian"[2] (para UTF-32 existen más posibilidades de ordenación de los bytes). Mientras tus documentos no dejen tu ordenador estás seguro —las diferentes aplicaciones del mismo ordenador utilizarán la misma ordenación de bytes. Pero en el momento que transfieras los documentos entre sistemas, tal vez a través de Internet, vas a necesitar una forma de indicar el orden en el que se han almacenado los bytes. De otra forma el sistema que recibe los datos no tiene forma de saber si la secuencia de dos bytes 4E 2D significa U+4E2D o U+2D4E.

Para resolver *este* problema, las codificaciones multibyte de Unicode definen el "Byte Orden Mark" (BOM)[3], que es un carácter especial no imprimible que se puede incluir al comienzo de tu documento para indica qué ordenación de bytes tiene el documento. Para UTF-16 la marca de ordenación de bytes es U+FEFF, por lo que si recibes un documento UTF-16 que comienza con los bytes FF FE, ya sabes en qué forma vienen ordenados los bytes; si comienza con FE FF sabes que el orden es el contrario.

Aún así, UTF-16 no es exactamente el ideal, especialmente si la mayor parte de los caracteres que utilizas son ASCII. Si lo piensas, incluso una página web china contendrá muchos caracteres ASCII —todos los elementos y atributos que rodean a los caracteres imprimibles chinos. Poder encontrar el "enésimo" carácter está bien, pero existe el problema de los caracteres que requieren más de dos bytes, lo que significa que no puedes *garantizar* que todos los caracteres ocupan exactamente dos bytes, por lo que en la práctica no puedes encontrar el carácter de la posición "enésima" en un tiempo constante a no ser que mantengas un índice separado. Y muchacho, te aseguro que hay mucho texto ASCII por el mundo...

Otras personas valoraron estas preguntas y plantearon una solución:

UTF-8

UTF-8 es un sistema de codificación de *longitud variable* para Unicode. Esto significa que los caracteres pueden utilizar diferente número de bytes. Para los caracteres ASCII utiliza un único byte por carácter. De hecho, utiliza exactamente

[2] Almacene los bytes en orden o invirtiendo el mismo.

[3] Nota del traductor: Marca de ordenación de bytes.

los mismos bytes que ASCII por lo que los 128 primeros caracteres son indistinguibles. Los caracteres "latinos extendidos" como la ñ o la ö utilizan dos bytes[4]. Los caracteres chinos utilizan tres bytes, y los caracteres más "raros" utilizan cuatro.

Desventajas: debido a que los caracteres pueden ocupar un número diferente de bytes, encontrar el carácter de la posición "enésima" es una operación de orden de complejidad O(n) —lo que significa que cuanto más larga sea la cadena, más tiempo lleva encontrar un carácter específico. Asimismo, hay que andar codificando y decodificando entre bytes y caracteres.

Ventajas: se trata de una codificación supereficiente de los caracteres ASCII. No es peor que UTF-16 para los caracteres latinos extendidos, y es mejor que UTF-32 para los caracteres chinos. También (aquí tendrás que confiar en mi, porque no te voy a mostrar la matemática involucrada), debido a la naturaleza exacta de la manipulación de los bits, no existen problemas de ordenación de bits. Un documento codificado en UTF-8 siempre contiene la misma cadena de bytes sea cual sea el ordenador y sistema operativo.

4.3. Inmersión

En Python 3 todas las cadenas de texto son secuencias de caracteres Unicode. En Python 3 no existen cadenas codificadas en UTF-8 o en CP-1252. No es correcto decir ¿Es esta cadena una cadena codificada en UTF-8?. UTF-8 es una forma de codificar caracteres en una secuencia de bytes. Si quieres convertir una cadena de caracteres en una secuencia de bytes en una codificación de caracteres particular, Python 3 puede ayudarte con ello. Si quieres tomar una secuencia de bytes y convertirla en una cadena de texto, también te puede ayudar Python 3. Los bytes no son caracteres, los bytes son bytes. Los caracteres son una abstracción. Una cadena es una secuencia de esas abstracciones.

```
1 >>> s = '快乐 Python'
2 >>> len(s)
3 9
4 >>> s[0]
5 '快'
6 >>> s + ' 3'
7 '快乐 Python 3'
```

1. *Línea 1:* Para crear una cadena de texto debes utilizar las comillas para delimitarla. Se pueden utilizar comillas simples (') o dobles (").

[4]Los bytes no son únicamente la codificación de Unicode como sucede en UTF-16, se efectúan diversos cambios para obtener la codificación en UTF-8.

2. *Línea 2:* La función interna len() devuelve la longitud de la cadena, el número de caracteres. Esta función es la misma que utilizas para conocer la longitud de una lista, tupla, conjunto o diccionario. Una cadena es como una tupla de caracteres.

3. *Línea 4:* De la misma forma que puedes obtener elementos individuales de una lista, puedes obtener caracteres individuales de una cadena mediante la notación de índices.

4. *Línea 6:* Como con las listas y tuplas, puedes contatenar las cadenas utilizando el operador +.

4.4. Formatear cadenas

> Las cadenas de texto se pueden definir utilizando comillas simples o dobles.

Vamos a echarle otro vistazo a parahumanos.py:

```
1  # parahumanos.py
2
3  SUFIJOS = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],
4             1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB',
5                    'YiB']}
6
7  def tamanyo_aproximado(tamanyo, un_kilobyte_es_1024_bytes=True):
8      '''Convierte un tamaño de fichero en formato legible por personas
9
10     Argumentos/parámetros:
11     tamanyo -- tamaño de fichero en bytes
12     un_kilobyte_es_1024_bytes -- si True (por defecto),
13                                  usa múltiplos de 1024
14                                  si False, usa múltiplos de 1000
15
16     retorna: string
17
18     '''
19     if tamanyo < 0:
20         raise ValueError('el número debe ser no negativo')
21
22     multiplo = 1024 if un_kilobyte_es_1024_bytes else 1000
23     for sufijo in SUFIJOS[multiplo]:
24         tamanyo /= multiplo
25         if tamanyo < multiplo:
26             return '{0:.1f} {1}'.format(tamanyo, sufijo)
27
28     raise ValueError('número demasiado grande')
29
30 if __name__ == '__main__':
31     print(tamanyo_aproximado(1000000000000, False))
32     print(tamanyo_aproximado(1000000000000))
```

1. *Línea 3:* 'KB', 'MB', 'GB', ... son cadenas.

2. *Línea 8:* Las cadenas de documentación (docstrings) son cadenas de texto. Como se expanden más allá de una línea se utilizan tres comillas al comienzo y al final para delimitarlas.

3. *Línea 18:* Estas comillas triples finalizan la cadena de documentación de esta función.

4. *Línea 20:* Otra cadena que se pasa como parámetro a la excepción con el fin de que sea legible por una persona.

5. *Línea 26:* Esto es... ¡Uff! ¿Qué car.. es esto?

Python 3 permite formatear valores en cadenas de texto. Aunque este sistema permite expresiones muy complejas, su uso más básico consiste en insertar un valor en una cadena de texto en el lugar definido por un "marcador".

```
1 >>> usuario = 'mark'
2 >>> clave = 'PapayaWhip'
3 >>> "La clave de {0} es {1}".format(usuario, clave)
4 "La clave de mark es PapayaWhip"
```

1. *Línea 2:* Realmente mi clave no es PapayaWhip.

2. *Línea 3:* Aquí hay mucho que contar. Primero, se observa una llamada a un método sobre una cadena de texto. *Las cadenas de texto son objetos*, y los objetos tienen métodos, como ya sabes. Segundo, la expresión completa se evalúa a una cadena. Tercero, {0} y {1} son *campos de reemplazo*, que se sustituyen con los parámetros que se pasen al método format().

4.4.1. Nombres de campos compuestos

En el ejemplo anterior se muestra el ejemplo más simple, aquél en el que los campos de reemplazo son números enteros. Los campos de reemplazo enteros se tratan como índices posicionales en la lista de parámetros del método format(). Eso significa que el {0} se reemplaza por el primer parámetro (usuario en este caso), {1} se reemplaza por el segundo (clave en este caso), etc. Puedes utilizar tantos campos de reemplazo posicional como parámetros se pasen en el método format(). Pero los campos de reemplazo permiten mucha más funcionalidad.

```
1 >>> import parahumanos
2 >>> mis_sufijos = parahumanos.SUFIJOS[1000]
3 >>> mis_sufijos
4 ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB']
5 >>> '1000{0[0]} = 1{0[1]}'.format(mis_sufijos)
6 '1000KB = 1MB'
```

1. *Línea 2:* En lugar de ejecutar una función del módulo parahumanos únicamente estás capturando una de las estructuras de datos que define; la lista de sufijos que representan las potencias de 1000.

2. *Línea 5:* Esta línea parece complicada, pero no lo es. {0} representa el primer párametro del método format(), mis_sufijos. Por eso {0[0]} se refiere al primer elemento de la lista que esté definida como el primer parámetro del método format(): 'KB'. Mientras que {0[1]} se refiere al segundo elemento de la misma

lista: 'MB'. Todo lo que está fuera de las llaves —incluido el 1000, los signos de igual, y los espacios— quedan sin tocar. El resultado final es: '1000KB = 1MB'.

> El {0} se reemplaza por el primer parámetro, {1} se reemplaza por el segundo.

Lo que este ejemplo te enseña es que los *especificadores de formato pueden utilizarse para acceder a los elementos y propiedades de las estructuras de datos utilizando (casi) sintaxis de Python.* Esto se denomina *nombres de campos compuestos.* Estos son los nombres de campos que funcionan:

- Pasar una lista y acceder a un elemento de la lista utilizando un índice (como en el ejemplo anterior).
- Pasar un diccionario y acceder a los valores del mismo utilizando una clave.
- Pasar un módulo y acceder a sus variables y funciones por nombre.
- Pasar una instancia de clase y acceder a sus propiedades y métodos por nombre.
- *Cualquier combinación de las anteriores.*

Solamente para despejar tu cerebro te muestro aquí un ejemplo que combina todo lo anterior.

```
>>> import parahumanos
>>> import sys
>>> '1MB = 1000{0.modules[parahumanos].SUFIJOS[1000][0]}'.format(sys)
'1MB = 1000KB'
```

Así es como funciona:

- El módulo sys almacena información sobre la instancia del lenguaje Python que se está ejecutando en ese momento. Puesto que lo has importado, puedes pasar el propio módulo sys como parámetro del método format(). Así que el campo de sustitución {0} se refiere al módulo sys.
- sys.modules es un diccionario que almacena todos los módulos que se han importando en la instancia de Python que se está ejecutando. Las claves son los nombres de los módulos en formato cadena de texto; los valores son propios módulos (objetos módulo). Por ello {0.modules} se refiere al diccionario que contiene todos módulos que se han importado en Python hasta este momento.

- sys.modules['parahumanos'] retorna el objeto parahumanos que acabas de importar. El campo de reemplazo sys.modules[parahumanos] se refiere al módulo parahumanos. Observa que existe una ligera diferencia de sintaxis. En el código de Python, las claves del diccionario es de tipo cadena; para referirse a ellas, es necesario poner comillas alrededor del nombre del módulo ('parahumanos'). Pero dentro de los campos de sustitución no se ponen las comillas alrededor del nombre de la clave del diccionario (parahumanos). Según el PEP 3101: Formateo avanzado de cadenas: "Las reglas para el parseo de la clave de un campo de sustitución son muy simples, si comienza por un dígito, se trata como numérica, en caso contrario se interpreta como una cadena".
- sys.modules['parahumanos'].SUFIJOS es el diccionario definido en el módulo parahumanos. El campo de sustitución sys.modules[parahumanos].SUFIJOS se refiere a este diccionario.
- sys.modules['parahumanos'].SUFIJOS[1000] es la lista de sufijos múltiplos de 1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'] Por lo que el campo de sustitución sys.modules[parahumanos].SUFIJOS[1000] se refiere a esta lista.
- sys.modules['parahumanos'].SUFIJOS[1000][0] es el primer elemento de la lista de sufijos: ['KB'].
- Por lo tanto el campo de sustitución sys.modules[parahumanos].SUFIJOS[1000][0] se refiere a la cadena 'KB'.

4.4.2. Especificaciones de formato

¡Pero espera! ¡Hay mas! Vamos a echar otro vistazo a la línea de código más extraña de parahumanos.py:

```
1 if tamanyo < multiplo:
2     return '{0:.1f} {1}'.format(tamanyo, sufijo)
```

Como ya sabemos, {1} se sustituye por el segundo parámetro sufijo del método format(). Pero ¿qué es {0:.1f}? Tiene dos partes, {0} que ya conoces, y :.f que no conoces. La segunda parte (desde los dos puntos hasta la letra f) define el especificador de forma que afina cómo debe sustituirse la variable al formatearla.

> Los especificadores de formato te permiten indicar cómo se debe efectuar la sustitución del texto, como sucede con la función printf() en el lenguaje C. Puedes añadir ceros o espacios de relleno delante del número, alinear cadenas, controlar la precisión de decimales o convertir el número a hexadecimal.

Dentro del campo de sustitución se utiliza el símbolo de dos puntos (:) para marcar el comienzo del especificador de formato. El especificador de formato .1 significa que se "redondee a la décima más próxima" (que se muestre únicamente un dígito después del punto decimal). El especificador "f" indica que el número debe mostrarse en formato punto fijo (por oposición a la notación exponencial u otra representación de un número). Si la variable tamanyo vale 698.24 y la variable sufijo vale "GB" la cadena formateada resultante es "698.2 GB", porque 698.24 se redondea con un solo dígito después del punto decimal.

```
>>> '{0:.1f} {1}'.format(698.24, 'GB')
'698.2 GB'
```

Para conocer los detalles exactos de los especificadores de formato puedes consultar el apartado Mini-lenguaje de especificación de formato[5] de la documentación oficial de Python 3.

4.5. Otros métodos habituales de manipulación de cadenas

Además de formatearlas, es posible hacer muchas otras cosas de utilidad con las cadenas de texto.

```
>>> s = '''Los archivos terminados son el re-
... sultado de años de estudio científi-
... fico combinados con la
... experiencia de años.'''
>>> s.splitlines()
['Los archivos terminados son el re-',
 'sultado de años de estudio científi-',
 'fico combinados con la ',
 'experiencia de años.']
>>> print(s.lower())
los archivos terminados son el re-
sultado de años de estudio científi-
fico combinados con la
experiencia de años.
>>> s.lower().count('l')
4
```

1. *Línea 1:* Puedes introducir cadenas multilínea en la consola interactiva de Python. Cuando inicias una cadena de texto multilínea debes pulsar la tecla

[5] http://docs.python.org/3.1/library/string.html#format-specification-mini-language

INTRO para continuar en la siguiente línea. Al teclear las triples comillas del final, se cierra la cadena de texto y el siguiente INTRO que pulses ejecutará la sentencia (en este caso asignará la cadena a la variable s).

2. *Línea 5:* El método splitlines() toma una cadena multilínea y devuelve una lista de cadenas de texto, una por cada línea que contuviese la cadena original. Observa que las líneas no incluyen los retornos de carro o finales de línea que tuviese la cadena original.

3. *Línea 10:* El método lower() convierte toda la cadena de texto a minúsculas (El método upper() convertiría toda la cadena de texto a mayúsculas).

4. *Línea 15:* El método count() cuenta el número de veces que aparece una subcadena en la cadena de texto. ¡Sí! Hay 4 caracteres "l" en la cadena.

Pongamos un caso muy común. Supón que tienes una cadena de texto en forma de parejas clave-valor, clave1=valor1&clave2=valor2, y quieres dividirla y crear un diccionario de la forma {clave1: valor1, clave2: valor2}.

```
1  >>> consulta = 'usuario=pilgrim&basededatos=master&clave=PapayaWhip'
2  >>> una_lista = consulta.split('&')
3  >>> una_lista
4  ['usuario=pilgrim', 'basededatos=master', 'clave=PapayaWhip']
5  >>> una_lista_de_listas = [v.split('=', 1) for v in una_lista]
6  >>> una_lista_de_listas
7  [['usuario', 'pilgrim'], ['basededatos', 'master'], ['clave', 'PapayaWhip']]
8  >>> a_dict = dict(una_lista_de_listas)
9  >>> a_dict
10 {'clave': 'PapayaWhip', 'usuario': 'pilgrim', 'basededatos': 'master'}
```

1. *Línea 2:* El método split() toma un parámetro, un delimitador, y divide la cadena en una lista de cadenas basándose en el delimitador proporcionado. En este ejemplo, el delimitador es el carácter &.

2. *Línea 5:* Ahora tenemos una lista de cadenas, cada una de ellas con una clave seguida del símbolo = y de un valor. Podemos utilizar las listas por comprensión para iterar sobre esta lista y dividir cada una de estas cadenas de texto en dos cadenas utilizando el método split pasándole un segundo parámetro que indica que únicamente utilice la primera ocurrencia del carácter separador (En teoría una cadena podría tener más de un símbolo igual si el valor, a su vez, contiene también el símbolo igual, por ejemplo: 'clave=valor=cero', con lo que 'clave=valor=cero'.split('=') daría como resultado ['clave', 'valor', 'cero']).

3. *Línea 8:* Finalmente, Python puede convertir esa lista de listas en un diccionario con solo pasarla como parámetro a la función dict().

El ejemplo anterior, explica un caso que se parece a lo que habría que hacer para reconocer los parámetros de una URL. pero en la vida real, el reconocimiento de los parámetros de una URL es más complejo. Si vas a tener que reconocer los parámetros que recibes mediante una URL utiliza la función de la librería urlib.parse denominada parse_qs()[6], que reconoce los casos más complejos.

4.5.1. Troceado de cadenas

Cuando ya has definido una cadena puedes recuperar cualquier parte de ella creando una nueva cadena de texto. A esto se denomina troceado/particionado de cadenas[7]. Esto funciona de forma idéntica a como funciona para las listas, lo que tiene sentido, porque las cadenas de texto no son más que cadenas de caracteres.

```
1  >>> una_cadena = 'Mi vocabulario comienza donde el tuyo termina'
2  >>> una_cadena[3:14]
3  'vocabulario'
4  >>> una_cadena[3:-3]
5  'vocabulario comienza donde el tuyo term'
6  >>> una_cadena[0:2]
7  'Mi'
8  >>> una_cadena[:23]
9  'Mi vocabulario comienza'
10 >>> una_cadena[23:]
11 ' donde el tuyo termina'
```

1. *Línea 2:* Puedes recuperar una parte de la cadena de texto, una parte de ella, especificando dos índices. El valor de retorno es una nueva cadena que comienza en el primer índice y termina en el elemento anterior al segundo índice.

2. *Línea 4:* Como sucede con las listas, puedes utilizar índices negativos para seleccionar.

3. *Línea 6:* Las cadenas también comienzan a contar en cero, por lo que una_cadena[0:2] devuelve los dos primeros elementos de la cadena, comenzando en la posición una_cadena[0] hasta la posición una_cadena[2], pero sin incluirla.

4. *Línea 8:* Si el índice de la parte izquierda vale 0 puedes omitirlo. De este modo, una_cadena[:23] es lo mismo que una_cadena[0:18]. Ya que en ausencia del primer índice se asume el número 0.

[6] http://docs.python.org/3.1/library/urllib.parse.html#urllib.parse.parse_qs
[7] Nota del traductor: slicing en inglés

5. *Línea 10:* De forma similar, si el índice de la parte derecha de la cadena coincide con la longitud de la cadena, puedes omitirlo. Así que una_cadena[23:] es lo mismo que una_cadena[23:45] al medir esta cadena 45 caracteres. Como ves, existe una estupenda simetría en esto, en esta cadena de 45 caracteres una_cadena[0:23] devuelve los primeros 23 caracteres, y una_cadena[23:] devuelve todo lo demás, salvo los 23 caracteres iniciales. De hecho una_cadena[:n] siempre retornará los primeros n caracteres, y una_cadena[n:] retornará el resto, independientemente de la longitud que tenga la cadena.

4.6. Cadenas de texto y Bytes

Los bytes son bytes; los caracteres son una abstracción. A una secuencia inmutable de caracteres Unicode se le llama *cadena de texto*. Una secuencia inmutable de números entre el 0 y el 255 es un objeto que se denomina *bytes*.

```
1  >>> by = b'abcd\x65'
2  >>> by
3  b'abcde'
4  >>> type(by)
5  <class 'bytes'>
6  >>> len(by)
7  5
8  >>> by += b'\xff'
9  >>> by
10 b'abcde\xff'
11 >>> len(by)
12 6
13 >>> by[0]
14 97
15 >>> by[0] = 102
16 Traceback (most recent call last):
17   File "<stdin>", line 1, in <module>
18 TypeError: 'bytes' object does not support item assignment
```

1. *Línea 1:* Para definir un objeto bytes se usa la sintaxis de "literal de bytes" que es b". Cada byte dentro del literal de bytes se interpreta como un carácter ASCII o un carácter codificado en número hexadecimal desde
   x00 a
   xFF (0-255).

2. *Línea 4:* El tipo de un objeto bytes es bytes.

3. *Línea 6:* Como sucede con las listas y cadenas, puedes conocer la longitud de un objeto bytes utilizando la función interna len().

4. *Línea 8:* Como sucede con las listas y cadenas, puedes utilizar el operador + para concatenar objetos bytes. El resultado es un nuevo objeto bytes.

5. *Línea 11:* Concatenar un objeto bytes de 5 bytes con uno de 1 byte da como resultado un objeto bytes de 6 bytes.

6. *Línea 13:* Como sucede con las listas y cadenas, puedes utilizar la notación de índices para obtener bytes individuales del objeto bytes. Los elementos individuales de una cadena son de tipo cadena; los elementos individuales de un objeto bytes son números enteros. Específicamente, enteros entre 0 y 255.

7. *Línea 15:* Un objeto bytes es inmutable; no puedes asignar bytes individuales. Si necesitas modificar bytes individuales de un objeto bytes, es necesario particionar y concatener para crear un nuevo objeto bytes que contenga los elementos deseados. La alternativa es convertir el objeto bytes en un bytearray que sí permite modificación.

```
1 >>> by = b'abcd\x65'
2 >>> barr = bytearray(by)
3 >>> barr
4 bytearray(b'abcde')
5 >>> len(barr)
6 5
7 >>> barr[0] = 102
8 >>> barr
9 bytearray(b'fbcde')
```

1. *Línea 2:* Para convertir un objeto bytes en un objeto modificable de tipo bytearray puedes utilizar la función interna bytearray().

2. *Línea 5:* Todos los métodos y operaciones que existen en el objeto bytes también están disponibles en el objeto bytearray.

3. *Línea 7:* Una de las diferencias es que al objeto bytearray es posible modificarle bytes individuales utilizando la notación de índice. El valor que se puede asignar debe estar entre 0 y 255.

Algo que *no se puede hacer* es mezclar bytes y cadenas.

```
1  >>> by = b'd'
2  >>> s = 'abcde'
3  >>> by + s
4  Traceback (most recent call last):
5    File "<stdin>", line 1, in <module>
6  TypeError: can't concat bytes to str
7  >>> s.count(by)
8  Traceback (most recent call last):
9    File "<stdin>", line 1, in <module>
10 TypeError: Can't convert 'bytes' object to str implicitly
11 >>> s.count(by.decode('ascii'))
12 1
```

1. *Línea 3:* No puedes concatenar bytes y cadenas. Son dos tipos de dato diferentes.

2. *Línea 7:* No puedes contar las veces que aparece una secuencia de bytes en una cadena, porque no existen bytes en una cadena. Una cadena es una secuencia de caracteres. Tal vez lo que querías contar era las veces que aparece la cadena que obtendrías después de haber decodificado la secuencia de bytes interpretándola a partir de una tabla de codificación de caracteres particular. Si es así, debes decirlo explícitamente. Python 3 no convertirá implicítamente bytes en cadenas o cadenas en bytes.

3. *Línea 11:* Por una sorprendente coincidencia esta línea de código dice "cuenta las ocurrencias de la cadena que se obtiene después de decodificar la secuencia de bytes en esta tabla de caracteres particular (ASCII)".

Y de este modo has llegado a la relación que existe entre las cadenas de texto y los bytes: los objetos bytes tienen un método decode() que toma como parámetro una tabla de codificación de caracteres y retorna una cadena. Y las cadenas de texto tienen un método denominado encode() que toma una tabla de codificación de caracteres y retorna un objeto bytes. En el ejemplo anterior, la decodificación fue relativamente directa —convertir una secuencia de bytes que estaba en la codificación de caracteres ASCII en una cadena de texto. Pero el mismo proceso funciona con cualquier tabla de codificación de caracteres siempre que dicha tabla soporte los caracteres existentes en la cadena —incluso con codificaciones heredadas (previas a Unicode).

```
1  >>> s = '快乐 Python'
2  >>> len(s)
3  9
4  >>> by = s.encode('utf-8')
5  >>> by
6  b'\xe5\xbf\xab\xe4\xb9\x90 Python'
7  >>> len(by)
8  13
9  >>> by = s.encode('gb18030')
10 >>> by
11 b'\xbf\xec\xc0\xd6 Python'
12 >>> len(by)
13 11
14 >>> by = s.encode('utf-16')
15 >>> by
16 b'\xff\xfe\xeb_PN \x00P\x00y\x00t\x00h\x00o\x00n\x00'
17 >>> len(by)
18 20
19 >>> vuelta = by.decode('utf-16')
20 '快乐 Python'
21 >>> vuelta == s
22 True
```

1. *Línea 1:* Esto es una cadena de texto, tiene 9 caracteres.
2. *Línea 4:* El resultado de codificar la cadena en UTF-8 es un objeto bytes. Tiene 13 bytes.
3. *Línea 9:* El resultado de codificar la cadena en GB18030 es un objeto bytes de 11 bytes.
4. *Línea 14:* El resultado de codificar la cadena en UTF-16 es un objeto bytes de 20 bytes.
5. *Línea 19:* Al decodificar el objeto bytes utilizando la codificación adecuada (la misma que se usó al codificarlo) se obtiene una cadena de texto. En este caso tiene 9 caracteres. Como se puede ver, es una cadena idéntica a la original.

4.7. Postdata: Codificación de caracteres del código fuente de Python

Python 3 asume que el código fuente —cada fichero .py— está codificado en UTF-8.

En Python 2, la codificación de caracteres por defecto de los ficheros .py era ASCII. En Python 3, la codificación por defecto de los ficheros es UTF-8

Si quisieras utilizar una codificación de caracteres diferente en el fichero con el código fuente, puedes incluir una declaración de codificación de caracteres en la primera línea cada fichero. La siguiente declaración define que el fichero se encuentra en una codificación windows-1252

```
1 # -*- coding: windows-1252 -*-
```

Técnicamente la indicación de la codificación de caracteres puede estar en la segunda línea si la primera línea es una declaración de lanzador de ejecución del estilo de UNIX.

```
1 #!/usr/bin/python3
2 # -*- coding: windows-1252 -*-
```

Para disponer de más información consulta la propuesta de mejora de Python PEP 263[8].

4.8. Lecturas recomendadas

Sobre Unicode en Python:

- Unicode en Python - http://docs.python.org/3.0/howto/unicode.html
- Qué hay nuevo en Python 3: Texto y datos en lugar de Unicode y 8-bits - http://docs.python.org/3.0/whatsnew/3.0.html#text-vs-data-instead-of-unicode-vs-8-bit

Sobre Unicode en general:

- El mínimo absoluto que todo programador debe conocer positiva y absolutamente sobre Unicode y codificación de caracteres (¡Sin excusas!): http://www.joelonsoftware.com/articles/Unicode.html
- Sobre las bondades de Unicode: http://www.tbray.org/ongoing/When/200x/2003/04/06/Unicode

[8] http://www.python.org/dev/peps/pep-0263/

- Sobre cadenas de caracteres:
  http://www.tbray.org/ongoing/When/200x/2003/04/13/Strings
- Caracteres y bytes:
  http://www.tbray.org/ongoing/When/200x/2003/04/26/UTF

Sobre codificación de caracteres en otros formatos:

- Codificación de caracteres en XML:
  http://feedparser.org/docs/character-encoding.html
- Codificación de caracteres en HTML:
  http://blog.whatwg.org/the-road-to-html-5-character-encoding

Sobre cadenas y formateo de cadenas:

- string—Operaciones comunes sobre cadenas:
  http://docs.python.org/3.1/library/string.html
- Sintaxis de formateo de cadenas de texto:
  http://docs.python.org/3.1/library/string.html#formatstrings
- Especificación del minilenguaje de formato:
  http://docs.python.org/3.1/library/string.html#format-specification-mini-language
- PEP 3101: Formateo avanzado de cadenas:
  http://www.python.org/dev/peps/pep-3101/

Capítulo 5

Expresiones regulares

Nivel de dificultad:◆ ◆ ◆ ◇ ◇

"Algunas personas, cuando se enfrentan a un problema, piensan: "¡Ya sé! usaré expresiones regulares". Y así, acaban enfrentándose a dos problemas."
—Jamie Zawinski

5.1. Inmersión

Todo lenguaje de programación moderno dispone de funciones internas para trabajar con cadenas. En Python las cadenas de texto tienen métodos para buscar y reemplazar: index(), find(), split(), count(), replace(), etc. Pero esos métodos están limitados a los casos más simples. Por ejemplo, el método index() busca por una única subcadena, y la búsqueda siempre distingue entre mayúsculas y minúsculas. Para poder hacer búsquedas que no distingan entre ellas debes utilizar s.lower() o s.upper() y asegurarte de que tus cadenas de búsqueda se encuentran en el mismo caso. Los métodos replace() y split().

Si tu objetivo se cumple con estos métodos deberías usarlos. Son rápidos, simples y sencillos de leer; y hay mucho que decir a favor del código legible, simple y rápido. Pero si te descubres escribiendo un montón de funciones para manipular cadenas con sentencias if para contemplar casos especiales, o te encuentras encadenando llamadas a split() y join() para trocear tus cadenas de texto, puede que necesites utilizar *expresiones regulares*.

Las expresiones regualres son una forma poderosa y (en su mayor parte) estándar de búsqueda, reemplazo y análisis de texto con patrones de caracteres

complejos. Aunque la sintaxis de las expresiones regulares es compacta y muy diferente del código *normal*, el resultado puede resultar ser más legible que una solución manual que utilice un montón de funciones de cadenas de texto encadenadas. Incluso existe un modo estándar de incluir comentarios dentro de las expresiones regulares, por lo que puedes incluir una documentación detallada dentro de ellas.

> Si has utilizado expresiones regulares en otros lenguajes (como Perl, JavaScript o PHP), la sintaxis de Python te será muy familiar. Puedes limitarte a leer las funciones disponibles y sus parámetros en el resumen de la documentación del módulo re[1]

5.2. Caso de estudio: direcciones de calles

Esta serie de ejemplos se inspira en un problema de la vida real que tuve en en el trabajo hace varios años, cuando necesité depurar y estandarizar una lista de direcciones postales exportadas de un sistema heredado antes de importarlas en un nuevo sistema[2]. Este ejemplo muestra la forma en la que abordé el problema:

```
1  >> s = '100 NORTH MAIN ROAD'
2  >>> s.replace('ROAD', 'RD.')
3  '100 NORTH MAIN RD.'
4  >>> s = '100 NORTH BROAD ROAD'
5  >>> s.replace('ROAD', 'RD.')
6  '100 NORTH BRD. RD.'
7  >>> s[:-4] + s[-4:].replace('ROAD', 'RD.')
8  '100 NORTH BROAD RD.'
9  >>> import re
10 >>> re.sub('ROAD$', 'RD.', s)
11 '100 NORTH BROAD RD.'
```

1. *Línea 2:* Mi objetivo es estandarizar las direcciones postales de forma que 'ROAD' siempre se escribiera como 'RD.'. En un primer vistazo pensé que era lo suficientemente simple como para que pudiera utilizar el método replace(). Después de todo, las cadenas de texto estaban en mayúsculas por lo que no sería un problema la existencia de posibles minúsculas. Y la cadena de búsqueda, 'ROAD', era una constante. Y en este simple ejemplo s.replace(), de hecho, funciona.

2. *Línea 5:* La vida, desafortunadamente, está llena de contraejemplos, y rápidamente encontré este caso. El problema aquí es que 'ROAD' aparece dos veces

[1] http://docs.python.org/dev/library/re.html#module-contents

[2] Como ves no me invento cosas de la nada, los ejemplos son realmente útiles.

en la dirección, una de ellas siendo parte del nombre de la calle 'BROAD' y otra por sí misma. El método replace() encuentra ambos casos y los reemplaza ciegamente; destruyendo las direcciones.

3. *Línea 7:* Para resolver el problema de las direcciones con más de una ocurrencia de la cadena de texto 'ROAD' puedes recurrir a algo como esto: únicamente buscar y reemplazar 'ROAD' en los últimos cuatro caracteres de la dirección s[-4:], y dejar el resto de la cadena igual, s[:-4]. Como ves, se está volviendo inmanejable. Por ejemplo, la forma la solución depende del tamaño de la cadena de búsqueda. Si intentases sustituir 'STREET' por 'ST.', necesitarías utilizar s[:-6] y s[-6:].replace(...). ¿Te gustaría volver dentro de seis meses a depurar este código? Sé que yo no.

4. *Línea 9:* Es el momento de pasar a las expresiones regulares. En Python esta funcionalidad se incluye en el módulo re.

5. *Línea 10:* Echa un vistazo al primer parámetro: 'ROAD$'. Esta simple expresión regular únicamente encuentra los casos en los que 'ROAD' aparece al final de la línea. El símbolo $ significa "fin de la cadena". Exite otro carácter, el circunflejo: ^, que significa "inicio de la cadena ". Mediante el uso de la función re.sub(), se busca en la cadena s la existencia de la expresión regular 'ROAD$' para sustituirla por 'RD.'. Esto permite encontrar ROAD al final de la cadena s, pero no la parte contenida en BROAD puesto que se encuentra en medio de la cadena s.

Continuando con mi relato sobre la depuración de las direcciones postales, pronto descubrí que el ejemplo anterior, encontrar 'ROAD' al final de la dirección, no era suficiente; no todas las direcciones incluían la destinación de la calle. Algunas direcciones simplemente terminaban con el nombre de la calle. La mayor parte de las veces no pasaba nada, pero si el nombre de la calle era 'BROAD' la expresión regular encontraba 'ROAD' al final de la cadena como parte de la palabra, que no era lo que quería yo.

> ênceuntra el comienzo de una cadena. $ encuentra el final.

```
1  >>> s = '100 BROAD'
2  >>> re.sub('ROAD$', 'RD.', s)
3  '100 BRD.'
4  >>> re.sub('\bROAD$', 'RD.', s)
5  '100 BROAD'
6  >>> re.sub(r'\bROAD$', 'RD.', s)
7  '100 BROAD'
8  >>> s = '100 BROAD ROAD APT. 3'
9  >>> re.sub(r'\bROAD$', 'RD.', s)
10 '100 BROAD ROAD APT. 3'
11 >>> re.sub(r'\bROAD\b', 'RD.', s)
12 '100 BROAD RD. APT 3'
```

1. *Línea 4:* Lo que yo realmente *quería* era buscar 'ROAD' cuando estuviera al final de la línea y fuese una palabra por sí misma (y no parte de una palabra mayor). Para expresar esto en una expresión regular debes utilizar \b, que indica "que un límite de palabra debe existir en ese lugar". En Python, expresar esta cadena es algo complicado debido a que el símbolo '\' suele indicar un carácter de escape, y hay que escribirlo dos veces para representarlo como tal. Esto hay quien lo llama la *plaga de las barras inclinadas invertidas*, y es el argumento para decir que las expresiones regulares son más sencillas en Perl que en Python. En el lado negativo, Perl mezcla la sintaxis de las expresiones regulares con otra sintaxis, por lo que si tienes un error, es más difícil saber si el error es en la sintaxis o en la expresión regular.

2. *Línea 6:* Para evitar la plaga de las barras inclinadas invertidas puedes utilizar lo que se llaman *cadenas de texto "crudas"*[3] mediante el uso del prefijo r delante de la cadena de texto. Este tipo de cadena de texto le dice a Python que nada de lo que contiene es un carácter de escape: la cadena '\t' representa al carácter tabulador, pero r'\t' es una cadena que contiene como primer carácter la barrra inclinada invertida seguida de la letra t. Por eso, recomiendo que siempre utilices cadenas de texto "crudas" cuando vayas a escribir una expresión regular; en caso contrario, las cosas se vuelven confusas en cuanto la expresión regular es algo compleja (y las expresiones regulares ya confunden suficientemente por sí mismas).

3. *Línea 9:* Desafortunadamente rápidamente encontré más casos que contradijeron mi razonamiento. En este caso la dirección contenía la palabra 'ROAD' pero no al final de la cadena de texto, ya que contenía también el número del apartamento después de la designación de la calle. Al no encontrarse al final de la cadena, no se sustituye nada porque la expresión regular no coincide.

[3] Nota del traductor: "raw" en inglés.

4. *Línea 11:* Para resolver este problema acabé quitando el carácter $ y poniendo otro \b. De esta forma la expresión regular significa "encuentra ROAD cuando es una palabra completa en cualquier parte de la cadena", tanto si está al principio, al final o en cualquier otra parte.

5.3. Caso de estudio: números romanos

Es muy probable que hayas visto números romanos en alguna parte incluso aunque no los hayas reconocido. Puedes haberlos visto en los créditos de las películas antiguas o programas de televisión ("Copyright MCMXLVI") o en las paredes de las bibliotecas y universidades ("Establecido en MDCCCLXXXVIII" en lugar de "establecido en 1888"). Puede que incluso los hayas visto en referencias bibliográficas. Es un sistema de representación numérica que se remonta a la época del imperio romano (de ahí el nombre).

En los números romanos existen siete caracteres que se repiten y combinan de diferentes formas para representar números:

- I = 1
- V = 5
- X = 10
- L = 50
- C = 100
- D = 500
- M = 1000

Existen una reglas generales para construir números romanos:

- Los caracteres son aditivos, I es 1, II es 2 y III es 3. VI es 6 (literalmente 5 + 1), VII es 7 (5+1+1) y XVIII es 18 (10+5+1+1+1).
- Los caracteres que representan unidades, decenas, centenas y unidades de millar (I, X, C y M) pueden aparecer juntos hasta tres veces como máximo. Para el 4 debes restar del carácter V, L ó D (cinco, cincuenta, quinientos) que se encuentre más próximo a la derecha. No se puede representar el cuatro como IIII, en su lugar hay que poner IV (5-1). El número 40 se representa como XL (10 menos que 50: 50-10). 41 = XLI, 42 = XLII, 43 = XLIII y luego 44 = XLIV (diez menos que cincuenta más uno menos que cinco: 50-10+5-1).

- De forma similar, para el número 9, debes restar del número siguiente más próximo que represente unidades, decenas, centenas o unidades de millar (I, X, C y M). 8 = VIII, pero 9 = IX (1 menos que 10), no 9 = VIIII puesto que el carácter I no puede repetirse cuatro veces seguidas. El número 90 se representa con XC y el 900 con CM.

- Los caracteres V, L y D no pueden repetirse; el número 10 siempre se representa como X y no como VV. El número 100 siempre se representa como C y nunca como LL.

- Los números romanos siempre se escriben de los caracteres que representan valores mayores a los menores y se leen de izquierda a derecha por lo que el orden de los caracteres importa mucho. {DC es el número 600; CD otro número, el 400 (500 - 100). CI es 101, mientras que IC no es un número romano válido porque no puedes restar I del C[4].

5.3.1. A búsqueda de coincidencias de las unidades de millar

.

¿Qué costaría conocer que una cadena de texto es un número romano válido? Vamos a hacerlo dígito a dígito para facilitar la tarea y la explicación. Puesto que los números romanos siempre se escriben del mayor al menor, vamos a comenzar por el mayor: las unidades de millar. Para los números 1000 y superiores los miles se representan por una serie de caracteres M.

```
1  >>> import re
2  >>> pattern = '^M?M?M?$'
3  >>> re.search(pattern, 'M')
4  <SRE_Match object at 0106FB58>
5  >>> re.search(pattern, 'MM')
6  <SRE_Match object at 0106C290>
7  >>> re.search(pattern, 'MMM')
8  <SRE_Match object at 0106AA38>
9  >>> re.search(pattern, 'MMMM')
10 >>> re.search(pattern, '')
11 <SRE_Match object at 0106F4A8>
```

1. *Línea 2:* Este patrón tiene tres partes: ^ identifica el comienzo de la línea únicamente. Si no lo indicáramos así, el resto del patrón validaría cualquier posición dentro de una cadena en la que se encontrase, cosa que no quieres. Lo que quieres es estar seguro de que los caracteres M se encuentran al comienzo

[4]Para representar el 99 deberías usar: XCIL (100 - 10 + 10 - 1)

de la cadena. M? indica que se valide si existe un carácter M de forma opcional en la posición indicada. Como se repite tres veces, lo que estás diciendo es que se valide si existe el carácter M de cero a tres veces (al principio de la cadena debido al ^). Y por último, $ valida el final de la línea. Cuando se combina con el carácter ^ al comienzo, significa que el patrón debe coincidir con la cadena de texto completa, por lo que en este caso únicamente es posible que tenga de cero a tres caracteres M'.

2. *Línea 3:* La esencia del módulo re es la función search(), que toma como primer parámetro una expresión regular (pattern) y como segundo una cadena de texto que es la que se comprobará para ver si coincide con la expresión regular. Si se encuentra una cadena identificada por las expresión regular, esta función retorna un objeto que tiene diversos métodos para describir la cadena encontrada. Si no se encuentra nada equivalente al patrón, la función retorna None. Por ahora lo único que te interesa es conocer si una cadena cumple el patrón, y para ello basta ver qué valor retorna la función search. 'M' cumple la expresión regular, porque el primer M opcional coincide con el primer caracter de la cadena y las siguientes M del patrón son ignoradas.

3. *Línea 5:* 'MM' cumple la expresión regular porque el primer y segundo M opcional coinciden con la primera y segunda letra M de la cadena. La tercera M opcional se ignora.

4. *Línea 7:* 'MMM' cumple la expresión regular porque los tres caracteres M coinciden.

5. *Línea 9:* 'MMMM no cumple la expresión regular. Los tres primeros caracteres coinciden, pero en este momento la expresión regular insiste en que se debe terminar la cadena (debido al carácter $), pero la cadena de texto no ha finalizado aún, existe una cuarta M. Por eso la función search() retorna None.

6. *Línea 10:* La cadena de textos vacía también cumple el patrón puesto que los tres caracteres M son opcionales.

> ? hace que un patrón sea opcional.

5.3.2. A la búsqueda de coincidencias de las centenas

Las centenas son más difíciles que los miles, porque existen varias formas exclusivas de representación dependiendo del valor.

- 100 = C

- 200 = CC
- 300 = CCC
- 400 = CD
- 500 = D
- 600 = DC
- 700 = DCC
- 800 = DCCC
- 900 = CM

Por lo que existen cuatro patrones posibles:

- CM
- CD
- De cero a tres caracteres C (Cero si el lugar de las centenas vale cero).
- D, seguido de cero a tres caracteres C.

Los dos últimos patrones se pueden combinar:

- Una D opcional, seguida de cero a tres caracteres C.

Este ejemplo muestra cómo validar las centenas de un número romano.

```
>>> import re
>>> pattern = '^M?M?M?(CM|CD|D?C?C?C?)$'
>>> re.search(pattern, 'MCM')
<SRE_Match object at 01070390>
>>> re.search(pattern, 'MD')
<SRE_Match object at 01073A50>
>>> re.search(pattern, 'MMMCCC')
<SRE_Match object at 010748A8>
>>> re.search(pattern, 'MCMC')
>>> re.search(pattern, '')
<SRE_Match object at 01071D98>
```

1. *Línea 2:* El patrón de esta expresión regular comienza igual que el anterior, se valida el comienzo de la cadena (^), luego de cero a tres unidades de millar (M?M?M?). Luego viene la parte nueva, se definen tres conjuntos de patrones mutuamente excluyentes. Para ello se utilizan los paréntesis y la barra vertical: CM, CD y D?C?C?C? (este último representa a una D opcional seguida de cero a tres C). El analizador de expresiones regulares comprueba cada uno de estos patrones en el orden en que se muestran (de izquierda a derecha), toma el primero que coincide e ignora el resto. Si no coincide ninguno, falla la búsqueda en la cadena de texto.

2. *Línea 3:* 'MCM' cumple el patrón porque la primera M coincide, la segunda y la tercera del patrón se ignoran, y la CM coincide (los patrones CD y D?C?C?C? no llegan a considerarse). MCM es la representación romana del 1900.

3. *Línea 5:* 'MD' cumple porque la primera M coincide, la segunda y tercera M del patrón se ignoran, y el patrón D?C?C?C? coincide en la D (los tres caracteres C son opcionales). MD es el número romano 1500.

4. *Línea 7:* 'MMMCCC' cumple la expresión regular porque coinciden las tres M opcionales, y el patrón D?C?C?C? coincide con CCC (la D es opcional y se ignora). MMMCCC es el número romano que representa el 3300.

5. *Línea 9:* 'MCMC' no cumple la expresión. La primera M del patrón coincide, las siguientes se ignoran, y CM coincide, pero al final $ porque espera que se haya acabado la cadena pero aún queda la C en la cadena de texto. La C no coincide como parte del patrón D?C?C?C? porque es mutuamente excluyente con el patrón CM que es el que se ha cumplido anteriormente.

6. *Línea 10:* La cadena vacía aún cumple este patrón. puesto que todos los caracteres M son opcionales y la cadena vacía también coincide con el patrón D?C?C?C? en el que todos los caracteres son también opcionales.

¡Vaya! ¿Ves qué fácil es que las expresiones regulares se vayan complicando? Y por ahora únicamente hemos incluido las unidades de millar y las centenas de los números romanos. Pero si has llegado hasta aquí, ahora las decenas y las unidades serán fáciles para tí, porque siguen exactamente el mismo patrón. Vamos a ver otra forma de expresar la expresión regular.

5.4. Utilización de la sintaxis {n,m}

En la sección anterior viste casos de caracteres que se podían repetir hasta tres veces. Existe otra forma de representar esto en las expresiones regulares que puede

resultar más legible. Primero observa el método que ya hemos usado en el ejemplo anterior.

```
1 >>> import re
2 >>> pattern = '^M?M?M?$'
3 >>> re.search(pattern, 'M')
4 <_sre.SRE_Match object at 0x008EE090>
5 >>> pattern = '^M?M?M?$'
6 >>> re.search(pattern, 'MM')
7 <_sre.SRE_Match object at 0x008EEB48>
8 >>> pattern = '^M?M?M?$'
9 >>> re.search(pattern, 'MMM')
10 <_sre.SRE_Match object at 0x008EE090>
11 >>> re.search(pattern, 'MMMM')
12 >>>
```

1. *Línea 3:* El patrón coincide con el inicio de la cadena, luego con la primera M, las restantes se ignoran por ser opcionales, para terminar coincidiendo el $ con el final de la cadena.

2. *Línea 6:* El patrón coincide con el inicio de la cadena, luego coinciden la primera y segunda M, la tercera se ignora, para terminar encontrando el final de la cadena.

3. *Línea 9:* El patrón coincide con el inicio de la cadena, luego las tres M para terminar con el final de la cadena.

4. *Línea 11:* El patrón coincide con el inicio de la cadena y las tres M, pero luego no coincide con el final de la cadena, que es lo que se espera, puesto que aún queda en la cadena una letra M que falta por coincidir. Por eso el patrón no se cumple y se devuelve None.

```
1 >>> pattern = '^M{0,3}$'
2 >>> re.search(pattern, 'M')
3 <_sre.SRE_Match object at 0x008EEB48>
4 >>> re.search(pattern, 'MM')
5 <_sre.SRE_Match object at 0x008EE090>
6 >>> re.search(pattern, 'MMM')
7 <_sre.SRE_Match object at 0x008EEDA8>
8 >>> re.search(pattern, 'MMMM')
9 >>>
```

1. *Línea 1:* Este patrón dice: Busca el comienzo de la cadena, luego busca de cero a tres M y termina con el final de la cadena. Los números 0 y 3 del ejemplo se pueden sustituir por cualquier combinación; si lo que quisieras fuera que al menos hubiera una M podrías utilizar M{1,3}.

2. *Línea 2:* El patrón coincide con el comienzo de la cadena, luego con la M de las tres posibles y termina encontrando el final de la cadena.

3. *Línea 4:* El patrón coincide con el comienzo de la cadena, luego con las dos M de las tres posibles y termina encontrando el final de la cadena.

4. *Línea 6:* El patrón coincide con el comienzo de la cadena, luego con las tres M y termina encontrando el final de la cadena.

5. *Línea 8:* El patrón coincide con el inicio de la cadena y las tres M, pero luego *no coincide* con el final de la cadena, que es lo que se espera, puesto que aún queda en la cadena una letra M que falta por coincidir. Dicho de otro modo, el patrón espera únicamente un máximo de tres M antes del final de la cadena pero en este caso hay cuatro. Por eso el patrón no se cumple y se devuelve None.

> {1,4} busca la coincidencia de una a cuatro veces del patrón relacionado.

5.4.1. Comprobación de las decenas y las unidades

Vamos a completar la expresión regular de búsqueda de números romanos para incluir a las decenas y unidades. Este ejemplo incluye las decenas.

```
1  >>> pattern = '^M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)$'
2  >>> re.search(pattern, 'MCMXL')
3  <_sre.SRE_Match object at 0x008EEB48>
4  >>> re.search(pattern, 'MCML')
5  <_sre.SRE_Match object at 0x008EEB48>
6  >>> re.search(pattern, 'MCMLX')
7  <_sre.SRE_Match object at 0x008EEB48>
8  >>> re.search(pattern, 'MCMLXXX')
9  <_sre.SRE_Match object at 0x008EEB48>
10 >>> re.search(pattern, 'MCMLXXXX')
11 >>>
```

1. *Línea 2:* El patrón encuentra el comienzo de la cadena, luego la primera M opcional, luego CM, luego XL y termina detectando el final de la cadena. Recuerda que la sintaxis (A—B—C) significa que se encuentre únicamente una de las tres coincidencias. Al coincidir con XL se ignoran XC y L?X?X?X?. MCMXL es el número romano 1940.

2. *Línea 4:* Se encuentra el comienzo de la cadena, luego la primera M opcional, seguido de CM, luego L?X?X?X?. De esta última coincide L y se ignoran las

tres X. Se finaliza al encontrar el final de la cadena. El número romano MCML representa al 1950.

3. *Línea 6:* Al mirar el patrón en esta cadena se encuentra el comienzo de la misma, luego la primera M opcional, seguido de CM, luego L?X?X?X?. Del patrón L?X?X?X? coincide con la L, luego con la primera X opcional y se ignoran la segunda y tercera X. Se termina encontrando el final de la cadena. El número romano MCMLX es 1960.

4. *Línea 8:* El patrón encuentra el comienzo de la misma, luego la primera M opcional, seguido de CM, luego L?X?X?X? sirve para identificar LXXX, terminando al encontrar el final de la cadena. El número romano MCMLXXX es 1980.

5. *Línea 10:* Encuentra el comienzo de la cadena, luego la primera M opcional, luego CM, luego la L opcional y las tres X opcionales, después de este punto falla la comprobación, se espera el final de la cadena pero queda una X. Por eso, falla toda la expresión regular y se devuelve None. MCMLXXXX no es un número romano válido.

> (A|B) coincide si la cadena contiene o el patrón A o el B

La expresión para las unidades sigue un patrón idéntico. Te ahorraré los detalles mostrándote el resultado final.

```
1 >>> pattern = '^M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$'
```

¿Cómo quedaría utilizando la sintaxis {n,m} En el siguiente ejemplo observas el resultado.

```
1 >>> pattern = '^M{0,3}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$'
2 >>> re.search(pattern, 'MDLV')
3 <_sre.SRE_Match object at 0x008EEB48>
4 >>> re.search(pattern, 'MMDCLXVI')
5 <_sre.SRE_Match object at 0x008EEB48>
6 >>> re.search(pattern, 'MMMDCCCLXXXVIII')
7 <_sre.SRE_Match object at 0x008EEB48>
8 >>> re.search(pattern, 'I')
9 <_sre.SRE_Match object at 0x008EEB48>
```

1. *Línea 2:* Encuentra el comienzo de la cadena, luego una de las tres posibles M, luego D?C{0,3}. De esta última subexpresión, coincide con la D y con cero de las tres posibles C. Lo siguiente que hace es coincidir con L?X{0,3} al encontrar la L y cero de tres X. Luego coincide con V?I{0,3} al encontrar la V y cero de

tres posibles I. Se termina con la coincidencia satisfecha al encontrar el fin de la cadena. El número romano MDLV representa al 1555.

2. *Línea 4:* Encuentra el comienzo de la cadena, luego dos de las tres posibles M; luego D?C{0,3} encontrando D y una C; luego coincide con L?X{0,3} al encontrar L y una X; luego coincide con V?I{0,3} al encontrar V y una de las tres posibles I, y termina con al final de la cadena. El número romano MMDCLXVI es el 2666.

3. *Línea 6:* Encuentra el comienzo de la cadena, luego las tres posibles M; luego D?C{0,3} encuentra D y tres C; luego coincide con L?X{0,3} al encontrar L y tres X; luego coincide con V?I{0,3} al encontrar V y las tres posibles I, y termina con al final de la cadena. El número romano MMMDCCCLXXXVIII es el 3888 y es el mayor que se puede escribir sin utilizar una sintaxis extendida.

4. *Línea 8:* Observa atentamente (me siento como un mago, "mirad atentamente niños, voy a sacar un conejo de mi chistera"). En este caso se encuentra el comienzo de la cadena, luego ninguna M de las tres posibles, luego coincide con D?C{0,3} a saltarse la D opcional y encontrar cero caracteres C, luego coincide con L?X{0,3} por el mismo motivo que antes, y lo mismo sucede con V?I{0,3}. Luego encuentra el final de la cadena. ¡Fantástico!

Si has seguido todas las explicaciones y las has entendido a la primera ya lo estás haciendo mejor que lo hice yo. Ahora imagínate tratando de comprender las expresiones regulares que haya escrito otra persona en medio de una función crítica para tu programa. O piensa simplemente en tener que volver a ver las expresiones regulares que escribiste hace unos meses. Yo lo he tenido que hacer y no es agradable.

Vamos a ver ahora una sintaxis alternativa que puede ayudarte que las expresiones regulares sean más comprensibles.

5.5. Expresiones regulares detalladas

Hasta ahora has visto lo que llamaré "expresiones regulares compactas". Son difíciles de leer e incluso, aunque averigües lo que hacen, no puedes estar seguro de acordarte seis meses después. En realidad, únicamente necesitas documentarlas. Para eso puedes utilizar la documentación "incrustada"[5].

Python te permite hacerlo mediante el uso de las *expresiones regulares detalladas*. Se diferencia de una expresión regular compacta en dos cosas:

[5]Nota del Traductor: **inline** en inglés.

- Los espacios en blanco se ignoran: espacios, tabuladores y retornos de carro o saltos de línea. No se tienen en cuenta para el cumplimiento o no de la expresión regular. Si quieres que exista realmente tal coincidencia será necesario que incluyas el carácter de escape delante de ellos.

- Se ignoran los comentarios. Un comentario en una expresión regular detallada es exactamente igual que en Python: comienza en el carácter # termina al acabarse la línea. En este caso, el comentario está incluido en una cadena de texto de varias líneas en lugar de estar directamente en ti código fuente, pero por lo demás es exactamente lo mismo.

```
1  >>> pattern = '''
2      ^                   # comienzo de la cadena
3      M{0,3}              # unidades de millar - 0 a 3 M
4      (CM|CD|D?C{0,3})    # centenas - 900 (CM), 400 (CD), 0-300 (0 a 3 C),
5                          #            o 500-800 (D, seguido po 0 a 3 C)
6      (XC|XL|L?X{0,3})    # decenas - 90 (XC), 40 (XL), 0-30 (0 a 3 X),
7                          #           o 50-80 (L, seguido de 0 a 3 X)
8      (IX|IV|V?I{0,3})    # unidades - 9 (IX), 4 (IV), 0-3 (0 a 3 I),
9                          #            o 5-8 (V, seguido de 0 a 3 I)
10     $                   # fin de la cadena
11     '''
12 >>> re.search(pattern, 'M', re.VERBOSE)
13 <_sre.SRE_Match object at 0x008EEB48>
14 >>> re.search(pattern, 'MCMLXXXIX', re.VERBOSE)
15 <_sre.SRE_Match object at 0x008EEB48>
16 >>> re.search(pattern, 'MMMDCCCLXXXVIII', re.VERBOSE)
17 <_sre.SRE_Match object at 0x008EEB48>
18 >>> re.search(pattern, 'M')
```

1. *Línea 12:* Lo más importante es recordar que cuando utilizas expresiones regulares detalladas debes añadir un parámetro más a la llamada a la función: re.VERBOSE es una constante definida en el módulo re que permite indicar a la función que el patrón debe interpretarse como una expresión regular detallada. Como puedes ver, este patrón tiene muchos espacios (que son ignorados) y varios comentarios (también ignorados). Una vez se han suprimido los espacios y los comentarios, la expresión regular es exactamente la misma que viste en la primera parte de este capítulo, solo que mucho más legible.

2. *Línea 14:* Esta expresión encuentra en primer lugar el comienzo de la cadena, luego una de las tres posibles M, luego CM, luego L y las tres posibles X, luego IX y se termina con el fin de la cadena.

3. *Línea 16:* Esta expresión encuentra en primer lugar el comienzo de la cadena, luego las tres posibles M, luego D y las tres posibles C, luego L y las tres posibles X, luego V y las tres posibles I y se termina con el fin de la cadena.

4. *Línea 18:* No se cumple. ¿Porqué? Porque no se le ha pasado el parámetro re.VERBOSE, por lo que la función re.search() está tratando a la expresión regular como si no fuese detallada, por lo que ha intentado encontrar en la cadena 'M' todos los espacios y comentarios que hemos introducido en el patrón. Python no puede detectar automáticamente si una expresión es detallada o no. Python asume siempre que una expresión regular es compacta, a no ser que se le diga lo contrario.

5.6. Caso de estudio: análisis de números de teléfono

Hasta el momento te has concentrado en identificar patrones completos. O el patrón coincide o no. Pero las expresiones regulares permiten hacer mucho más. Cuando una cadena de texto coincide con el patrón de la expresión regular puedes recuperar trozos de la cadena a partir del objeto que se retorna. Puedes recuperar qué partes de la cadena han coincidido con qué partes de la expresión regular.

Este ejemplo está sacado de otro problema real que me encontré en el trabajo. El problema: analizar un número de teléfono americano. El cliente quería que el número se introdujese de forma libre (en un único campo de texto en pantalla), pero luego quería almacenar en la base de datos de la compañía de forma separada el código de área, principal, número y opcionalmente, una extensión. Miré un poco en la web buscando ejemplos de expresiones regulares que hicieran esto, pero no encontré ninguna suficientemente permisiva.

\d coincide con cualquier dígito numérico (0-9). \D coincide con cualquier carácter que no sea dígito.

A continuación muestro algunos ejemplos de números de teléfono que necesitaba que se pudieran aceptar:

- 800-555-1212
- 800 555 1212
- 800.555.1212
- (800) 555-1212
- 800-555-1212-1234

- 800-555-1212x1234
- 800-555-1212 ext. 1234
- work 1-(800) 555.1212 #1234

¡Cuánta variedad! En cada uno de los casos anteriores necesitaba identificar el código de área 800, el troncal 500 y el resto del número 1212. Asimismo, para aquellos que presentaban la extensión necesitaba conocer que era 1234.

```
1 >>> phonePattern = re.compile(r'^(\d{3})-(\d{3})-(\d{4})$')
2 >>> phonePattern.search('800-555-1212').groups()
3 ('800', '555', '1212')
4 >>> phonePattern.search('800-555-1212-1234')
5 >>> phonePattern.search('800-555-1212-1234').groups()
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8 AttributeError: 'NoneType' object has no attribute 'groups'
```

1. *Línea 1:* Las expresiones regulares siempre se leen de izquierda a derecha. Esta detecta el comienzo de la línea y luego (\d{3}). ¿Qué es (\d{3})? Bueno, \d significa "cualquier dígito numérico" (0 a 9). El {3} significa que "coincida exactamente tres veces" (tres dígitos en este caso), es una variación de la sintaxis {n,m} que viste antes. Al ponerlo todo entre paréntesis se está diciendo "identifica exactamente tres dígitos numéricos y *recuérdalos como un grupo para que te pueda preguntar por ellos más tarde*"

2. *Línea 2:* Para acceder a los grupos, que se guardan durante la búsqueda de la coincidencia de la cadena con expresión regular, se utiliza el método groups() en el objeto que devuelve el método search(). Este métod (groups()) retornará una tupla con tantos elementos como grupos se definieran en la expresion regular. En este caso se han definido tres grupos, dos con tres dígitos y un tercero con cuatro.

3. *Línea 4:* Esta expresión regular no es perfecta, porque no es capaz de manejar números telefónicos con extensiones. Por eso será necesario expandir la expresión regular.

4. *Línea 5:* Y en esta línea se observa el porqué no debes encadenar en una única sentencia search() y group(). Si la función search() no encuentra ninguna coincidencia en la cadena devuelve None, no un objeto de coincidencia. La llamada a None.groups() eleva una excepción perfectamente clara: None no dispone de la función groups(). (Por supuesto, no será tan fácil ni tan obvio si la excepción sucede en lo más profundo de tu programa. Sí, hablo por experiencia.)

```
1 >>> phonePattern = re.compile(r'^(\d{3})-(\d{3})-(\d{4})-(\d+)$')
2 >>> phonePattern.search('800-555-1212-1234').groups()
3 ('800', '555', '1212', '1234')
4 >>> phonePattern.search('800 555 1212 1234')
5 >>>
6 >>> phonePattern.search('800-555-1212')
7 >>>
```

1. *Línea 1:* Esta expresion regular es prácticamente igual a la anterior. Como antes, coincide con el inicio de la cadena, luego con un grupo de tres dígitos decimales, luego con el guión, otro grupo de tres dígitos decimales, otro guión, y luego con el grupo de cuatro dígitos decimales. Lo que es nuevo es que después debe coincidir con otro guión seguido de uno o más dígitos decimales, para terminar con el fin de la cadena.

2. *Línea 2:* El método groups() ahora retorna una tupla de cuatro elementos, puesto que la expresión regular ahora define cuatro grupos a recordar.

3. *Línea 4:* Desafortunadamente esta expresión regular tampoco es la solución definitiva, porque asume siempre que las diferentes partes del teléfono están separadas por guiones. ¿Qué pasa si están separadas por puntos, espacios o comas? Necesitas una solución más general para que coincida con diferentes tipos de separadores.

4. *Línea 6:* ¡Ups! No solamente no hace todo lo que queremos sino que en realidad es un paso atrás, puesto que ahora no es capaz de identificar números de teléfono sin extensiones. Esto no era lo que querías, si la extensión está ahí quieres conocerla, pero si no está sigues queriendo conocer las diferentes partes del número de teléfono.

El siguiente ejemplo muestra una expresión regular para manejar diferentes separadores entre las diferentes partes del número de teléfono.

```
1 >>> phonePattern = re.compile(r'^(\d{3})\D+(\d{3})\D+(\d{4})\D+(\d+)$')
2 >>> phonePattern.search('800 555 1212 1234').groups()
3 ('800', '555', '1212', '1234')
4 >>> phonePattern.search('800-555-1212-1234').groups()
5 ('800', '555', '1212', '1234')
6 >>> phonePattern.search('80055512121234')
7 >>>
8 >>> phonePattern.search('800-555-1212')
9 >>>
```

1. *Línea 1:* Agárrate a tu sombrero. Primero se busca la coincidencia del inicio de la cadena, luego un grupo de tres caracteres y luego \D+. ¿Qué es eso? Bueno, \D coincide con cualquier carácter que no sea dígito, y + significa "1 o más". Por eso esta expresión \D+ encuentra uno o más caracteres que no sean dígitos. Esto es lo que vamos a utilizar para intentar identificar muchos tipos diferentes de separadores (en lugar de únicamente el guión).

2. *Línea 2:* El uso de \D+ en lugar de - significa que puedes encontrar números de teléfono que utilicen espacios como separadores en lugar de guiones.

3. *Línea 4:* Por supuesto aún se encuentran los números de teléfonos que utilizan como separador el guión.

4. *Línea 6:* Desafortunadamente aún no tenemos la respuesta final porque se está asumiendo que debe haber al menos un separador. ¿Qué pasa si el número no tiene ningún separador?

5. *Línea 8:* ¡Ups! Y aún no se ha arreglado el problema de que sea opcional, y no obligatoria, la existencia de las extensiones. Ahora tienes dos problemas, pero se pueden resolver ambos con la misma técnica.

El siguiente ejemplo muestra una expresión regular para manejar números telefónicos sin separadores.

```
1 >>> phonePattern = re.compile(r'^(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$')
2 >>> phonePattern.search('80055512121234').groups()
3 ('800', '555', '1212', '1234')
4 >>> phonePattern.search('800.555.1212 x1234').groups()
5 ('800', '555', '1212', '1234')
6 >>> phonePattern.search('800-555-1212').groups()
7 ('800', '555', '1212', '')
8 >>> phonePattern.search('(800)5551212 x1234')
9 >>>
```

1. *Línea 1:* El único cambio que has hecho desde el último caso es cambiar + por *. En lugar de \D+ entre las partes del número de teléfono, ahora utilizas \D*. ¿Recuerdas que + significa "1 o más"? Pues * significa "0 o más". Por eso ahora deberías ser capaz de encontrar los números de teléfono incluso cuando no exista ningún separador.

2. *Línea 2:* ¡Funciona! ¿Porqué? Encuentras el comienzo de la cadena, luego un grupo de tres dígitos (800), luego cero caracteres no númericos, luego otro grupo de tres dígitos (555), luego cero caracteres no numéricos, luego un grupo de cuatro dígitos (1212), luego cero caracteres no numéricos, luego un grupo arbitrario de dígitos (1234) y luego el fin de la cadena.

3. *Línea 4:* También funcionan otras variaciones: puntos en lugar de guiones, y espacios o x antes de la extensión.

4. *Línea 6:* También se ha resuelto finalmente el problema de las extensiones. Vuelven a ser opcionales. Si no se encuentra la extensión, la función groups() sigue retornando una tupla de cuatro elementos, pero el cuarto elemento es una cadena vacía.

5. *Línea 8:* Odio tener que ser el portador de malas noticias, pero aún no hemos terminado. ¿Cuál es el problema aquí? Existe un carácter extra antes del código de área pero la expresión regular asume que el primer carácter debe ser lo primero de la cadena. No pasa nada, puedes utilizar la misma técnica de "cero o más caracteres no numéricos" para saltarte todos los caracteres no numéricos iniciales.

El siguiente ejemplo muestra como manejar los caracteres previos al número de teléfono:

```
1 >>> phonePattern = re.compile(r'^\D*(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$')
2 >>> phonePattern.search('(800)5551212 ext. 1234').groups()
3 ('800', '555', '1212', '1234')
4 >>> phonePattern.search('800-555-1212').groups()
5 ('800', '555', '1212', '')
6 >>> phonePattern.search('work 1-(800) 555.1212 #1234')
7 >>>
```

1. *Línea 1:* Esto es igual que en el ejemplo anterior salvo que ahora estás buscando \D*, cero o más caracteres numéricos, antes del primero grupo (el código de área). Observa que no se "recuerdan" esos caracteres, no están en un grupo (no están entre paréntesis). Si aparecen, simplemente se saltan y luego se comienza a reconocer el codigo de área que será almacenado por la búsqueda (al encontrarse entre paréntesis).

2. *Línea 2:* Ahora puedes reconocer satisfactoriamente el número de teléfono incluso con el paréntesis de la izquierda antes del código de área. El paréntesis de la derecha del código de área ya estaba controlado, se trata como un carácter no numérico y se detecta con el \D* posterior al primer grupo.

3. *Línea 4:* Solamente, para tener la seguridad de que no has roto nada que funcionase. Puesto que los caracteres iniciales son totalmente opcionales, primero detecta el comienzo de la cadena, luego cero caracteres no numéricos, luego un grupo de tres dígitos de área (800), luego un carácter no numérico (el guión), luego un grupo de tres dígitos (555), luego un carácter no numérico (el guión),

luego un grupo de cuatro dígitos (1212), luego cero caracteres no numéricos, luego un grupo de cero dígitos opcionales, para terminar con el final de la cadena.

4. *Línea 6:* Por eso las expresiones regulares me hacen querer sacarme los ojos con un objeto punzante. ¿Porqué no funciona con este número de teléfono? Porque hay un 1 antes del código de área pero has asumido que todos los caracteres que van delante del código de área serán no numéricos (\D*), ¡¡¡Aarggghhhh!!!

Vamos a resumir por un momento. Hasta ahora todas las expresiones regulares han buscado desde el comienzo de la cadena. Pero ahora ves que existe un número indeterminado de cosas al comienzo que puede interesarte ignorar. En lugar de intentar hacer coincidir todas las combinaciones posibles lo mejor es que las ignores. Vamos a intentarlo de una forma distinta: sin expresar explícitamente el comienzo de la cadena. Esta opción se muestra en el siguiente ejemplo.

```
1 >>> phonePattern = re.compile(r'(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$')
2 >>> phonePattern.search('work 1-(800) 555.1212 #1234').groups()
3 ('800', '555', '1212', '1234')
4 >>> phonePattern.search('800-555-1212')
5 ('800', '555', '1212', '')
6 >>> phonePattern.search('80055512121234')
7 ('800', '555', '1212', '1234')
```

1. *Línea 1:* Observa la ausencia del ^ en esta expresión regular, ya no vas a obligar a que la coincidencia sea desde el comienzo de la cadena. No hay nada que diga que tienes que hacer coincidir la cadena completa. El motor de proceso de las expresiones regulares se encargará de hacer el trabajo duro descubriend el lugar en el que la cadena de entrada comienza a coincidir.

2. *Línea 2:* Ahora puedes analizar un número de teléfono que incluya caracteres y un dígito previo, además de cualquier clase de separadores alrededor de cada parte del número.

3. *Línea 4:* Test de seguridad. Aún funciona.

4. *Línea 6:* Y esto también.

¿Ves qué rápido comienza uno a perder el control de las expresiones regulares? Échale un vistazo a cualquiera de los ejemplos anteriores. ¿Puedes identificar aún las diferencias entre uno y otro?

Aunque puedas comprender la respuesta final (y es la última respuesta, y si encuentras un caso para el que no funcione ¡no quiero conocerlo!), vamos a escribir

la expresión regular de forma detallada antes de que se te olvide porqué elegiste las opciones que elegiste:

```
1  >>> phonePattern = re.compile(r'''
2                  # No busca el inicio, puede empezar en cualquier sitio
3      (\d{3}) # el código de área tiene tres dígitos (ej. '800')
4      \D*     # separador opcional
5      (\d{3}) # el troncal sin 3 dígitos (ej. '555')
6      \D*     # separador opcional
7      (\d{4}) # el resto del número: 4 dígitos (ej. '1212')
8      \D*     # separador opcional
9      (\d*)   # extensión opcional, cualquier número de dígitos
10     $       # fin de la cadena
11     ''', re.VERBOSE)
12 >>> phonePattern.search('work 1-(800) 555.1212 #1234').groups()
13 ('800', '555', '1212', '1234')
14 >>> phonePattern.search('800-555-1212')
15 ('800', '555', '1212', '')
```

1. *Línea 12:* Aparte de estar escrita en varias líneas, esta expresión es exactamente la misma que la del ejemplo anterior, por lo que analiza las mismas cadenas.

2. *Línea 14:* Validación final de seguridad. Sí, aún funciona. Has finalizado.

5.7. Resumen

Solamente has visto la punta más pequeña del iceberg de lo que las expresiones regulares pueden hacer. En otras palabras, aunque estés totalmente abrumado por ellas ahora mismo, creéme, no has visto casi nada de ellas aún.

Deberías estar familiarizado con las siguientes técnicas:

- \^ coincide con el comienzo de la cadena.
- \$ coincide con el final de la cadena.
- \b coincide con un límite de palabra.
- \d coincide con cualquier dígito numérico.
- \D coincide con cualquier carácter no numérico.
- x? coincide con un carácter x 0 a 1 veces.

- x* coincide con un carácter x 0 o más veces.
- x+ coincide con un carácter x 1 o más veces.
- x{n,m} coincide con carácter x entre n y m veces.
- (a—b—c) coincide con a o b o c.
- (x) en general es un *grupo a recordar*. Puedes obtener el valor de lo que ha coincidido mediante el método group() del objeto retornado por la llamada a re.search().

Las expresiones regulares son muy potentes pero no son la solución a todos los problemas. Deberías aprender a manejarlas de forma que sepas cuándo son apropiadas, cuándo pueden ayudarte a resolver un problema y cuándo te producirán más problemas que los que resuelven.

Capítulo 6

Cierres y generadores

Nivel de dificultad:◆ ◆ ◆ ◇ ◇

"Mi forma de deletrear es temblorosa.
Deletreo bien pero me tiembla la voz
así que las letras acaban en los lugares equivocados."
—Winnie the Pooh

6.1. Inmersión

Por razones que superan toda comprensión, siempre he estado fascinado por los lenguajes. No por los lenguajes de programación. Bueno sí, lenguajes de programación, pero también idiomas naturales. Toma como ejemplo el inglés, es un idioma esquizofrénico que toma prestadas palabras del Alemán, Francés, Español y latín (por decir algunos). En realidad "tomar prestado" es una frase equivocada, "saquea" sería más adecuada. O quizás "asimila" – como los Borg, sí eso me gusta:

> Somos los Borg. Añadiremos tus particularidades linguísticas y etimológicas a las nuestras. Es inútil que te resistas.

En este capítulo vas a aprender sobre nombres en plural. También sobre funciones que retornan otras funciones, expresiones regulares avanzadas y generadores. Pero primero hablemos sobre cómo crear nombres en plural. (Si no has leído el capítulo 5 sobre expresiones regulares ahora puede ser un buen momento; este capítulo asume una comprensión básica de ellas y rápidamente desciende a usos más avanzados).

Si has crecido en un país de habla inglesa o has aprendido inglés en el ámbito escolar, probablemente estés familiarizado con sus reglas básicas de formación del plural:

- Si una palabra termina en S, X o Z añádele ES. *Bass* se convierte en *basses*, *fax* en *faxes* y waltz en waltzes.
- Si una palabra termina en una H sonora, añade ES, si termina en una H muda, añade S. ¿Qué es una H sonora? una que se combina con otras letras para crear un sonido que se puede escuchar. Por eso *coach* se convierte en *coaches* y rash en rashes, porque se pronuncian tanto la CH como la SH. Pero *cheetah* se convierte en *cheetahs*, puesto que la H es muda.
- Si una palabra termina en una Y que suene como I, se cambia la Y por IES; si la Y se combina con una vocal para de otro modo, simplemente añade una S. Por eso *vacancy* se convierte en *vacancies* y day se convierte en days.
- Si todo lo demás falla, simplemente añadie uan S y cruza los dedos por que hayas acertado.

(Sé que hay muchas excepciones, *man* se convierte en *men*, *woman* en *women*, sin embargo *human* se convierte en *humans*. *Mouse* se convierte en *mice* y *louse* en *lice*, pero *house* se convierte en *houses*. *Knife* en *knives* y *wife* en *wives*, pero *lowlife* se convierte en *lowlifes*. Y todo ello sin meterme con las palabras que tienen su propio plural, como *sheep*, *deer* y *haiku*).

Otros idiomas, por supuesto, son completamente diferentes.

Vamos a diseñar una librería de Python que pase a plural los nombres ingleses de forma automática. Vamos a comenzar con estas cuatro reglas, pero recuerda que, inevitablemente, necesitarás añadir más.

6.2. Lo sé, ¡vamos a usar expresiones regulares!

Así que estás analizando palabras, lo que, al menos en inglés, significa que estás analizando cadenas de caracteres. Tienes reglas que requieren la existencida de diferentes combinaciones de caracteres, y después de encontrarlas necesitas hacerles modificaciones. ¡Esto parece un buen trabajo para las expresiones regulares!

```
1  import re
2
3  def plural(nombre):
4      if re.search('[sxz]$', nombre):
5          return re.sub('$', 'es', nombre)
6      elif re.search('[^aeioudgkprt]h$', nombre):
7          return re.sub('$', 'es', nombre)
8      elif re.search('[^aeiou]y$', nombre):
9          return re.sub('y$', 'ies', nombre)
10     else:
11         return nombre + 's'
```

1. *Línea 4:* Esto es una expresión regular que utiliza una sintaxis que no has visto en el capítulo dedicado a ellas. Los corchetes cuadrados indican que se encuentre exactamente uno de los caracteres encerrados entre ellos. Por eso [xyz] significa "o s o x o z, pero únicamente uno de ellos". El símbolo $ sí debería serte familiar, coincide con el final de la cadena. Al combinarlos esta expresión regular comprueba si la variable nombre finaliza con s, x o z.

2. *Línea 5:* La función re.sub() ejecuta una sustitución de cadena de texto basándose en una expresión regular.

Vamos a ver en detalle las sustituciones de texto utilizando expresiones regulares.

```
1  >>> import re
2  >>> re.search('[abc]', 'Mark')
3  <_sre.SRE_Match object at 0x001C1FA8>
4  >>> re.sub('[abc]', 'o', 'Mark')
5  'Mork'
6  >>> re.sub('[abc]', 'o', 'rock')
7  'rook'
8  >>> re.sub('[abc]', 'o', 'caps')
9  'oops'
```

1. *Línea 2:* La cadena Mark ¿contiene a, b o c? sí, contiene la a.

2. *Línea 4:* Ok, ahora vamos a buscar a, b o c y reemplazarlos por una o. Mark se convierte en Mork.

3. *Línea 6:* La misma función convierte rock en rook.

4. *Línea 8:* Podrías creer que caps se convierte en oaps, pero no lo hace. re.sub() reemplaza *todas* las coincidencias, no solamente la primera. Por eso, esta expresión regular convierte caps en oops, porque coinciden tanto la c como la a, así que ambas se convierten en o.

Y ahora volvamos a la función plural()...

```
1 def plural(nombre):
2     if re.search('[sxz]$', nombre):
3         return re.sub('$', 'es', nombre)
4     elif re.search('[^aeioudgkprt]h$', nombre):
5         return re.sub('$', 'es', nombre)
6     elif re.search('[^aeiou]y$', nombre):
7         return re.sub('y$', 'ies', nombre)
8     else:
9         return nombre + 's'
```

1. *Línea 3:* Aquí estás reemplazando el final de la cadena (que se encuentra con $) con la cadena es. En otras palabras, estás añadiendo es al final de la cadena. Podrías conseguir lo mismo con la concatenación de cadenas, por ejemplo nombre + 'es', pero he elegido utilizar expresiones regulares para cada regla, por razones que quedarán claras más adelante.

2. *Línea 4:* Mira atentamente, esta es otra variación nueva. El ^ en el primer carácter de los corchetes indica algo especial: negación. [âbc] significa "busca por un único carácter pero que sea cualquiera salvo a, b o c". Por eso [âeioudgkprt] significa que se busque por cualquier carácter salvo los indicados. Luego ese carácter deberá tener detrás una h y después de ella debe venir el final de la cadena. Estamos buscando por palabras que terminen en H sonoras.

3. *Línea 6:* Aquí seguimos el mismo patrón, busca palabras que terminen en Y, en las que delante de ella no exista una vocal. Estamos buscando por palabras que terminen en Y que suenen como I.

Veamos en detalle el uso de la negación en expresiones regulares.

```
1 >>> import re
2 >>> re.search('[^aeiou]y$', 'vacancy')
3 <_sre.SRE_Match object at 0x001C1FA8>
4 >>> re.search('[^aeiou]y$', 'boy')
5 >>>
6 >>> re.search('[^aeiou]y$', 'day')
7 >>>
8 >>> re.search('[^aeiou]y$', 'pita')
9 >>>
```

1. *Línea 2:* La palabra vacancy coincide con esta expresión regular, porque finaliza en cy, y la c no es una vocal.

2. *Línea 4:* La palabra boy no coincide porque finaliza en oy, y la expresión regular dice expresamente que delante de la y no puede haber una o. La palabra day tampoco coincide por una causa similar, puesto que termina en ay.

3. *Línea 8:* La palabra pita no coincide, puesto que no termina en y.

```
1 >>> re.sub('y$', 'ies', 'vacancy')
2 'vacancies'
3 >>> re.sub('y$', 'ies', 'agency')
4 'agencies'
5 >>> re.sub('([^aeiou])y$', r'\1ies', 'vacancy')
6 'vacancies'
```

1. *Línea 1:* Esta expresión regular convierte vacancy en vacancies y agency en agencies, que es lo que querías. Observa que también convierte boy en boies, pero eso no pasará porque antes habremos efectuado un re.search() para descubrir si debemos hacer la sustitución re.sub().

2. *Línea 5:* Aunque sea de pasada, me gustaría apuntar que es posible combinar las dos expresiones regulares en una única sentencia (la primera expresión regular para descubrir si se debe aplicar una regla y la segunda para aplicarla manteniendo el texto correcto). Se muestra como quedaría. La mayor parte te debe ser familiar del capítulo dedicado a las expresiones regulares. Utilizamos un grupo para recordar el carácter que se encuentra delante de la letra y. Luego, en la cadena de sustitución se utiliza una sintaxis nueva \1, que sirve para indicar que en ese punto se debe poner el valor del grupo guardado. En este ejemplo, el valor del grupo es la letra c de delante de la letra y; cuando se efectúa la sustitución, se sustituye la c en su mismo lugar, y los caracteres ies en el lugar de la y. (Si se hubiesen guardado más grupos, se podrían incluir con \2, \3 y así sucesivamente.

Las sustitución mediante el uso de expresiones regulares es un mecanismo muy potente, y la sintaxis
1 lo hace aún más. Pero combinar toda la operación en una única expresión regular la hace mucho más difícil de leer al no describir directamente la forma en la que se explicaron las reglas del plural. Estas reglas decían originalmente algo así como "si la palabra termina en S, X o Z, entonces añade ES". Si se echa un vistazo al código de la función, las dos líneas de código dicen casi exactemente eso mismo.

6.3. Una lista de funciones

Ahora vas a añadir un nuevo nivel de abstración. Comenzaste por definir una lista de reglas: si pasa esto, haz esto oto, en caso contrario ve a la siguiente regla. Vamos a complicar un poco parte del programa para poder simplificar otra parte.

```
1  import re
2
3  def match_sxz(noun):
4      return re.search('[sxz]$', noun)
5
6  def apply_sxz(noun):
7      return re.sub('$', 'es', noun)
8
9  def match_h(noun):
10     return re.search('[^aeioudgkprt]h$', noun)
11
12 def apply_h(noun):
13     return re.sub('$', 'es', noun)
14
15 def match_y(noun):
16     return re.search('[^aeiou]y$', noun)
17
18 def apply_y(noun):
19     return re.sub('y$', 'ies', noun)
20
21 def match_default(noun):
22     return True
23
24 def apply_default(noun):
25     return noun + 's'
26
27 rules = ((match_sxz, apply_sxz),
28          (match_h, apply_h),
29          (match_y, apply_y),
30          (match_default, apply_default)
31          )
32
33 def plural(noun):
34     for matches_rule, apply_rule in rules:
35         if matches_rule(noun):
36             return apply_rule(noun)
```

1. *Función* ***match_y****:* Ahora cada regla de búsqueda se convierte en una función que devuelve el resultado de la búsqueda: re.search().

2. *Función **apply_y**:* Cada regla de sustitución tine su propia función que llama a re.sub() para aplicar la regla apropiada.

3. *Línea 27:* En lugar de tener una función plural() con múltiples reglas, tenemos la estructura rules, que es una secuencia formada por parejas en la que cada una de ellas está formada, a su vez, por dos funciones.

4. *Línea 34:* Puesto que las reglas se han pasado las funciones contenidas en la estructura de datos, la nueva función plural() puede reducirse a unas cuantas líneas de código. Mediante el uso de un bucle for puedes obtener en cada paso las funciones de búsqueda y sustitución de la estructura de datos rules. En la primera iteración del bucle for la variable matches_rule referenciará a la función match_sxz y la variable apply_rule referenciará a la función apply_sxz. En la segunda iteración (asumiendo que alcanzas a ello), se le asigna match_h a matches_rule y apply_h a apply_rule. Está garantizado que la función retorna algo en todos los caso, porque la última regla de búsqueda (match_default) retorna True siempre, lo que significa que se aplicaría en última instancia la regla correspondiente (match_default).

La variable "rules" es una secuencia de pares de funciones.

La razón por la que esta técnica funciona es que "todo en Python es un objeto", funciones incluidas. La estructura de datos rules contiene funciones –nombres de función, sino objetos función reales. Cuando se asignan en el bucle for las variables matches_rule y apply_rule apuntan a funciones reales que puedes llamar. En la primera iteración del bucle for esto supone que se llama la función matches_sxz(noun) y si retorna una coincidencia se llama a la función apply_sxz(noun).

Si este nivel adicional de abstracción te resulta confuso, intenta verlo así. Este bucle for es equivalente a lo siguiente:

```
1 def plural(noun):
2     if match_sxz(noun):
3         return apply_sxz(noun)
4     if match_h(noun):
5         return apply_h(noun)
6     if match_y(noun):
7         return apply_y(noun)
8     if match_default(noun):
9         return apply_default(noun)
```

La ventaja es que ahora la función plural() es más simple. Toma una secuencia de reglas, definidas en otra parte y cuyo número puede variar, e itera a través de ella de forma genérica.

1. Obtiene una pareja: función de búsqueda - función de sustitución.
2. Comprueba si la función de búsqueda retorna True.
3. ¿Coincide? Entonces ejecuta la función de sustitución y devuelve el resultado.
4. ¿No coincide? Vuelve al paso uno.

La reglas pueden definirse en otra parte, de cualquier forma. A la función plural() no le importa.

¿Merecía la pena añadir este nivel de abstracción? Tal vez aún no te lo parezca. Vamos a considerar lo que supondría añadir una nueva regla. En el primer ejemplo, requeriría añadir una sentencia if a la función plural(). En este segundo ejemplo, requeriria añadir dos funciones match_algo() y apply_algo() y luego añadirlas a la secuencia rules para especificar en qué orden deben llamarse estas nuevas funciones en relación a las reglas que ya existían.

Pero realmente esto que hemos hecho es un hito en el camino hacia la siguiente sección. Sigamos...

6.4. Una lista de patrones

Realmente no es necesario definir una función para cada patrón de búsqueda y de sustitución. Nunca los llamas directamente; los añades a la secuencia de reglas (rules) y los llamas desde ahí. Es más, cada función sigue uno de los dos patrones. Todas las funciones de búqueda llaman a re.search() y todas las funciones de sustitución llaman a re.sub(). Vamos a sacar los patrones para que definir nuevas reglas sea más sencillo.

```
1 import re
2
3 def build_match_and_apply_functions(pattern, search, replace):
4     def matches_rule(word):
5         return re.search(pattern, word)
6     def apply_rule(word):
7         return re.sub(search, replace, word)
8     return (matches_rule, apply_rule)
```

1. *Línea 3:* build_match_and_apply_functions es una función que construye otras funciones dinámicamente. Toma los parámetros y define la función matches_rule() que llama a re.search() con el pattern que se haya pasado y el parámetro word que se pasará a la función cuando se llame en el futuro. ¡Vaya!

2. *Línea 6:* La construcción de la función de sustitución es similar. Es una función que toma un parámetro word y llama a re.sub() con él y los parámetros search y replace que se pasaron a la función constructora. Esta técnica de utilizar los valores de los parámetros exteriores a una función dentr ode una función dinámica se denomina *closures*[1]. En el fondo se están definiendo constantes que se utilizan dentro detro de la función que se está construyendo: la función construida toma un único parámetro (word) y los otros dos valores utilizados (search y replace) son los que tuvieran almacenados en el momento en que se definió la función.

3. *Línea 8:* Finalmente, la función retorna una tupla con las dos funciones recién creadas. Las constantes definidas dentro de esas funciones (pattern en la función match_rule(), y search y replace en la función apply_rule()) conservan los valores dentro de cada una de ellas, incluso después de finalizar la ejecución de la función constructora. Esto resulta ser muy práctico.

Si te resulta muy confuso (y debería, puesto que es algo bastante avanzado y extraño), puede quedarte más claro cuando veas cómo usarlo.

```
1 patterns = \
2   (
3     ('[sxz]$',           '$',  'es'),
4     ('[^aeioudgkprt]h$', '$',  'es'),
5     ('(qu|[^aeiou])y$',  'y$', 'ies'),
6     ('$',                '$',  's')
7   )
8 rules = [build_match_and_apply_functions(pattern, search, replace)
9          for (pattern, search, replace) in patterns]
```

1. *Línea 1:* Ahora las reglas se definen como una tupla de tuplas de cadenas (no son funciones). La primera cadena en cada trío es el patrón de búsqueda que se usará en re.search() para descubrir si una cadena coincide. La segunda y la tercera cadenas de cada grupo es la expresión de búsqueda y reemplazo que se utilizarán en re.sub() para modificar un nombre y ponerlo en plural.

2. *Línea 6:* Hay un ligero cambio aquí, en la regla por defecto. En el ejemplo anterior, la función match_default() retornaba True, dando a entender que si ninguna otra regla coincidía, el código simplemente debería añadir una s al final de la palabra. Este ejemplo hacer algo que es funcionalmente equivalente. La expresión regular final simplemente pregunta si la palabra tiene final ($ coincide con el final de la cadena). Desde luego todas las cadenas tienen final,

[1] Nota del Traductor: en español se utiliza la palabra "cierre" para referirse a este término.

incluso la cadena vacía, por lo que esta expresión siempre coincide. Así que sirve para el mismo propósito que la función match_default() del ejemplo anterior: asegura que si no coincide una regla más específica, el código añade una s al final de la palabra.

3. *Línea 8:* Esta línea es magia. Toma la secuencia de cadenas de patterns y la convierte en una secuencia de funciones. ¿Cómo? "mapeando" las cadenas con la función build_match_and_apply_functions. Toma un triplete de cadenas y llama a la función con las tres cadenas como argumentos. La función retorna una tupla de dos funciones. Esto significa que las variable rules acaba siendo equivalente a la del ejemplo anterior: una lista de tuplas, en la que cada una de ellas contiene un par de funciones. La primera función es la función de búsqueda que llama a re.search() y la segunda que es la función de sustitución que llama a re.sub().

Para finalizar esta versión del programa se muestra el punto de entrada al mismo, la función plural().

```
1 def plural(noun):
2     for matches_rule, apply_rule in rules:
3         if matches_rule(noun):
4             return apply_rule(noun)
```

Como la lista rules es igual que en el ejemplo anterior (realmente lo es), no deberías sorprenderte al ver que la función plural() no ha cambiado en nada. Es totalmente genérica; toma una lista de funciones de reglas y las llama en orden. No le importa cómo se han definido las reglas. En el ejemplo anterior, se definieron funciones separadas. Ahora se han creado funciones dinámicas al mapearlas con la función build_match_and_apply_functions a partir de una serie de cadenas de texto. No importa, la función plural() sigue funcionando igual.

6.5. Un fichero de patrones

Hemos eliminado todo el código duplicado y añadido suficientes abstracciones para que las reglas de formación de plurales del inglés queden definidas en una lista de cadenas. El siguiente paso lógico es extraer estas reglas y ponerlas en un fichero separado, en el que se puedan modificar de forma separada del código que las utiliza.

Primero vamos a crear el fichero de texto que contenga las reglas que necesitas. No vamos a crear estructuras de datos complejas, simplemente cadenas de texto separadas por espacios en blanco en tres columnas. Vamos a llamarlo plural4-rules.txt.

```
[sxz]$              $       es
[^aeioudgkprt]h$    $       es
[^aeiou]y$          y$      ies
$                   $       s
```

Ahora veamos cómo utilizar este fichero de reglas.

```
1  import re
2
3  def build_match_and_apply_functions(pattern, search, replace):
4      def matches_rule(word):
5          return re.search(pattern, word)
6      def apply_rule(word):
7          return re.sub(search, replace, word)
8      return (matches_rule, apply_rule)
9
10 rules = []
11 with open('plural4-rules.txt', encoding='utf-8') as pattern_file:
12     for line in pattern_file:
13         pattern, search, replace = line.split(None, 3)
14         rules.append(build_match_and_apply_functions(
15                 pattern, search, replace))
```

1. *Línea 3:* La función build_match_and_apply_functions no ha cambiado. Aún utilizamos los cierres para construir dos funciones dinámicas por cada llamada, que utilizan las variables definidas en la función externa.

2. *Línea 11:* La función global open() abre un fichero y devuelve un objeto fichero. En este caso, el fichero que estamos abriendo contiene las cadenas de texto que son los patrones de formación de los nombres en plural La sentencia with crea un *contexto*: cuando el bloque with termina, Python cerrará automáticamente el fichero, incluso si sucede una excepción dentro del bloque with. Lo verás con más detalle en el capítulo 11 dedicado a los ficheros.

3. *Línea 12:* La sentencia for lee los datos de un fichero abierto: una línea cada vez, y asigna el valor de dicha línea a la variable line. Lo veremos en mayor detalle en el capítulo 11 dedicado a los ficheros.

4. *Línea 13:* Cada línea del fichero contiene tres valores que están separados por espacios en blanco o tabuladores. Para obtenerlos, se utiliza el método de cadenas de texto split(). El primer parámetro del método split() es None, que significa que "trocea la cadena en cualquier espacio en blanco (tabuladores incluidos, sin distinción)". El segundo parámetro es 3, que signfica que "trocea la cadena hasta 3 veces, y luego deje el resto de la línea". Una línea como [sxy]$ $ es se trocea en la siguiente lista ['[sxy]$', '$', 'es'], lo que significa que

pattern contendrá el valor '[sxy]$', search el valor '$' y replace el valor 'es'. Como ves, esto hace mucho para tan poco código escrito.

5. *Línea 14:* Finlmente, pasas los valores pattern, search y replace a la función build_match_and_apply_functions, que retorna una tupla con las dos funciones creadas. Esta tupla se añade a la lista rules, por lo que, al finalizar, la lista rules contiene la lista de funciones de búsqueda y sustitución que la función plural() necesita.

6.6. Generadores

¿No sería estupendo tener una función genérica que fuese capaz de recuperar el fichero de reglas? Obtener las reglas, validar si hay coincidencia, aplicar la transformación apropiada y seguir con la siguiente regla. Esto es lo único que la función plural() tiene que hacer.

```
1  def rules(rules_filename):
2      with open(rules_filename, encoding='utf-8') as pattern_file:
3          for line in pattern_file:
4              pattern, search, replace = line.split(None, 3)
5              yield build_match_and_apply_functions(pattern, search, replace)
6
7  def plural(noun, rules_filename='plural5-rules.txt'):
8      for matches_rule, apply_rule in rules(rules_filename):
9          if matches_rule(noun):
10             return apply_rule(noun)
11     raise ValueError('no matching rule for {0}'.format(noun))
```

¿Cómo funciona este código? Vamos a verlo primero con un ejemplo interactivo.

```
1  >>> def make_counter(x):
2  ...     print('entrando en  make_counter')
3  ...     while True:
4  ...         yield x
5  ...         print('incrementando x')
6  ...         x = x + 1
7  ...
8  >>> counter = make_counter(2)
9  >>> counter
10 <generator object at 0x001C9C10>
11 >>> next(counter)
12 entrando en make_counter
13 2
14 >>> next(counter)
15 incrementando x
16 3
17 >>> next(counter)
18 incrementando x
19 4
```

1. *Línea 4:* La presencia de la sentencia yield en la función make_counter significa que ésta no es una función "normal". Al llamarla lo que sucede es que se retorna un *generador* que puede utilizarse para generar sucesivos valores de x.

2. *Línea 8:* Para crear una instancia del generador make_counter simplemente hay que llamarlo como a cualquier otra función. Observa que esto en realidad no ejecuta el código de la función. Lo puedes comprobar porque la primera línea de la función make_counter() llama a la función print() y en esta llamada no se ha imprimido nada en pantalla.

3. *Línea 9:* La función make_counter() retorna un objeto generador.

4. *Línea 11:* El método next() del generador retorna su siguiente valor. La primera vez que ejecutas next() se ejecuta el código de la función make_counter() hasta la primera sentencia yield que se ejecute, y se retorna el valor que aparece en la sentencia yield[2]. En este caso, el valor será 2, porque originalmente se creó el generador con la llamdada make_counter(2).

5. *Línea 14:* Al llamar repetidas veces al método next() del mismo objeto generador, la ejecución del objeto se reinicia en el mismo lugar en el que se quedó (después del yield anterior) y continua hasta que vuelve a encontrar

[2]N.del T.: En español "yield" puede traducirse como "ceder". En Python es como si al llegar la ejecución de esta sentencia, se cediera el paso devolviendo el valor que aparezca como si fuese un return pero sin terminar la función. La siguiente vez que se ejecute el next() continuará por el lugar en el que se cedío el paso.

otra sentencia yield. Todas las variables y, en general, el estado local de la función queda almacenado al llegar a la sentencia yield y se recupera al llamar de nuevo a la función next(). La siguiente línea de código que se ejecuta llama a un print() que muestra incrementando x. Después de eso se ejecuta la sentencia x = x + 1. Luego se ejecuta el bucle while y lo primero que sucede es que se vuelve a ejecutar yield x, lo que salva el estado en la situación actual y devuelve el valor que tiene x (que ahora es 3).

6. *Línea 17:* La siguiente vez que llamas a next(counter), sucede lo mismo de nuevo, pero ahora el valor de x es 4.

Puesto que make_counter() establece un bucle infinito, teóricamente se puede ejecutar infinitas veces la llamada al método next(), y se mantendría el incremento de x y la impresión de sus sucesivos valores. Pero vamos a ser un poco más productivos en el uso de los generadores.

6.7. Un generador de la serie de Fibonacci

> "yield" pausa la función, "next()" continúa desde donde se quedó.

```
1 def fib(max):
2     a, b = 0, 1
3     while a < max:
4         yield a
5         a, b = b, a + b
```

1. *Línea 2:* La serie de Fibonacci es una secuencia números en la que cada uno de los números es la suma de los dos números anteriores de la serie. Comienza en 0 y 1. Va subiendo lentamente al principio, y luego más rápidamente. Para comenzar la secuencia necesitas dos variables: a comienza valiendo 0 y b comienza en 1.

2. *Línea 4:* a es el número actual de la secuencia, por lo que se puede ceder el control y retornar dicho valor.

3. *Línea 5:* b es el siguiente número de la secuencia, por lo que hay que asignarlo a a, pero también hay que calcular antes el nuevo valor siguiente (a + b) y asignarlo a b para su uso posterior. Observa que esto sucede en paralelo; si a es 3 y b es 5, entonces a, b = b, a + b hará que a valga 5 (el valor previo de b) y b valdrá 8 (la suma de los valores previos de a y b).

De este modo tienes una función que va retornando los sucesivos números de la serie de Fibonacci. Es evidente que también podrías hacer esto con recursión, pero de este modo es más fácil de leer. Además, así también es más fácil trabajar con los bucles for.

```
1 >>> from fibonacci import fib
2 >>> for n in fib(1000):
3 ...     print(n, end=' ')
4 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
5 >>> list(fib(1000))
6 [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987]
```

1. *Línea 2:* Los generadores, como fib(), se pueden utilizar directamente en los bucles for. El bucle for llamará automáticamente a la función next() para recuperar los valores del generador fib() y lo asignará a la variable del índice (n).

2. *Línea 3:* Cada vez que se pasa por el bucle for, n guarda un nuevo valor obtenido de la sentencia yield de la función fib(), y todo lo que hay que hacer en el bucle es imprimirlo. Una vez la función fib() se queda sin números (a se hace mayor que el valor max, que en este caso es 1000), el bucle for termina sin problemas.

3. *Línea 5:* Esta es una forma idiomática de Python muy útil: pasar un generador a la función list() que iterará a través de todo el generador (como en el bucle for del ejemplo anterior) y retorna la lista con todos los valores.

6.8. Un generador de reglas de plurales

Volvamos al código con las reglas de formación del plural para ver como funciona plural().

```
1  def rules(rules_filename):
2      with open(rules_filename, encoding='utf-8') as pattern_file:
3          for line in pattern_file:
4              pattern, search, replace = line.split(None, 3)
5              yield build_match_and_apply_functions(pattern, search, replace)
6
7  def plural(noun, rules_filename='plural5-rules.txt'):
8      for matches_rule, apply_rule in rules(rules_filename):
9          if matches_rule(noun):
10             return apply_rule(noun)
11     raise ValueError('no matching rule for {0}'.format(noun))
```

1. *Línea 4:* No hay ninguna magia aquí. Recuerda que las líneas del fichero de reglas tienen tres valores separados por espacios en blanco, por lo que utilizas line.split(None, 3) para retornar las tres "columnas" y asignarlas a las tres variables locales.

2. *Línea 5: Y lo siguiente es un yield.* ¿Qué es lo que se cede? las dos funciones que se construyen dinámicamente con la ya conocida build_match_and_apply_functions, que es idéntica a los ejemplos anteriores. En otras palabras, rules() es un generador que va retornando funciones de búsqueda y sustitución *bajo demanda.*

3. *Línea 8:* Puesto que rules() es un generador, puedes utilizarlo directamente en un bucle for. La primera vez, el bucle for llama a la función rules(), que abre el fichero de patrones, lee la primera línea, construye de forma dinámica las funciones de búsqueda y sustitución de los patrones de esa línea y cede el control devolviendo ambas funciones. La segunda vuelta en el bucle for continúa en el lugar exacto en el que se cedió el control por parte del generador rules(). Lo primero que hará es leer la siguiente línea del fichero (que continúa abierto), construirá dinámicamente otras dos funciones de búsqueda y sustitución basadas en los patrones de esa línea del fichero y, de nuevo, cederá el control devolviendo las dos nuevas funciones.

¿Qué es lo que hemos ganado con respecto de la versión anterior? Tiempo de inicio. En la versión anterior, cuando se importaba el módulo, se leía el fichero completo y se construía una lista con todas las reglas posibles. Todo ello, antes de que se pudiera ejecuta la función plural(). Con los generadores, puedes hacero todo de forma "perezosa": primero lees la primera regla, creas las funciones y las pruebas, si con ello se pasa el nombre a plural, no es necesario seguir leyendo el resto del fichero, ni crear otras funciones.

¿Qué se pierde? ¡Rendimiento! Cada vez que llamas a la función plural(), el generador rules() vuelve a iniciarse desde el principio (se genera un nuevo objeto cada vez que se llama a la función generador rules()) —lo que significa que se reabre el fichero de patrones y se lee desde el comienzo, una línea cada vez.

¿Cómo podríamos tener lo mejor de los dos mundos?: coste mínimo de inicio (sin ejecutar ningún código en el import), y máximo rendimiento (sin construir las mismas funciones una y otra vez). ¡Ah! y todo ello manteniendo las reglas en un fichero separado (porque el código es código y los datos son datos) de forma que no haya que leer la misma línea del fichero dos veces.

Para hacer eso, necesitas construir un *iterador*. Pero antes de hacerlo, es necesario que aprendas a manejar las clases de objetos en Python.

6.9. Lecturas recomendadas

- Generadores simples (PEP 255): http://www.python.org/dev/peps/pep-0255/
- Comprendiendo la sentencia "with" de Python: http://effbot.org/zone/python-with-statement.htm
- Cierres en Python: http://ynniv.com/blog/2007/08/closures-in-python.html
- Números de Fibonacci: http://en.wikipedia.org/wiki/Fibonacci_number
- Nombres plurales irregulares en inglés: http://www2.gsu.edu/~wwwesl/egw/crump.htm

Capítulo 7

Clases e iteradores

Nivel de dificultad:◆ ◆ ◆ ◇ ◇

"El Este está al Este y el Oeste al Oeste,
y nunca ambos se encontrarán."
—Ruyard Kipling

7.1. Inmersión

Los generadores son únicamente un caso especial de iteradores. Una función que entrega valores es una forma buena de crear un iterador sin llegar a crearlo. Déjame enseñarte lo que quiero decir.

¿Recuerdas el generador de la serie de Fibonacci? Aquí lo tienes construido como un iterador:

```
class Fib:
    '''iterador que genera los números de la secuencia de Fibonacci'''

    def __init__(self, max):
        self.max = max

    def __iter__(self):
        self.a = 0
        self.b = 1
        return self

    def __next__(self):
        fib = self.a
        if fib > self.max:
            raise StopIteration
        self.a, self.b = self.b, self.a + self.b
        return fib
```

Vamos a ver una línea cada vez.

```
class Fib:
```

¿class? ¿Qué es una clase?

7.2. Cómo se definen clases

Python es un lenguaje orientado a objetos: Puedes definir tus propias clases de objetos, heredar de tus clases o de las preexistentes y crear instancias de las clases que has definido.

Es sencillo definir una clase en Python. Como con las funciones no existe una definición del interface separada. Simplemente define la clase y el código. Una clase en Python comienza con la palabra reservada class, seguida del nombre de la clase. Técnicamente es todo lo que se necesita puesto que no necesita heredar de ninguna otra.

```
class PapayaWhip:
    pass
```

1. *Línea 1:* El nombre de esta clase es PapayaWhip y no hereda de ninguna otra. Los nombres de las clases se suelen escribir con el primer carácter en mayúsculas, CadaPalabraDeEstaForma pero esto es únicamente por convención, no es un requisito obligatorio.

2. *Línea 2:* Probablemente lo hasta adivinado, pero el contenido de la clase

está siempre indentado, como pasa con el código de una función, una sentencia if, un bucle for o cualquier otro bloque de código. La primera línea no indentada indica el final de la clase y se encuentra fuera de la misma.

Esta clase PapayaWhip no define ningún método o atributos, pero es correcta sintácticamente. Como necesita que exista algo en el contenido de la clase se escribe la sentencia pass. Esta palabra reservada de Python significa únicamente "sigue adelante, no hay nada que hacer aquí". Es una palabra reservada que no hace nada y, por ello, una buena forma de marcar un sitio cuando tienes funciones o clases a medio escribir.

La sentencia pass de Python es como una pareja vacía de llaves ({}) en Java o C.

Muchas clases heredan de otras, pero esta no. Muchas clases definen métodos, pero esta no. No hay nada que tenga que tener obligatoriamente una clase de Python, salvo el nombre. En particular, los programadores de C++ pueden encontrar extraño que las clases de Python no tengan que tener constructores y destructores explicitos. Aunque no es obligatorio, las clases de Python pueden tener algo parecido a un constructor: el método __init__().

7.2.1. El método __init__()

Este ejemplo muestra la inicializacion de la clase Fib utilizando el método __init__().

```
1 class Fib:
2     '''iterador que genera los números de la secuencia de Fibonacci'''
3
4     def __init__(self, max):
```

1. *Línea 2:* Las clases pueden (y deberían) tener docstrings, tal y como sucede con los módulos y funciones.

2. *Línea 4:* El método __init__() se llama de forma automática por Python inmediatamente después de que se haya creado una instancia de la clase. Es tentador —pero técnicamente incorrecto— llamar a este método el "constructor" de la clase. Es tentador, porque recuerda a los constructores de C++ (por convención, el método __init__() se suele poner como el primer método de la clase), e incluso suena como uno. Es incorrecto, porque cuando se llama a este método en Python, el objeto ya ha sido construido y ya dispones de una referencia a una instancia válida de la clase (self).

El primer parámetro de todo método de una clase, incluyendo el método __init__(), siempre es una referencia al objeto actual, a la instancia actual de la clase. Por convención, este parámetro se suele llamar self. Este parámetro ocupa el rol de la palabra reservada this de C++ o Java, pero self no es una palabra reservada en Python, es simplemente una convención en para el nombre del primer parámetro de los métodos de una clase. En cualquier caso, por favor no lo llames de otra forma que no sea self; esta convención es muy fuerte y todo el mundo la usa.

En el método __init__(), self se refiere al objeto recién creado; en otros métodos de la clase se refiere a la instancia cuyo método se llamó. Aunque necesitas especificar self explícitamente cuando defines el método, no lo especificas cuando se llama. Python lo hace por ti automáticamente.

7.3. Instanciar clases

Instanciar clases en Python es inmediato. Para crear un objeto de la clase, simplemente llama a la clase como si fuese una función, pasándole los parámetros que requiera el método __init__(). El valor de retorno será el nuevo objeto.

```
1 >>> import fibonacci2
2 >>> fib = fibonacci2.Fib(100)
3 >>> fib
4 <fibonacci2.Fib object at 0x00DB8810>
5 >>> fib.__class__
6 <class 'fibonacci2.Fib'>
7 >>> fib.__doc__
8 '''iterador que genera los números de la secuencia de Fibonacci'''
```

1. *Línea 2:* Se crea una instancia de la clase Fib (definida en el módulo fibonacci2) y se asigna la instancia creada a la variable fib. Se pasa un parámetro que será el parámetro max del método __init__() de la clase Fib.

2. *Línea 3:* La variable fib se refiere ahora a un objeto que es instancia de la clase Fib.

3. *Línea 5:* Toda instancia de una clase tiene el atributo interno __class__ que es la clase del objeto. Muchos programadores java estarán familiarizados con la clase Class, que contiene métodos como getName() y getSuperClass() para conseguir información de metadatos del objeto. En Python, esta clase de metadados está disponible mediante el uso de atributos, pero la idea es la misma.

4. *Línea 7:* Puedes acceder al docstring de la instancia como se hace con cualquier

otro módulo o función. Todos los objetos que son instancia de una misma clase comparten el mismo docstring

En Python, basta con llamar a una clase como si fuese una función para crear un nuevo objeto de la clase. No existe ningún operador new como sucede en C++ o Java.

7.4. Variables de las instancias

En el siguiente código:

```
1 class Fib:
2     def __init__(self, max):
3         self.max = max
```

1. *Línea 3:* ¿Qué es self.max? Es una variable de la instancia. Completamente diferente al parámetro max, que se pasa como parámetro del método. self.max es una variable del objeto creado. Lo que significa que puedes acceder a ella desde otros métodos.

```
1 class Fib:
2     def __init__(self, max):
3         self.max = max
4     .
5     .
6     .
7     def __next__(self):
8         fib = self.a
9         if fib > self.max:
```

1. *Línea 3:* self.max se crea en el método __init__(), por ser el primero que se llama.
2. *Línea 9:* ...y se utiliza en el método __next__().

Las variables de instancia son específicas de cada objeto de la clase. Por ejemplo, si creas dos objetos Fib con diferentes valores máximos cada uno de ellos recordará sus propios valores.

```
>>> import fibonacci2
>>> fib1 = fibonacci2.Fib(100)
>>> fib2 = fibonacci2.Fib(200)
>>> fib1.max
100
>>> fib2.max
200
```

7.5. Un iterador para la serie de Fibonacci

Ahora estás preparado para aprender cómo construir un iterador. Un iterador es una clase que define el método __iter__().

> Los tres métodos de clase, __init__, __iter__ y __next__, comienzan y terminan con un par de guiones bajos (_). ¿Porqué? No es nada mágico, pero habitualmente significa que son métodos "especiales". Lo único que tienen en "especial" estos métodos especiales es que no se llaman directamente; Python los llama cuando utilizas otra sintaxis en la clase o en una instancia de la clase.

```
class Fib:
    def __init__(self, max):
        self.max = max

    def __iter__(self):
        self.a = 0
        self.b = 1
        return self

    def __next__(self):
        fib = self.a
        if fib > self.max:
            raise StopIteration
        self.a, self.b = self.b, self.a + self.b
        return fib
```

1. *Línea 1:* Para poder construir un iterador desde cero fib necesita ser una clase, no una función.

2. *Línea 2:* Al llamar a Fib(max) se está creando un objeto que es instancia de esta clase y llamándose a su método __init__() con el parámetro max. El método __init__() guarda el valor máximo como una variable del objeto de forma que los otros métodos de la instancia puedan utilizarlo más tarde.

3. *Línea 5:* El método __iter__() se llama siempre que alguien llama a iter(fib) (Como verás en un minuo, el bucle for llamará a este método automáticamente, pero tú también puedes llamarlo manualmente). Después de efectuar la inicialización necesaria de comienzo de la iteración (en este caso inicializar self.a y self.b) el método __iter__() puede retornar cualquier objeto que implemente el método __next__(). En este caso (y en la mayoría), __iter__() se limita a devolver self, puesto que la propia clase implementa el método __next__().

4. *Línea 10:* El método __next__() se llama siempre que alguien llame al método next() sobre un iterador de una instancia de una clase. Esto adquirirá todo su sentido en un minuto.

5. *Línea 13:* Cuando el método __next__() lanza la excepción StopIteration le está indicando a quién lo haya llamado que el iterador se ha agotado. Al contrario que la mayoría de las excepciones, no se trata de un error. es una condición normal que simplemente significa que no quedan más valores que generar. Si el llamante es un bucle for se dará cuenta de que se ha elevado esta excepción y finalizará el bucle sin que se produzca ningún error (En otras palabrás, se tragará la excepción). Esta pequeña magia es el secreto clave del uso de iteradores en los bucles for.

6. *Línea 15:* Para devolver el siguiente valor del iterador, el método __next__() simplemente utiliza return para devolver el valor. No se utiliza yield, que únicamente se utiliza para los generadores. Cuando se está creando un iterador desde cero, se utiliza return en el método __next__().

¿Estás ya totalmente confundido? Excelente. Veamos cómo utilizar el iterador.

```
1 >>> from fibonacci2 import Fib
2 >>> for n in Fib(1000):
3 ...     print(n, end=' ')
4 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

¡Exactamente igual que en el ejemplo del generador! Es idéntico a como usabas el generador. Pero... ¿cómo?.

Existe algo de trabajo ejecutándose en los bucles for.

- El bucle for llama a Fib(1000), que retorna un objeto que es instancia de la clase Fib. Llamémoslo fib_inst.
- En secreto, pero de forma inteligente, el bucle for llama a iter(fib_inst), que retorna un objeto iterador. Vamos a llamar a este objeto fib_iter. En este caso, fib_iter == fib_inst, porque el método fib_inst.__iter__() retorna self, pero el bucle for no lo sabe, ni le importa.

- Para recorrer el bucle a través del iterador, el bucle for llama a next(fib_iter), que, a su vez, llama a fib_iter.__next__(), el método __next__() del objeto fib_iter, que realiza los cálculos necesarios y devuelve el siguiente elemento de la serie de fibonacci. El bucle for toma este valor y lo asigna a n, luego ejecuta el cuerpo del bucle para ese valor de n.

- ¿Cómo sabe el bucle for cuando parar? ¡Me alegro de que lo preguntes! Cuando next(fib_iter) eleva la excepción StopIteration el bucle for la captura finaliza sin error (Cualquier otra excepción se elevaría normalmente). ¿Y dónde has visto que se lanze esta excepción StopIteration? En el método __next__() ¡Por supuesto!

7.6. Un iterador para reglas de formación de plurales

Ahora es el final. Vamos a reescribir el generador de reglas de formación de plural como un iterador.

> iter(f) llama a f.__iter__(). next(f) llama a f.__next__().

```
1  class LazyRules:
2      rules_filename = 'plural6-rules.txt'
3
4      def __init__(self):
5          self.pattern_file = open(self.rules_filename, encoding='utf-8')
6          self.cache = []
7
8      def __iter__(self):
9          self.cache_index = 0
10         return self
11
12     def __next__(self):
13         self.cache_index += 1
14         if len(self.cache) >= self.cache_index:
15             return self.cache[self.cache_index - 1]
16
17         if self.pattern_file.closed:
18             raise StopIteration
19
20         line = self.pattern_file.readline()
21         if not line:
22             self.pattern_file.close()
23             raise StopIteration
24
25         pattern, search, replace = line.split(None, 3)
26         funcs = build_match_and_apply_functions(
27             pattern, search, replace)
28         self.cache.append(funcs)
29         return funcs
30
31 rules = LazyRules()
```

Como esta clase implementa los métodos __iter__() y __next__() puede utilizarse como un iterador. Al final del código se crea una instancia de la clase y se asigna a la variable rules.

Vamos a ver la clase poco a poco.

```
1  class LazyRules:
2      rules_filename = 'plural6-rules.txt'
3
4      def __init__(self):
5          self.pattern_file = open(self.rules_filename, encoding='utf-8')
6          self.cache = []
```

1. *Línea 5:* Cuando instanciamos la clase LazyRules, se abre el fichero de patrones pero no se lee nada de él (Eso se hace más tarde).

2. *Línea 6:* Después de abrir el fichero de patrones se inicializa la caché. Utilizarás la caché más tarde (en el método __next__()) según se lean las filas del fichero de patrones.

Antes de continuar vamos a echarle un vistazo a rules_filename. No está definida en el método __iter__(). De hecho no está definida dentro de *ningún* método. Está definida al nivel de la clase. Es una *variable de clase* y, aunque puedes acceder a ella igual que a cualquier variable de instancia (self.rules_filename), es compartida en todas las instancias de la clase LazyRules.

```
1  >>> import plural6
2  >>> r1 = plural6.LazyRules()
3  >>> r2 = plural6.LazyRules()
4  >>> r1.rules_filename
5  'plural6-rules.txt'
6  >>> r2.rules_filename
7  'plural6-rules.txt'
8  >>> r2.rules_filename = 'r2-override.txt'
9  >>> r2.rules_filename
10 'r2-override.txt'
11 >>> r1.rules_filename
12 'plural6-rules.txt'
13 >>> r2.__class__.rules_filename
14 'plural6-rules.txt'
15 >>> r2.__class__.rules_filename = 'papayawhip.txt'
16 >>> r1.rules_filename
17 'papayawhip.txt'
18 >>> r2.rules_filename
19 'r2-overridetxt'
```

1. *Línea 4:* Cada instancia de la clase hereda el atributo rules_filename con el valor definido para la clase.

2. *Línea 8:* La modificación del valor de la variable en una instancia no afecta al valor de las otras instancias...

3. *Línea 13:* ...ni cambia el atributo de la clase. Puedes acceder al atributo de la clase (por oposición al atributo de la instancia individual) mediante el uso del atributo especial __class__ que accede a la clase.

4. *Línea 15:* Si modificas el atributo de la clase, todas las instancias que heredan ese atributo (como r1 en este ejemplo) se verán afectadas.

5. *Línea 18:* Todas las instancias que han modificado ese atributo, sustituyendo su valor (como r2 aquí) no se verán afectadas.

Y ahora volvamos a nuestro espectáculo.

```
1  def __iter__(self):
2          self.cache_index = 0
3          return self
```

1. *Línea 1:* El método __iter__() se llamará cada vez que alguien —digamos un bucle for— llame a iter(rules).

2. *Línea 3:* Todo método __iter__() debe devolver un iterador. En este caso devuelve self puesto que esta clase define un método __next__() que será responsable de devolver los diferentes valores durante las iteraciones.

```
1  def __next__(self):
2          .
3          .
4          .
5          pattern, search, replace = line.split(None, 3)
6          funcs = build_match_and_apply_functions(
7              pattern, search, replace)
8          self.cache.append(funcs)
9          return funcs
```

1. *Línea 1:* El método __next__() se llama cuando alguien —digamos que un bucle for— llama a next(rules). La mejor forma de explicar este método es comenzando del final hacia atrás. Por lo que vamos a hacer eso.

2. *Línea 6:* La última parte de esta función debería serte familiar. La función build_match_and_apply_functions() no ha cambiado; es igual que siempre.

3. *Línea 8:* La única diferencia es que, antes de retornar el valor (que se almacena en la tupla funcs), vamos a salvarlas en self.cache.

Sigamos viendo la parte anterior...

```
1  def __next__(self):
2          .
3          .
4          .
5          line = self.pattern_file.readline()
6          if not line:
7              self.pattern_file.close()
8              raise StopIteration
9          .
10         .
11         .
```

1. *Línea 5:* Veamos una técnica avanzada de acceso a ficheros. El método readline() (nota: singular, no el plural readlines()) que lee exactamente una línea de un fichero abierto. Específicamente, la siguiente línea (Los objetos *fichero* también son iteradores...).

2. *Línea 6:* Si el método readline() lee algo (quedaban líneas por leer en el fichero), la variable line no será vacía. Incluso si la línea fuese vacía, la variable contendría una cadena de un carácter '
n' (el retorno de carro). Si la variable line es realmente una cadena vacía significará que no quedan líneas por leer en el fichero.

3. *Línea 8:* Cuando alcanzamos el final del fichero deberíamos cerrarlo y elevar la excepción mágica StopIteration. Recuerda que llegamos a esta parte de la función porque necesitamos encontrar una función de búsqueda y otra de sustitución para la siguiente regla. La siguiente regla tiene que venir en la siguiente línea del fichero... ¡pero si no hay línea siguiente! Entonces, no tenemos que retornar ningún valor. Las iteraciones han terminado (Se acabó la fiesta...).

Si seguimos moviéndonos hacia el comienzo del método __next__()...

```
1     def __next__(self):
2         self.cache_index += 1
3         if len(self.cache) >= self.cache_index:
4             return self.cache[self.cache_index - 1]
5
6         if self.pattern_file.closed:
7             raise StopIteration
8         .
9         .
10        .
```

1. *Línea 4:* self.cache contendrá una lista con las funciones que necesitamos para buscar y aplicar las diferentes reglas (¡Al menos esto te debería resultar familiar!). self.cache_index mantiene el índice del elemento de la caché que se debe retornar. Si no hemos consumido toda la caché (si la longitud de self.cache es mayor que self.cache_index), ¡tenemos un elemento en la caché para retornar! Podemos devolver las funciones de búsqueda y sustitución de la caché en lugar de construirlas desde cero.

2. *Línea 7:* Por otra parte, si no obtenemos un elemento de la caché y el fichero se ha cerrado (lo que podrá haber sucedido en la parte de más abajo del método, como se vio anteriormente) entonces ya no hay nada más que hacer. Si el fichero se ha cerrado, significa que lo hemos leido completamente —ya

hemos recorrido todas las líneas del fichero de patrones y hemos construido y cacheado las funciones de búsqueda y sustitución de cada patrón. El fichero se ha agotado y la caché también, ¡Uff! ¡yo también estoy agotado! Espera un momento, casi hemos terminado.

Si lo ponemos todo junto esto es lo que sucede cuando:

- Cuando el módulo es importado crea una única instancia de la clase LazyRules, que denominamos rules, que abre el fichero de patrones pero no lo lee.
- Cuando pedimos la primera pareja de funciones de búsqueda y sustitución, busca en la caché pero está vacía. Por lo que lee una línea del fichero de patrones, construye la pareja de funciones de búsqueda y sustitución para ellas y las guarda en la caché (además de retornarlas).
- Digamos, por simplicidad, que la primera regla coincidió con la búsqueda. Si es así no se busca ndad más y no se lee nada más del fichero de patrones.
- Además, por continuar con el argumento, supón que el programa que está usando este objeto llama a la función plural() de nuevo para pasar al plural una palabra diferente. El bucle for de la función plural() llamará a la función iter(rules), que resetea el índice de la caché pero no resetea el fichero abierto.
- La primera vez en esta nueva iteración, el bucle for pedirá el valor de rules, que llama al método __next__(). Esta vez, sin embargo, la caché tiene ya una pareja de funciones de búsqueda y sustitución, la correspondiente a los patrones de la primera línea del fichero de patrones, puesto que fueron construidas y cacheadas al generar el plural de la palabra anterior y por eso están en la caché. El índice de la caché se incrementa y no se toca el fichero abierto.
- Vamos a decir, en aras de continuar el argumento, que esta vez la primera regla no coincide, por lo que el bucle for da otra vuelta y pide otro valor de la variable rules. Por lo que se invoca por segunda vez al método __next__(). Esta vez la caché está agotada —solamente contenía un elemento, y estamos solicitando un segundo— por lo que el método __next__() continúa. Se lee otra línea del fichero abierto, se construyen las funciones de búsqueda y sustitución de los patrones y se introducen en la caché.
- Este proceso de lectura, construcción y caché continua mientras las reglas del fichero no coincidan con la palabra que estamos intentando poner en plural. Si se llega a encontrar una coincidencia antes del final del fichero, se utiliza y termina, con el fichero aún abierto. El puntero del fichero permanecerá dondequiera que se parase la lectura, a la espera de la siguiente sentencia readline().

Mientras tanto, la caché ha ido ocupándose con más elementos y si se volviera a intentar poner en plural a otra palabra, se probará primero con los elementos de la caché antes de intentar leer la siguiente línea del fichero de patrones.

Hemos alcanzado el nirvana de la generación de plurales.

1. **Coste de inicio mínimo.** Lo único que se hace al realizar el `import` es instanciar un objeto de una clase y abrir un fichero (pero sin leer de él).
2. **Máximo rendimiento.** El ejemplo anterior leería el fichero cada vez que hubiera que poner en plural una palabra. Esta versión cachea las funciones según se van construyendo y, en el peor caso, solamente leerá del fichero de patrones una única vez, no importa cuantas palabras pongas en plural.
3. **Separación de código y datos.** Todos los patrones se almacenan en un fichero separado. El código es código y los datos son datos y nunca se deberán de encontrar.

¿Es realmente el nirvana? Bueno, sí y no. Hay algo que hay que tener en cuenta con el ejemplo de `LazyRules`: El fichero de patrones se abre en el método `__init__()` y permanece abierto hasta que se alcanza la regla final. Python cerrará el fichero cuando se acabe la ejecución, o después de que la última instancia de la clase `LazyRules` sea destruida, pero eso puede ser *mucho* tiempo. Si esta clase es parte de un proceso de larga duración, el intérprete de Python puede que no acabe nunca y el objeto `LazyRules` puede que nunca sea destruido.

Hay formas de evitar esto. En lugar de abrir el fichero durante el método `__init__()` y dejarlo abierto mientras lees las reglas una línea cada vez, podrías abrir el fichero, leer todas las reglas y cerrarlo inmediatamente. O podrías abrir el fichero, leer una regla, guardar la posición del fichero con el método `tell()`, cerrar el fichero y, más tarde, reabrirlo y utilizar el método `seek()` para continuar leyendo donde lo dejaste. O podrías no preocuparte de dejar el fichero abierto, como pasa en este ejemplo. Programar es diseñar, y diseñar es siempre una continua elección entre decisiones que presentan ventajas e inconvenientes. Dejar un fichero abierto demasiado tiempo puede suponer un problema; hacer el código demasiado complejo podría ser un problema. Cuál es el problema mayor depende del equipo de desarrollo, la aplicación y tu entorno de ejecución.

7.7. Lecturas recomendadas

- Tipos de iteradores: http://docs.python.org/3.1/library/stdtypes.html#iterator-types
- PEP 234: Iteradores: http://www.python.org/dev/peps/pep-0234/
- PEP 255: Generadores simples: http://www.python.org/dev/peps/pep-0255/
- Técnicas de generadores para programadores de sistemas: http://www.dabeaz.com/generators/

Capítulo 8

Iteradores avanzados

Nivel de dificultad:◆ ◆ ◆ ◆ ◇

"Las pulgas grandes tienen pulgas más pequeñas sobre sus espaldas que les pican y las pulgas pequeñas tienen pulgas aún más pequeñas, y así hasta el infinito."
—August de Morgan

8.1. Inmersión

Hawaii + idaho + iowa + ohio == states. O, por ponerlo de otra manera, 510199 + 98153 + 9301 + 3593 == 621246. ¿Estoy hablando en clave? No, solamente es un rompecabezas.

Déjame aclarártelo.

```
1  HAWAII + IDAHO + IOWA + OHIO == STATES
2  510199 + 98153 + 9301 + 3593 == 621246
3
4  H = 5
5  A = 1
6  W = 0
7  I = 9
8  D = 8
9  O = 3
10 S = 6
11 T = 2
12 E = 4
```

Las letras forman palabras existentes, pero si sustituyes cada palabra por un dígito del 0 a 9, también forman una equación aritmética. El truco consiste en

descubrir cómo se emparejan letras y dígitos. Todas las apariciones de una misma letra deben sustituirse por el mismo dígito, no de puede repetir ningún dígito y ninguna palabra debe comenzar por el dígito 0.

> El rompecabezas más conocido de este tipo es: SEND + MORE = MONEY.

A este tipo de rompecabezas se les llama *alfaméticos* o *criptaritmos*. En este capitulo nos sumergiremos en un increíble programa escrito originariamente por Raymond Hettinger. Este programa resuelve este tipo de rompecabezas en *únicamente 14 líneas de código*.

```
1  import re
2  import itertools
3
4  def solve(puzzle):
5      words = re.findall('[A-Z]+', puzzle.upper())
6      unique_characters = set(''.join(words))
7      assert len(unique_characters) <= 10, 'Demasiadas letras'
8      first_letters = {word[0] for word in words}
9      n = len(first_letters)
10     sorted_characters = ''.join(first_letters) + \
11         ''.join(unique_characters - first_letters)
12     characters = tuple(ord(c) for c in sorted_characters)
13     digits = tuple(ord(c) for c in '0123456789')
14     zero = digits[0]
15     for guess in itertools.permutations(digits, len(characters)):
16         if zero not in guess[:n]:
17             equation = puzzle.translate(dict(zip(characters, guess)))
18             if eval(equation):
19                 return equation
20
21 if __name__ == '__main__':
22     import sys
23     for puzzle in sys.argv[1:]:
24         print(puzzle)
25         solution = solve(puzzle)
26         if solution:
27             print(solution)
```

Puedes ejecutar este programa desde la línea de comando. En Linux sería así. (Puede tardar un poco, dependiendo de la velocidad de tu ordenador, y no hay barra de progreso, ¡sé paciente!)

```
you@localhost:~/diveintopython3/examples$ python3 alphametics.py
 "HAWAII + IDAHO + IOWA + OHIO == STATES"
HAWAII + IDAHO + IOWA + OHIO = STATES
510199 + 98153 + 9301 + 3593 == 621246
you@localhost:~/diveintopython3/examples$ python3 alphametics.py
 "I + LOVE + YOU == DORA"
I + LOVE + YOU == DORA
1 + 2784 + 975 == 3760
you@localhost:~/diveintopython3/examples$ python3 alphametics.py
 "SEND + MORE == MONEY"
SEND + MORE == MONEY
9567 + 1085 == 10652
```

8.2. Encontrar todas las ocurrencias de un patrón

Lo primero que este solucionador de rompecabezas hace es encontrar todas las letras de la A a la Z del puzzle.

```
>>> import re
>>> re.findall('[0-9]+', '16 2-by-4s in rows of 8')
['16', '2', '4', '8']
>>> re.findall('[A-Z]+', 'SEND + MORE == MONEY')
['SEND', 'MORE', 'MONEY']
```

1. *Línea 2:* El módulo re contiene la implementación de Python de las expresiones regulares. Tiene una función muy simple de usar denominada findall() que toma un patrón de expresión regular y una cadena y encuentra todas las ocurrencias del patrón en la cadena. En este caso, el patrón debe coincidir con una secuencia de números. La función findall() devuelve una lista de las subcadenas que han coincidido con el patrón.

2. *Línea 4:* En este caso el patrón de la expresión regular coincide con una secuencia de letras. De nuevo, el valor de retorno es una lista y cada elemento de la lista es una cadena que representa una ocurrencia del patrón en la cadena original.

Aquí hay otro ejemplo que te servirá para forzar tu cerebro un poco.

```
>>> re.findall(' s.*? s', "The sixth sick sheikh's sixth sheep's sick.")
[' sixth s', " sheikh's s", " sheep's s"]
```

Este es el trabalenguas más difícil en el idioma inglés.

¿Sorprendido? La expresión regular busca un espacio, una letra s, y luego la serie de caacteres más corta posible formada por cualquier carácter (.*?), luego un espacio y luego otra s. Bien, echando un vistazo a la cadena de entrada vemos cinco coincidencias.

1. The sixth sick sheikh's sixth sheep's sick.
2. The sixth sick sheikh's sixth sheep's sick.
3. The sixth sick sheikh's sixth sheep's sick.
4. The sixth sick sheikh's sixth sheep's sick.
5. The sixth sick sheikh's sixth sheep's sick.

Pero la función re.findall() solamente devolvió tres coincidencias. En concreto, la primera, la tercera y la quinta. ¿Porqué? porque *no devuelve coincidencias solapadas*. La primera coincidencia se solapa con la segunda, por lo que se devuelve la primera y la segunda se salta. Luego la tercera se solapa con la cuarta, por lo que se devuelve la tercera y la cuarta se salta. Finalmente se devuelve la quinta coincidencia. Tres coincidencias, no cinco.

Esto no tiene nada que ver con el solucionador de rompecabezas alfaméticos, simplemente pensé que era interesante.

8.3. Encontrar los elementos únicos de una secuencia

Los conjuntos hacen que esta tarea sea trivial.

```
>>> a_list = ['The', 'sixth', 'sick', "sheik's", 'sixth', "sheep's", 'sick']
>>> set(a_list)
{'sixth', 'The', "sheep's", 'sick', "sheik's"}
>>> a_string = 'EAST IS EAST'
>>> set(a_string)
{'A', ' ', 'E', 'I', 'S', 'T'}
>>> words = ['SEND', 'MORE', 'MONEY']
>>> ''.join(words)
'SENDMOREMONEY'
>>> set(''.join(words))
{'E', 'D', 'M', 'O', 'N', 'S', 'R', 'Y'}
```

1. *Línea 2:* Dada una lista de varias cadenas, la funcion set() devolverá un conjunto de cadenas únicas de la lista. Esto cobra sentido si piensas en ello como si fuese un bucle for. Toma el primer elemento de la lista y lo pone en el conjunto. Luego el segundo, tercero, cuarto, quinto —un momento, este ya está en el conjunto, por lo que no se vuelve a incluir, porque los conjuntos de Python no permiten duplicados. El sexto, séptimo —de nuevo un duplicado, por lo que se no se vuelve a incluir. ¿El resultado final? Todos los elementos sin repeticiones de la lista original. La lista original ni siquiera necesita estar ordenada primero.

2. *Línea 5:* La misma técnica funciona con cadenas, puesto que una cadena es simplemente una secuencia de caracteres.

3. *Línea 8:* Dada una lista de cadenas, ''.join(a_list) concatena todas las cadenas en una única cadena.

4. *Línea 10:* Dada una cadena (secuencia de caracteres -al usar la función join-), esta línea de código retorna todos los caracteres sin duplicados.

El solucionador de rompecabezas alfaméticos utiliza esta técnica para construir un conjunto con todos los caracteres, sin repeticiones, existentes en el rompecabezas.

```
1 unique_characters = set(''.join(words))
```

Esta lista se utiliza después para asignar dígitos a los caracteres según el solucionador itera a través de las posibles soluciones.

8.4. Hacer aserciones

Como en muchos lenguajes de programación, en Python existe la sentencia assert. Veamos cómo funciona.

```
1 >>> assert 1 + 1 == 2
2 >>> assert 1 + 1 == 3
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in <module>
5 AssertionError
6 >>> assert 2 + 2 == 5, "Solamente para valores muy grandes de 2"
7 Traceback (most recent call last):
8   File "<stdin>", line 1, in <module>
9 AssertionError: Only for very large values of 2
```

1. *Línea 1:* La sentencia assert va seguida de cualquier expresión válida en Python. En este caso, la expresion es 1 + 1 == 2 que se evalúa a True, por lo que la sentencia assert no hace nada.

2. *Línea 2:* Sin embargo, si la expresión evalúa a False, la sentencia assert eleva una excepción AssertionError.

3. *Línea 6:* Puedes incluir un mensaje legible por una persona, que se imprime si se eleva la excepción AssertionError.

Por ello, esta línea de código:

```
1 assert len(unique_characters) <= 10, 'Demasiadas letras'
```

...es equivalente a ésta:

```
1 if len(unique_characters) > 10:
2     raise AssertionError('Demasiadas letras')
```

El solucionador de rompecabezas alfaméticos utiliza esta sentencia assert para terminar en caso de que el rompecabezas contenga más de diez letras diferentes. Como a cada letra se le asigna un dígito único y únicamente existen diez dígitos, un rompecabezas con más de diez letras diferentes no tendría solución posible.

8.5. Expresiones generadoras

Una expresión generadora es como una función generadora, pero sin escribir la función.

```
1 >>> unique_characters = {'E', 'D', 'M', 'O', 'N', 'S', 'R', 'Y'}
2 >>> gen = (ord(c) for c in unique_characters)
3 >>> gen
4 <generator object <genexpr> at 0x00BADC10>
5 >>> next(gen)
6 69
7 >>> next(gen)
8 68
9 >>> tuple(ord(c) for c in unique_characters)
10 (69, 68, 77, 79, 78, 83, 82, 89)
```

1. *Línea 2:* Una expresión generadora es como una función anónima que va devolviendo valores. La expresión en sí misma se parece bastante a una *lista por compresión*, pero se envuelve entre paréntesis en lugar de corchetes.

2. *Línea 3:* La expresión generadora devuelve... un iterador.

3. *Linea 5:* Al llamar a la función next(gen) se devuelve el siguiente valor del iterador.

4. *Línea 9:* Si quieres, puedes iterar a través de todos los valores convertirlo en una tupla, lista o conjunto, simplemente pasando la expresión generadora como parámetro de las funciones constructoras tupla(), list() o set(). En estos casos no necesitas utilizar el conjunto extra de paréntesis —simplemente pasa la expresión "desnuda" ord(c) for c unique_characters a la función tuple() y Python es capaz de saber que se trata de una expresión generadora.

   El uso de una expresión generadora en lugar de una lista por compresión puede ahorrar procesador y memoria RAM. Si necesitas construir una lista únicamente para luego tirarla (por ejemplo, para pasarla como parámetro a una función tuple() o set()), utiliza en su lugar una expresión generadora.

Esta es otra forma de hacer lo mismo pero utilizando una función generadora.

```
1 def ord_map(a_string):
2     for c in a_string:
3         yield ord(c)
4
5 gen = ord_map(unique_characters)
```

La expresión generadora es más compacta pero funcionalmente equivalente.

8.6. Cálculo de permutaciones... ¡De forma perezosa!

Ante todo ¿qué diablos son las permutaciones? Se trata de un concepto matemático. En realidad existen diferentes definiciones dependiendo del tipo de matemáticas que esté haciendo. Aquí estamos hablando de combinatoria, pero si no tiene sentido para ti, no te preocupes. Como siempre "la wikipedia es tu amiga": http://es.wikipedia.org/wiki/Permutación

La idea es que tomes una lista de cosas (podrían ser números, letras o osos bailarines) y encuentres todas las formas posibles de dividirlos en listas más pequeñas. Todas las listas más pequeñas tendrán el mismo tamaño, que puede ser desde 1 al número total de elementos. ¡Ah! y no se pueden repetir. Los matemáticos dicen

"vamos a encontrar las permutaciones de tres elementos tomados de dos en dos", lo que significa que la secuencia original tiene tres elementos y quieren encontrar todas las posibles listas ordenadas formadas por dos de ellos.

```
1  >>> import itertools
2  >>> perms = itertools.permutations([1, 2, 3], 2)
3  >>> next(perms)
4  (1, 2)
5  >>> next(perms)
6  (1, 3)
7  >>> next(perms)
8  (2, 1)
9  >>> next(perms)
10 (2, 3)
11 >>> next(perms)
12 (3, 1)
13 >>> next(perms)
14 (3, 2)
15 >>> next(perms)
16 Traceback (most recent call last):
17   File "<stdin>", line 1, in <module>
18 StopIteration
```

1. *Línea 1:* El módulo itertools contiene toda clase de funciones útiles, incluida una denominada permutations() que hace el trabajo duro de encontrar permutaciones.

2. *Línea 2:* La función permutations() toma como parámetros una secuencia (en este caso una lista de tres enteros) y un número, que es el número de elementos que debe contener cada grupo más pequeño. La función retorna un iterador, que puedes utilizar en un bucle for o en cualquier sitio en el que haya que iterar. En este ejemplo vamos a ir paso a paso con el iterador de forma manual para mostrar todos los valores.

3. *Línea 3:* La primera permutación de [1, 2, 3] tomando dos elementos cada vez es (1, 2).

4. *Línea 8:* Observa que las permutaciones que se van obteniendo son ordenadas: (2, 1) es diferente a (1, 2).

5. *Línea 15:* ¡Eso es! ya están todas las permutaciones de [1, 2, 3] tomadas de dos en dos. Las parejas (1, 1) y (2, 2) no se muestran nunca porque contienen repeticiones del mismo elemento por lo que no son permutaciones. Cuando un existen más permutaciones el iterador eleva una excepción StopIteration.

La función permutations() no necesita tomar una lista, puede tomar como parámetro cualquier secuencia —incluso una cadena de caracteres.

El módulo itertools contiene muchas utilidades

```
1  >>> import itertools
2  >>> perms = itertools.permutations('ABC', 3)
3  >>> next(perms)
4  ('A', 'B', 'C')
5  >>> next(perms)
6  ('A', 'C', 'B')
7  >>> next(perms)
8  ('B', 'A', 'C')
9  >>> next(perms)
10 ('B', 'C', 'A')
11 >>> next(perms)
12 ('C', 'A', 'B')
13 >>> next(perms)
14 ('C', 'B', 'A')
15 >>> next(perms)
16 Traceback (most recent call last):
17   File "<stdin>", line 1, in <module>
18 StopIteration
19 >>> list(itertools.permutations('ABC', 3))
20 [('A', 'B', 'C'), ('A', 'C', 'B'),
21  ('B', 'A', 'C'), ('B', 'C', 'A'),
22  ('C', 'A', 'B'), ('C', 'B', 'A')]
```

1. *Línea 2:* Una cadena de caracteres es una secuencia de caracteres. Para los propósitos de obtener permutaciones, la cadena 'ABC' es equivalente a la lista ['A', 'B', 'C'].

2. *Línea 4:* La primera permutación de los tres elementos ['A', 'B', 'C'] tomados de tres en tres, es ('A', 'B', 'C'). Hay otras cinco permutaciones —los mismos tres caracteres en cualquier orden posible.

3. *Línea 19:* Puesto que la función permutations() retorna siempre un iterador, una forma fácil de depurar las permutaciones es pasar ese iterador a la función interna list() para ver de forma inmediata todas las permutaciones posibles.

```
1  >>> import itertools
2  >>> list(itertools.product('ABC', '123'))
3  [('A', '1'), ('A', '2'), ('A', '3'),
4   ('B', '1'), ('B', '2'), ('B', '3'),
5   ('C', '1'), ('C', '2'), ('C', '3')]
6  >>> list(itertools.combinations('ABC', 2))
7  [('A', 'B'), ('A', 'C'), ('B', 'C')]
```

1. *Línea 2:* La función itertools.product() devuelve un iterador que contiene el producto cartesiano de dos secuencias.

2. *Línea 6:* La función itertools.combinations() devuelve un iterador con todas las posibles combinaciones de una longitud determinada. Es como la función itertools.permutation(), con la diferencia que las combinaciones no contienen los elementos repetidos en los que la única diferencia es el orden. Por eso itertools.permutations('ABC', 2) retorna ('A', 'B') y ('B', 'A') (entre otros), pero itertools.combinations('ABC', 2) no retornará ('B', 'A') al ser un uplicado de ('A', 'B') en orden diferente.

```
1  >>> names = list(open('examples/favorite-people.txt', encoding='utf-8'))
2  >>> names
3  ['Dora\n', 'Ethan\n', 'Wesley\n', 'John\n', 'Anne\n',
4  'Mike\n', 'Chris\n', 'Sarah\n', 'Alex\n', 'Lizzie\n']
5  >>> names = [name.rstrip() for name in names]
6  >>> names
7  ['Dora', 'Ethan', 'Wesley', 'John', 'Anne',
8  'Mike', 'Chris', 'Sarah', 'Alex', 'Lizzie']
9  >>> names = sorted(names)
10 >>> names
11 ['Alex', 'Anne', 'Chris', 'Dora', 'Ethan',
12 'John', 'Lizzie', 'Mike', 'Sarah', 'Wesley']
13 >>> names = sorted(names, key=len)
14 >>> names
15 ['Alex', 'Anne', 'Dora', 'John', 'Mike',
16 'Chris', 'Ethan', 'Sarah', 'Lizzie', 'Wesley']
```

1. *Línea 1:* Esta forma de leer un fichero retorna una lista formada por todas las líneas del fichero de texto.

2. *Línea 5:* Desafortunadamente (para este ejemplo), también incluye los retornos de carro al final de cada línea. Esta lista por compresión utiliza el método de cadenas rstrip() para eliminar los espacios en blanco del final de cada línea (Las cadenas de texto también tienen una función lstrip() para eliminar los espacios del comienzo de la cadena y un método strip() para eliminarlos por ambos lados).

3. *Línea 9:* La función sorted() toma una lista y la retorna ordenada. Por defecto, la ordena alfabéticamente.

4. *Línea 13:* Pero la función sorted() puede tomar un parámetro más denominado key que se utiliza para ordenar con él. En este caso, la función que se utiliza es len() por lo que ordena mediante len(cada elemento), por lo que los nombres más cortos van al principio, seguidos de los más largos.

¿Qué tiene esto que ver con el módulo itertools? Me alegro de que me hagas esa pregunta.

```
1  (...como continuación de la consola interactiva anterior...)
2  >>> import itertools
3  >>> groups = itertools.groupby(names, len)
4  >>> groups
5  <itertools.groupby object at 0x00BB20C0>
6  >>> list(groups)
7  [(4, <itertools._grouper object at 0x00BA8BF0>),
8   (5, <itertools._grouper object at 0x00BB4050>),
9   (6, <itertools._grouper object at 0x00BB4030>)]
10 >>> groups = itertools.groupby(names, len)
11 >>> for name_length, name_iter in groups:
12 ...     print('Nombres con {0:d} letras:'.format(name_length))
13 ...     for name in name_iter:
14 ...         print(name)
15 ...
16 Nombres con 4 letras:
17 Alex
18 Anne
19 Dora
20 John
21 Mike
22 Nombres con 5 letras:
23 Chris
24 Ethan
25 Sarah
26 Nombres con 6 letras:
27 Lizzie
28 Wesley
```

1. *Línea 3:* La función itertools.groupby() toma una secuencia y una función clave para retornar un iterador que genera parejas. Cada una de ellas contiene el resultado de la función que se usa como clave (key(cada elemento)) y otro iterador que contiene a todos los elementos que comparten el mismo resultado de la función clave.

2. *Línea 10:* Al haber utilizado la función list() en la línea anterior el iterador se consumió, puesto que ya se ha recorrido cada elemento del iterador para construir la lista. No hay un botón "reset" en un iterador, no puedes volver a usarlo una vez se ha "gastado". Si quieres volver a iterar por él (como en el bucle for que viene a continuación), necesitas llamar de nuevo a itertools.groupby() para crear un nuevo iterador.

3. *Línea 11:* En este ejemplo, dada una lista de nombres *ya ordenada por longitud*, la función itertools.groupby(names, len) pondrá todos los nombres de cuatro

letras en un iterador, los de cinco letras en otro, y así sucesivamente. La función groupby() es completamente genérica; podería agrupar cadenas por la primera letra, los números por el número de factores o cualuqier otra función *clave* que puedas imaginar.

La función itertools.groupby() solamente funciona si la secuencia de entrada ya está ordenada por la función de ordenación. En el ejemplo anterior, agrupamos una lista de nombres mediante la función len(). Esto funcionó bien porque la lista de entrada ya estaba ordenada por longitud.

¿Estás atento?

```
1  >>> list(range(0, 3))
2  [0, 1, 2]
3  >>> list(range(10, 13))
4  [10, 11, 12]
5  >>> list(itertools.chain(range(0, 3), range(10, 13)))
6  [0, 1, 2, 10, 11, 12]
7  >>> list(zip(range(0, 3), range(10, 13)))
8  [(0, 10), (1, 11), (2, 12)]
9  >>> list(zip(range(0, 3), range(10, 14)))
10 [(0, 10), (1, 11), (2, 12)]
11 >>> list(itertools.zip_longest(range(0, 3), range(10, 14)))
12 [(0, 10), (1, 11), (2, 12), (None, 13)]
```

1. *Línea 5:* La función itertools.chain() toma dos iteradores y retorna un iterador que contiene todos los elementos del primer iterador seguidos de todos los elementos del segundo iterador (en realidad puede tomar como parámetros cualquier número de iteradores, los encadenará a todos en el orden en el que se pasaron a la función).

2. *Línea 7:* La función zip() hace algo que suele ser extremadamente útil: toma cualquier número de secuencias y retorna un iterador que retorna tuplas con un elemento de cada secuencia (comenzando por el principio), la primera tupla contiene el primer elemento de cada secuencia, la segunda el segundo de cada secuencia, luego el tercero, y así sucesivamente.

3. *Línea 9:* La función zip() para cuando se acaba la secuencia más corta. range(10,14) tiene cuatro elementos (10, 11, 12 y 13), pero range(0,3) solamente tiene tres, por lo que la función zip() retorna un iterador con tres tuplas.

4. *Línea 11:* De otra parte, la función itertools.zip_longest() itera hasta el final de la secuencia más larga, insertando valores None en los elementos que corresponden a las secuencias que, por ser más cortas, ya se han consumido totalmente.

De acuerdo, todo eso es muy interesante, pero ¿cómo se relaciona con el solucionador de rompecabezas alfaméticos? Así:

```
1 >>> characters = ('S', 'M', 'E', 'D', 'O', 'N', 'R', 'Y')
2 >>> guess = ('1', '2', '0', '3', '4', '5', '6', '7')
3 >>> tuple(zip(characters, guess))
4 (('S', '1'), ('M', '2'), ('E', '0'), ('D', '3'),
5  ('O', '4'), ('N', '5'), ('R', '6'), ('Y', '7'))
6 >>> dict(zip(characters, guess))
7 {'E': '0', 'D': '3', 'M': '2', 'O': '4',
8  'N': '5', 'S': '1', 'R': '6', 'Y': '7'}
```

1. *Línea 3:* Dada una lista de letras y una lista de dígitos (cada uno representado aquí por una cadena de un carácter de longitud), la función zip() creará parejas de letras y dígitos en orden.

2. *Línea 6:* ¿Porqué es tan útil? Porque esta estructura de datos es exactamente la estructura adecuada para pasarla a la función dict() con el fin de crear un diccionario que utilice letras como claves y los dígitos asociados como valores (No es la única forma de hacerlo, por supuesto. Podrías utilizar un diccionario por compresión para crear el diccionario directamente). Aunque la representación impresa del diccionario muestra las parejas en orden diferente (los diccionarios no tiene "orden" por sí mismos), puedes observar que cada letra está asociada con el dígito en base a la ordenación original de las secuencias characters y guess.

El solucionador de rompecabezas alfaméticos utiliza esta técnica para crear un diccionario que empareja las letras del rompecabezas con los dígitos de la solución, para cada posible solución.

```
1 characters = tuple(ord(c) for c in sorted_characters)
2 digits = tuple(ord(c) for c in '0123456789')
3 ...
4 for guess in itertools.permutations(digits, len(characters)):
5     ...
6     equation = puzzle.translate(dict(zip(characters, guess)))
```

Pero ¿qué hace el método translate() —línea 6—? Ahora es cuando estamos llegando a la parte *realmente* divertida.

8.7. Una nueva forma de manipular listas

Las cadenas de Python tienen muchos métodos. Algunos de ellos los aprendiste en el capítulo dedicado a las cadenas: lower(), count() y format(). Ahora quiero explicarte una técnica potente pero poco conocida de manipulación de listas: el método translate().

```
1 >>> translation_table = {ord('A'): ord('O')}
2 >>> translation_table
3 {65: 79}
4 >>> 'MARK'.translate(translation_table)
5 'MORK'
```

1. *Línea 1:* La traducción de cadenas comienza con una tabla de traducción, que es un diccionario que empareja un carácter con otro. En realidad, decir "carácter" es incorrecto —la tabla de traducción realmente empareja un *byte* con otro.

2. *Línea 2:* Recuerda que en Python 3 los bytes son enteros. La función ord() devuelve el valor ASCII de un carácter, lo que, en el caso de A-Z, siempre es un byte entre el 65 y el 90.

3. *Línea 4:* El método translate() se aplica sobre una cadena de texto utilizando una tabla de traducción. Reemplaza todas las ocurrencias de las claves de la tabla de traducción por los valores correspondientes. En este caso se "traduce" MARK a MORK.

¿Qué tiene esto que ver con la resolución de rompecabezas alfaméticos?. Como verás a continuación: todo.

```
1  >>> characters = tuple(ord(c) for c in 'SMEDONRY')
2  >>> characters
3  (83, 77, 69, 68, 79, 78, 82, 89)
4  >>> guess = tuple(ord(c) for c in '91570682')
5  >>> guess
6  (57, 49, 53, 55, 48, 54, 56, 50)
7  >>> translation_table = dict(zip(characters, guess))
8  >>> translation_table
9  {68: 55, 69: 53, 77: 49, 78: 54, 79: 48, 82: 56, 83: 57, 89: 50}
10 >>> 'SEND + MORE == MONEY'.translate(translation_table)
11 '9567 + 1085 == 10652'
```

1. *Línea 1:* Mediante una expresión generadora calculamos rápidamente los valores de los bytes de cada carácter de una cadena. characters es un ejemplo del valor que puede contener sorted_characters en la función alphametics.solve().

2. *Línea 4:* Mediante el uso de otra expresión generadora calculamos rápidamente los valores de los bytes de cada dígito de esta cadena. El resultado, guess, se encuentra en la forma que retorna la función itertools.permutations() en la función alphametics.solve().

3. *Línea 7:* Esta tabla de traducción se genera mediante la función zip() uniendo los characters y guess en un diccionario. Esto es exactamente lo que la función alphametics.solve() hace en el bucle for.

4. *Línea 10:* Finalmente, pasamos esta tabla de traducción al método translate() aplicándolo a la cadena original del rompecabezas. Esto convierte cada letra de la cadena en el dígito que le corresponde (basado en las letras existentes en characters y los dígitos de guess). El resultado es una expresión válida de Python, aunque en forma de cadena de texto.

Esto es impresionante. Pero ¿Qué puedes hacer con una cadena que representa a una expresión válida de Python?

8.8. Evaluación de cadenas de texto como expresiones de Python

Esta es la pieza final del rompecabezas (o mejor dicho, la pieza final del solucionador de rompecabezas). Después de tanta manipulación de cadenas tan moderna, nos ha quedado una cadena como '9567 + 1085 == 10652'. Pero es una cadena de texto, y ¿para qué nos vale una cadena de texto? Pide paso eval(), la herramienta universal para evaluación de expresiones en Python.

```
>>> eval('1 + 1 == 2')
True
>>> eval('1 + 1 == 3')
False
>>> eval('9567 + 1085 == 10652')
True
```

Pero espera, ¡hay más! La función eval() no se limita a expresiones booleanas. Puede manejar *cualquier* expresión en Python y devolver *cualquier* tipo de datos.

```
1 >>> eval('"A" + "B"')
2 'AB'
3 >>> eval('"MARK".translate({65: 79})')
4 'MORK'
5 >>> eval('"AAAAA".count("A")')
6 5
7 >>> eval('["*"] * 5')
8 ['*', '*', '*', '*', '*']
```

Pero espera, ¡que eso no es todo!

```
1 >>> x = 5
2 >>> eval("x * 5")
3 25
4 >>> eval("pow(x, 2)")
5 25
6 >>> import math
7 >>> eval("math.sqrt(x)")
8 2.2360679774997898
```

1. *Línea 2:* La expresión que eval() recibe como parámetro puede referenciar a cualquier variable global definida fuera de la función eval(). Si se llama desde una función, también puede referenciar variables locales.
2. *Línea 4:* Y funciones.
3. *Línea 7:* Y módulos.

Pero espera un minuto...

```
1 >>> import subprocess
2 >>> eval("subprocess.getoutput('ls ~')")
3 'Desktop           Library         Pictures \
4  Documents         Movies          Public   \
5  Music             Sites'
6 >>> eval("subprocess.getoutput('rm /some/random/file')")
```

1. *Línea 2:* El módulo subprocess te permite ejecutar comandos de la línea de comandos y obtener el resultado como una cadena de Python.
2. *Línea 6:* Los comandos de la línea de comandos pueden producir resultados permanentes.

Es incluso peor que esto, porque existe una función global __import__() que toma como parámetro el nombre de un módulo como cadena de texto, importa el

módulo y devuelve una referencia a él. Combinado con el poder de la función eval() puedes construir una expresión que te borre todos los ficheros:

```
>>> eval("__import__('subprocess').getoutput('rm /some/random/file')")
```

Ahora imagina la salida de 'rm -rf '. Realmente no habría ninguna salida por pantalla, pero tampoco te habría quedado ningún fichero en tu cuenta de usuario.

eval() es MALIGNO

Bueno, lo maligno de de eval es la posibilidad de evaluar expresiones procedentes de fuentes que no sean de confianza. Solamente deberías utilizar eval() para entradas "de confianza". Lo complicado es saber qué es de "confianza". Pero hay algo que debes tener seguro: no deberías tomar este solucionador de rompecabezas alfaméticos y ponerlo en Internet como un servicio web. No cometas el error de pensar, "la función hace un montón de manipulaciones de cadenas antes de evaluar la cadena; no puedo imaginar cómo alguien podría explotar eso". Alguien descubrirá una forma de introducir algún código maligno que traspase toda la manipulación de cadenas previa(echa un vistazo a: http://www.securityfocus.com/blogs/746 y verás que cosas más raras han pasado), y podrás irte despidiendo de tu servidor.

Pero... ¿Seguro que existe una forma de evaluar expresiones de forma segura? ¿Para que eval() se ejecute en un entorno aislado en el que no se pueda acceder a él desde el exterior? Sí y no.

```
1  >>> x = 5
2  >>> eval("x * 5", {}, {})
3  Traceback (most recent call last):
4    File "<stdin>", line 1, in <module>
5    File "<string>", line 1, in <module>
6  NameError: name 'x' is not defined
7  >>> eval("x * 5", {"x": x}, {})
8  >>> import math
9  >>> eval("math.sqrt(x)", {"x": x}, {})
10 Traceback (most recent call last):
11   File "<stdin>", line 1, in <module>
12   File "<string>", line 1, in <module>
13 NameError: name 'math' is not defined
```

1. *Línea 2:* El segundo y tercer parámetro que se pasen a la función eval() actúan como los espacios de nombre local y global para evaluar la expresión. En este caso ambos están vacíos, lo que significa que cuando se evalúa la cadena "x * 5" no existe referencia a x ni en el espacio de nombres local ni en el global, por lo que eval() genera una excepción.

2. *Línea 7:* Puedes incluir selectivamente valores específicos en el espacio de nombres global listándolos individualmente. Entonces, serán las únicas variables que estarán disponibles durante la evaluación.

3. *Línea 9:* Incluso aunque hayas importado el módulo math, no se incluyó en el espacio de nombres que se pasó a la función eval(), por lo que la evaluación falla.

Parece que fue fácil. ¡Déjame ya que haga un servicio web con el solucionador de rompecabezas alfamético!

```
>>> eval("pow(5, 2)", {}, {})
25
>>> eval("__import__('math').sqrt(5)", {}, {})
2.2360679774997898
```

1. *Línea 1:* Incluso aunque hayas pasado diccionarios vacíos para el espacio de nombres local y global, siguien estando disponibles todas las funciones internas de Python. Por eso pow(5, 2) funciona, porque 5 y 2 son literales y pow() es una función interna.

2. *Línea 3:* Desafortunadamente (y si no ves porqué es desafortunado, sigue leyendo), la función __import__() es interna, por lo que también funciona.

Vale, eso significa que aún puedes hacer cosas malignas, incluso aunque explicitamente establezcas los espacios de nombre local y global a diccionarios vacíos, cuando llamas a la función eval().

```
>>> eval("__import__('subprocess').getoutput('rm /some/random/file')", {}, {})
```

Vale, estoy contento de no haber hecho el servicio web del solucionador de rompecabezas alfamético. ¿Hay *algún* modo de usar eval() de forma segura? Bueno, sí y no.

```
>>> eval("__import__('math').sqrt(5)",
...       {"__builtins__":None}, {})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name '__import__' is not defined
>>> eval("__import__('subprocess').getoutput('rm -rf /')",
...       {"__builtins__":None}, {})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name '__import__' is not defined
```

1. *Línea 2:* Para evaluar de forma segura expresiones en las que no confíes, necesitas definir el diccionario del espacio de nombres global para que mapee "__builtins__" a None, el valor nulo de Python. Internamente, las funciones "internas" se encuentran en un pseudo-módulo denominado "__builtins__". Este pseudo-módulo (el conjunto de funciones internas) está disponible para evaluar expresiones a menos que expresamente lo sustituyas.

2. *Línea 8:* Asegúrate de que sustituyes "__builtins__". No __builtin__, __built-ins__ o alguna otra variación parecida que te exponga a riesgos catastróficos.

¿Así que ahora ya es seguro utilizar eval()? Bueno, sí y no.

```
1 >>> eval("2 ** 2147483647",
2 ...      {"__builtins__":None}, {})
```

Incluso sin acceso a las funciones internas, puedes lanzar aún un ataque de denegación del servicio. Por ejemplo, elevar 2 a la potencia 2147483647 hará que el uso de la CPU del servidor se eleve al 100 % durante algún tiempo (si pruebas esto en modo interactivo pulsa Ctrl-C varias veces hasta que se cancele). Técnicamente esta expresión *retornará* un valor en algún momento, pero mientras tanto el servidor estará ocupado.

Al final, *es* posible evaluar de forma segura expresiones Python de fuentes que no sean de confianza, para una definición de "seguro" que no es muy útil en la vida real. Está bien que hagas pruebas, y está bien si solamente le pasas datos de fuentes de confianza. Pero cualquier otra cosa es está *buscando problemas*.

8.9. Juntándolo todo

Para recapitular: este programa resuelve rompecabezas alfaméticos mediante la fuerza bruta, a través de la búsqueda exhaustiva de todas las posibles combinaciones de solución. Para hacer esto, los pasos que sigue son:

1. Encuentra todas las letras del rompecabezas con la función re.findall().

2. Determina las letras que son, sin repetición, utilizando conjuntos y la función set().

3. Comprueba si hay más de 10 letras diferentes (lo que significaría que el rompecabezas no tiene solución) con la sentencia assert.

4. Convierte las letras a sus equivalentes en ASCII con un objeto generador.

5. Calcula todas las posibles soluciones con la función itertools.permutations().
6. Convierte cada una de las soluciones posibles a una expresión en Python con el método de cadenas de texto translate().
7. Prueba cada una de las posibles soluciones evaluando la expresión Python con la función eval().
8. Devuelve la primera solución que se evalúa a True.

...en sólo catorce líneas de código.

8.10. Lecturas recomendadas

- el módulo itertools:
  http://docs.python.org/3.1/library/itertools.html
- itertools—Funciones iteradoras para bucles eficientes:
  http://www.doughellmann.com/PyMOTW/itertools/
- Video de Raymond Hettinger con la charla "Inteligencia Artificial sencilla con Python" en la PyCon 2009:
  http://blip.tv/file/1947373/
- Receta 576615 - solucionador de rompecabezas alfaméticos de Raymond Hettinger para Python 2:
  http://code.activestate.com/recipes/576615/
- Más recetas de Raymond Kettinger en el repositorio de código ActiveState:
  http://code.activestate.com/recipes/users/178123/
- Alfamética en la wikipedia:
  http://en.wikipedia.org/wiki/Verbal_arithmetic
- Indice alfamético:
  http://www.tkcs-collins.com/truman/alphamet/index.shtml
  Con muchos rompecabezas:
  http://www.tkcs-collins.com/truman/alphamet/alphamet.shtml
  Y un generador de rompecabezas alfaméticos:
  http://www.tkcs-collins.com/truman/alphamet/alpha_gen.shtml

Muchas gracias a Raymond Hattinger por permitirme relicenciar su código para que pudiera portarlo a Python 3 y utilizarlo como base para este capítulo.

Capítulo 9

Pruebas unitarias

Nivel de dificultad:◆ ◆ ◇ ◇ ◇

"La certidumbre no es prueba de que algo sea cierto.
Hemos estado tan seguros de tantas cosas que luego no lo eran."
—Oliver Wendell Holmes, Jr

9.1. (Sin) Inmersión

En este capítulo vas a escribir y depurar un conjunto de funciones de utilidad para convertir números romanos (en ambos sentidos). En el caso de estudio de los números romanos ya vimos cual es la mecánica para construir y validar números romanos. Ahora vamos a volver a él para considerar lo que nos llevaría expandirlo para que funcione como una utilidad en ambos sentidos.

Las reglas para los números romanos sugieren una serie de interesantes observaciones:

- Solamente existe una forma correcta de representar un número cualquiera con los dígitos romanos.
- Lo contrario también es verdad: si una cadena de caracteres es un número romano válido, representa a un único número (solamente se puede interpretar de una forma).
- Existe un rango limitado de valores que se puedan expresar como números romanos, en concreto del `1` al `3999`. Los romanos tenían varias maneras de expresar números mayores, por ejemplo poniendo una barra sobre un número

para indicar que el valor normal que representaba tenía que multiplicarse por 1000. Para los propósitos que perseguimos en este capítulo, vamos a suponer que los números romanos van del 1 al 3999.

- No existe ninguna forma de representar el 0 en números romanos.
- No hay forma de representar números negativos en números romanos.
- No hay forma de representar fracciones o números no enteros con números romanos.

Vamos a comenzar explicando lo que el módulo roman.py debería hacer. Tendrá que tener dos funciones principales to_roman() y from_romman(). La función to_roman() debería tomar como parámetro un número entero de 1 a 3999 y devolver su representación en números Romanos como una cadena...

Paremos aquí. Vamos a comenzar haciendo algo un poco inesperado: escribir un caso de prueba que valide si la función to_roman() hace lo que quieres que haga. Lo has leído bien: vas a escribir código que valide la función que aún no has escrito.

A esto se le llama *desarrollo dirigido por las pruebas —test driven development, o TDD—*. El conjunto formado por las dos funciones de conversión —to_roman(), y from_roman()— se puede escribir y probar como una unidad, separadamente de cualquier programa mayor que lo utilice. Python dispone de un marco de trabajo (framework) para pruebas unitarias, el módulo se llama, apropiadamente, unittest.

La prueba unitaria es una parte importante de una estrategia de desarrollo centrada en las pruebas. Si escribes pruebas unitarias es importante escribirlas pronto y mantenerlas actualizadas con el resto del código y con los cambios de requisitos. Muchas personas dicen que las pruebas se escriban antes que el código que se vaya a probar, y este es el estilo que voy a enseñarte en este capítulo. De todos modos, las pruebas unitarias son útiles independientemente de cuando las escribas.

- Antes de escribir el código, el escribir las pruebas unitarias te oblica a detallar los requisitos de forma útil.
- Durante la escritura del código, las pruebas unitarias evitan que escribas demasiad código. Cuando pasan los todos los casos de prueba, la función está completa.
- Cuando se está reestructurando el código[1], pueden ayudar a probar que la nueva versión se comporta de igual manera que la vieja.

[1] refactoring

- Cuando se mantiene el código, disponer de pruebas te ayuda a cubrirte el trasero cuando alquien viene gritando que tu último cambio en el código ha roto el antiguo que ya funcionaba.
- Cuando codificas en un equipo, disponer de un juego de pruebas completo disminuye en gran medida la probabilidad de que tu código rompa el de otra persona, ya que puedes ejecutar los casos de prueba antes de introducir cambios (He visto esto pasar en las competiciones de código. Un equipo trocea el trabajo asignado, todo el mundo toma las especificaciones de su tarea, escribe los casos de prueba para ella en primer lugar, luego comparte sus casos de prueba con el resto del equipo. De ese modo, nadie puede perderse demasiado desarrollando código que no funcione bien con el del resto).

9.2. Una única pregunta

Un caso de prueba (unitaria) contesta a una única pregunta sobre el código que está probando. Un caso de prueba (unitaria) debería ser capaz de...

- ...ejecutarse completamente por sí mismo, sin necesidad de ninguna entrada manual. Las pruebas unitarias deben ser automáticas.
- ...determinar por sí misma si la función que se está probando ha funcionado correctamente o a fallado, sin la necesidad de que exista una persona que interprete los resultados.
- ...ejecutarse de forma aislada, separada de cualquier otro caso de prueba (incluso aunque estos otros prueben las mismas funciones). Cada caso de prueba es una isla.

Dado esto, construyamos un caso de prueba para el primer requisito:

> Toda prueba es una isla.

- La función `to_roman()` debería devolver el número Romano que represente a los números enteros del `1` al `3999`.

Lo que no es obvio es cómo el código siguiente efectúa dicho cálculo. Define una clase que no tiene método `__init__()`. La clase *tiene* otro método, pero nunca se llama. El código tiene un bloque `__main__`, pero este bloque ni referencia a la clase ni a su método. Pero hace algo, te lo juro.

```
1  # romantest1.py
2  import roman1
3  import unittest
4
5  class KnownValues(unittest.TestCase):
6      known_values = ( (1, 'I'), (2, 'II'),
7                       (3, 'III'), (4, 'IV'),
8                       (5, 'V'), (6, 'VI'),
9                       (7, 'VII'), (8, 'VIII'),
10                      (9, 'IX'), (10, 'X'),
11                      (50, 'L'), (100, 'C'),
12                      (500, 'D'), (1000, 'M'),
13                      (31, 'XXXI'), (148, 'CXLVIII'),
14                      (294, 'CCXCIV'), (312, 'CCCXII'),
15                      (421, 'CDXXI'), (528, 'DXXVIII'),
16                      (621, 'DCXXI'), (782, 'DCCLXXXII'),
17                      (870, 'DCCCLXX'), (941, 'CMXLI'),
18                      (1043, 'MXLIII'), (1110, 'MCX'),
19                      (1226, 'MCCXXVI'), (1301, 'MCCCI'),
20                      (1485, 'MCDLXXXV'), (1509, 'MDIX'),
21                      (1607, 'MDCVII'), (1754, 'MDCCLIV'),
22                      (1832, 'MDCCCXXXII'), (1993, 'MCMXCIII'),
23                      (2074, 'MMLXXIV'), (2152, 'MMCLII'),
24                      (2212, 'MMCCXII'), (2343, 'MMCCCXLIII'),
25                      (2499, 'MMCDXCIX'), (2574, 'MMDLXXIV'),
26                      (2646, 'MMDCXLVI'), (2723, 'MMDCCXXIII'),
27                      (2892, 'MMDCCCXCII'), (2975, 'MMCMLXXV'),
28                      (3051, 'MMMLI'), (3185, 'MMMCLXXXV'),
29                      (3250, 'MMMCCL'), (3313, 'MMMCCCXIII'),
30                      (3408, 'MMMCDVIII'), (3501, 'MMMDI'),
31                      (3610, 'MMMDCX'), (3743, 'MMMDCCXLIII'),
32                      (3844, 'MMMDCCCXLIV'), (3888, 'MMMDCCCLXXXVIII'),
33                      (3940, 'MMMCMXL'), (3999, 'MMMCMXCIX'))
34
35     def test_to_roman_known_values(self):
36         '''to_roman should give known result with known input'''
37         for integer, numeral in self.known_values:
38             result = roman1.to_roman(integer)
39             self.assertEqual(numeral, result)
40
41 if __name__ == '__main__':
42     unittest.main()
```

1. *Línea 5:* Para escribir un caso de prueba, lo primero es crear una subclase de TestCase del módulo unittest. Esta clase proporciona muchos métodos útiles que se puedne utilizar en los casos de prueba para comprobar condiciones específicas.

2. *Línea 33:* Esto es una lista de pares de números enteros y sus números romanos equivalentes que he verificado manualmente. Incluye a los diez primeros números, el mayor, todos los números que se convierten un único carácter romano, y una muestra aleatoria de otros números válidos. No es necesario probar todas las entradas posibles, pero deberían probarse los casos de prueba límite.

3. *Línea 35:* Cada caso de prueba debe escribirse en su propio método, que no debe tomar parámetros y no debe devolver ningún valor. Si el método finaliza normalmente sin elevar ninguna excepción, se considera que la prueba ha pasado; si el método eleva una excepción, se considera que la prueba ha fallado.

4. *Línea 38:* Este es el punto en el que se llama a la función to_roman() (bueno, la función aún no se ha escrito, pero cuando esté escrita, esta será la línea que la llamará). Observa que con esto has definido la API de la función to_roman(): debe tomar como parámetro un número entero (el número a convertir) y devolver una cadena (la representación del número entero en números romanos). Si la API fuese diferente a esta, este test fallará. Observa también que no estamos capturando excepciones cuando llamamos a la función to_roman(). Es intencionado, to_roman() no debería devolver una excepción cuando la llames con una entrada válida, y todas las entradas previstas en este caso de prueba son válidas. Si to_roman() elevase una excepción, esta prueba debería considerarse fallida.

5. *Línea 38:* Asumiendo que la función to_roman() fuese definida correctamente, llamada correctamente, ejecutada correctamente y retornase un valor, el último paso es validar si el valor retornado es el *correcto*. Esta es una pregunta habitual, y la clase TestCase proporciona un método, assertEqual, para validar si dos valores son iguales. Si el resultado que retorna to_roman() (result) no coincide con el valor que se estaba esperando (numeral), assertEqual elevará una excepción y la prueba fallará. Si los dos valores son iguales, assertEqual no hará nada. Si todos los valores que retorna to_roman() coinciden con los valores esperados, la función assertEqual nunca eleva una excepción, por lo que la función test_to_roman_known_values termina normalmente, lo que significa que la función to_roman() ha pasado esta prueba.

Escribe un caso de prueba que falle, luego codifica hasta que funcione.

Cuando ya tienes un caso de prueba, puedes comenzar a codificar la función to_roman(). Primero deberías crearla como una función vacía y asegurarte de que la prueba falla. Si la prueba funciona antes de que hayas escrito ningún código, ¡¡tus pruebas no están probando nada!! La

prueba unitaria es como un baile: la prueba va llevando, el código la va siguiendo. Escribe una prueba que falle, luego codifica hasta que el código pase la prueba.

```
1 # roman1.py
2
3 def to_roman(n):
4     '''convert integer to Roman numeral'''
5     pass
```

1. *Línea 5:* En este momento debes definir la API de la función to_roman(), pero no quieres codificarla aún (Las pruebas deben fallar primero). Para conseguirlo, utiliza la palabra reservada de Python pass que, precisamente, sirve para no hacer nada.

Ejecuta romantest1.py en la línea de comando para ejecutar la prueba. Si lo llamas con la opción -v de la línea de comando, te mostrará una salida con más información de forma que puedas ver lo que está sucediendo conforme se ejecutan los casos de prueba. Con suerte, la salida deberá parecerse a esto:

```
1 you@localhost:~/diveintopython3/examples$ python3 romantest1.py -v
2 test_to_roman_known_values (__main__.KnownValues)
3 to_roman should give known result with known input ... FAIL
4
5 ======================================================================
6 FAIL: to_roman should give known result with known input
7 ----------------------------------------------------------------------
8 Traceback (most recent call last):
9   File "romantest1.py", line 73, in test_to_roman_known_values
10    self.assertEqual(numeral, result)
11 AssertionError: 'I' != None
12
13 ----------------------------------------------------------------------
14 Ran 1 test in 0.016s
15
16 FAILED (failures=1)
```

1. *Línea 2:* Al ejecutar el programa se ejecuta unittest.main(), que ejecuta el caso de prueba. Cada caso de prueba es un método de una clase en romantest.py. No se requiere ninguna organización de estas clases de prueba; pueden contener un método de prueba cada una o puede existir una única clase con muchos métodos de prueba. El único requisito es que cada clase de prueba debe heredar de unittest.TestCase.

2. *Línea 3:* Para cada caso de prueba, el módulo unittest imprimirá el docstring

(cadena de documentación) del méotodo y si la prueba ha pasado o ha fallado. En este caso, como se esperaba, la prueba ha fallado.

3. *Línea 11:* Para cada caso de prueba que falle, unittest muestra la información de traza que muestra exactamente lo que ha sucedido. En este caso la llamada a assertEqual() elevó la excepción AssertionError porque estaba esperando que to_roman(1) devolviese 'I', y no o ha devuelto (Debido a que no hay valor de retorno explícito, la función devolvió None, el valor nulo de Python).

4. *Línea 14:* Después del detalle de cada prueba, unittest muestra un resumen de los test que se han ejecutado y el tiempo que se ha tardado.

5. *Línea 16:* En conjunto, la ejecución de la prueba ha fallado puesto que al menos un caso de prueba lo ha hecho. Cuando un caso de prueba no pasa, unittest distingue entre fallos y errores. Un fallo se produce cuando se llama a un método assertXYZ, como assertEqual o assertRaises, que fallan porque la condición que se evalúa no sea verdadera o la excepción que se esperaba no ocurrió. Un error es cualquier otra clase de excepción en el código que estás probando o en el propio caso de prueba.

Ahora, finalmente, puedes escribir la función to_roman().

```
1  roman_numeral_map = (('M',  1000),
2                       ('CM', 900),
3                       ('D',  500),
4                       ('CD', 400),
5                       ('C',  100),
6                       ('XC', 90),
7                       ('L',  50),
8                       ('XL', 40),
9                       ('X',  10),
10                      ('IX', 9),
11                      ('V',  5),
12                      ('IV', 4),
13                      ('I',  1))
14
15 def to_roman(n):
16     '''convert integer to Roman numeral'''
17     result = ''
18     for numeral, integer in roman_numeral_map:
19         while n >= integer:
20             result += numeral
21             n -= integer
22     return result
```

1. *Línea 13:* La variable roman_numeral_map es una tupla de tuplas que define tres cosas: las representaciones de caracteres de los números romanos básicos; el orden de los números romanos (en orden descendente, desde M a I), y el valor de cada número romano. Cada tupla interna es una pareja (número romano, valor). No solamente define los números romanos de un único carácter, también define las parejas de dos caracteres como CM (çien menos que mil"). Así el código de la función to_roman se hace más simple.

2. *Línea 19:* Es aquí en donde la estructura de datos roman_numeral_map se muestra muy útil, porque no necesitas ninguna lógica especial para controlar la regla de restado. Para convertir números romanos solamente tienes que iterar a través de la tupla roman_numeral_map a la búsqueda del mayor valor entero que sea menor o igual al valor introducido. Una vez encontrado, se concatena su representación en número romano al final de la salida, se resta el valor correspondiente del valor inicial y se repite hasta que se consuman todos los elementos de la tupla.

Si aún no ves claro cómo funciona la función to_roman() añade una llamada a print() al final del bucle while:

```
1 ...
2 while n >= integer:
3     result += numeral
4     n -= integer
5     print('restando {0} de la entrada, sumando {1} a la salida'.format(
6                                                         integer, numeral)
7           )
8 ...
```

Con estas sentencias, la salida es la siguiente:

```
1 >>> import roman1
2 >>> roman1.to_roman(1424)
3 restando 1000 de la entrada, sumando M a la salida
4 restando 400 de la entrada, sumando CD a la salida
5 restando 10 de la entrada, sumando X a la salida
6 restando 10 de la entrada, sumando X a la salida
7 restando 4 de la entrada, sumando IV a la salida
8 'MCDXXIV'
```

Por lo que parece que funciona bien la función to_roman(), al menos en esta prueba manual. Pero, ¿Pasará el caso de prueba que escribimos antes?

```
you@localhost:~/diveintopython3/examples$ python3 romantest1.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok

----------------------------------------------------------------------
Ran 1 test in 0.016s

OK
```

1. *Línea 3:* ¡Bien! la función to_roman() pasa los valores conocidos del caso de prueba. No es exhaustivo, pero revisa la función con una amplia variedad de entradas. Incluyendo entradas cuyo resultado son todos los números romanos de un único carácter, el valor mayor (3999) y la entrada que produce el número romano más largo posible (3888). En este punto puedes confiar razonablemente en que la función está bien hecha y funciona para cualquier valor de entrada válido que puedas consultar.

   ¿Valores "válidos¿ ¿Qué es lo que pasará con valores de entrada no válidos?

9.3. "Para y préndele fuego"

No es suficiente con probar que las funciones pasan las pruebas cuando éstas incluyen valores correctos; tienes que probar que las funciones fallan cuando se les pasa una entrada no válida. Y que el fallo no es uno cualquiera; deben fallar de la forma que esperas.

> La forma de parar la ejecución para indicar un fallo es elevar una excepción

```
>>> import roman1
>>> roman1.to_roman(4000)
'MMMM'
>>> roman1.to_roman(5000)
'MMMMM'
>>> roman1.to_roman(9000)
'MMMMMMMM'
```

1. *Línea 6:* Esto no es lo que querías —ni siquiera es un número romano válido. De hecho, todos estos números están fuera del rango de los que son aceptables como entrada a la función, pero aún así, la función devuelve valores falsos. Devolver valores "malos" sin advertencia alguna es algo *maaaalo*. Si un programa

debe fallar, lo mejor es que lo haga lo más cerca del error, rápida y evidentemente. Mejor "parar y prenderle fuego" como diche el dicho. La forma de hacerlo en Python es elevar una excepción.

La pregunta que te tienes que hacer es, *¿Cómo puedo expresar esto como un requisito que se pueda probar?* ¿Qué tal esto para comenzar?:

La función to_roman() debería elevar una excepción OutOfRangeError cuando se le pase un número entero mayor que 3999.

¿Cómo sería esta prueba?

```
1 class ToRomanBadInput(unittest.TestCase):
2     def test_too_large(self):
3         '''to_roman debería fallar con una entrada muy grande'''
4         self.assertRaises(roman2.OutOfRangeError, roman2.to_roman, 4000)
```

1. *Línea 1:* Como en el caso de prueba anterior, has creado una clase que hereda de unittest.TestCase. Puedes crear más de una prueba por cada clase (como verás más adelante en este mismo capítulo), pero aquí he elegido crear una clase nueva porque esta prueba es algo diferente a la anterior. En este ejemplo, mantendremos todas las pruebas sobre entradas válidas en una misma clase y todas las pruebas que validen entradas no válidas en otra clase.

2. *Línea 2:* Como en el ejemplo anterior, la prueba es un método de la clase que tenga un nombre que comience por test.

3. *Línea 4:* La clase unittest.TestCase proporciona el método assertRaises, que toma los siguientes parámetros: la excepción que se debe esperar, la función que se está probando y los parámetros que hay que pasarle a la función (Si la función que estás probando toma más de un parámetro hay que pasarlos todos assertRaises en el orden que los espera la función, para que assertRaises los pueda pasar, a su vez, a la función a probar.

Presta atención a esta última línea de código. En lugar de llamar directamente a la función to_roman() y validar a mano si eleva una excepción concreta (mediante un bloque try ... except), el método assertRaises se encarga de ello por nosotros. Todo lo que hay que hacer es decirle la excepción que estamos esperado (roman.OutOfRangeError, la función (to_roman()) y los parámetros de la función (4000). El método assertRaises se encarga de llamar a la función to_roman() y valida que eleve roman2.OutRangeError.

Observa también que estás pasando la propia función to_roman() como un parámetro; no estás ejecutándola y no estás pasando el nombre de la función como una cadena. ¿He mencionado ya lo oportuno que es que todo en Python sea un objeto?

¿Qué sucede cuando ejecutas esta "suite" de pruebas con esta nueva prueba?

```
you@localhost:~/diveintopython3/examples$ python3 romantest2.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_too_large (__main__.ToRomanBadInput)
to_roman debería fallar con una entrada muy grande ... ERROR

======================================================================
ERROR: to_roman debería fallar con una entrada muy grande
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest2.py", line 78, in test_too_large
    self.assertRaises(roman2.OutOfRangeError, roman2.to_roman, 4000)
AttributeError: 'module' object has no attribute 'OutOfRangeError'

----------------------------------------------------------------------
Ran 2 tests in 0.000s

FAILED (errors=1)
```

1. *Línea 5:* Deberías esperar que fallara (puesto que aún no has escrito ningún código que pase la prueba), pero... en realidad no "falló", en su lugar se produjo un "error". Hay una sutil pero importante diferencia. Una prueba unitaria puede terminar con tres tipos de valores: pasar la prueba, fallar y error. Pasar la prueba significa que el código hace lo que se espera. "fallar" es lo que hacía la primera prueba que hicimos (hasta que escribimos el código que permitía pasar la prueba) —ejecutaba el código pero el resultado no era el esperado. "error" significa que el código ni siquiera se ejecutó apropiadamente.

2. *Línea 13:* ¿Porqué el código no se ejecutó correctamente? La traza del error lo indica. El módulo que estás probando no dispone de una excepción denominada OutOfRangeError. Recuerda, has pasado esta excepción al método assertRaises() porque es la excepción que quieres que que la función eleve cuando se le pasa una entrada fuera del rango válido de valores. Pero la excepción aún no existe, por lo que la llamada al método assertRaises() falla. Ni siquiera ha tenido la oportunidad de probar el funcionamiento de la función to_roman(); no llega tan lejos.

Para resolver este problema necesitas definir la excepción OutOfRangeError en el fichero roman2.py.

```
1 class OutOfRangeError(ValueError):
2     pass
```

1. *Línea 1:* Las excepciones son clases. Un error "fuera de rango" es un tipo de error del valor —el parámetro está fuera del rango de los valores aceptables. Por ello esta excepción hereda de la excepción propia de Python ValueError. Esto no es estrictamente necesario (podría heredar directamente de la clase Exception), pero parece más correcto.

2. *Línea 2:* Las excepciones no suelen hacer nada, pero necesitas al menos una línea de código para crear una clase. Al llamar a la sentencia pass conseguimos que no se haga nada y que exista la línea de código necesaria, así que ya tenemos creada la clase de la excepción.

Ahora podemos intentar ejecutar la suite de pruebas de nuevo.

```
1  you@localhost:~/diveintopython3/examples$ python3 romantest2.py -v
2  test_to_roman_known_values (__main__.KnownValues)
3  to_roman should give known result with known input ... ok
4  test_too_large (__main__.ToRomanBadInput)
5  to_roman debería fallar con una entrada muy grande ... FAIL
6
7  ======================================================================
8  FAIL: to_roman debería fallar con una entrada muy grande
9  ----------------------------------------------------------------------
10 Traceback (most recent call last):
11   File "romantest2.py", line 78, in test_too_large
12     self.assertRaises(roman2.OutOfRangeError, roman2.to_roman, 4000)
13 AssertionError: OutOfRangeError not raised by to_roman
14
15 ----------------------------------------------------------------------
16 Ran 2 tests in 0.016s
17
18 FAILED (failures=1)
```

1. *Línea 5:* Aún no pasa la nueva prueba, pero ya no devuelve un error. En su lugar, la prueba falla. ¡Es un avance! Significa que la llamada al método assertRaises() se completó con éxito esta vez, y el entorno de pruebas unitarias realmente comprobó el funcionamiento de la función to_roman().

2. *Línea 13:* Es evidente que la función to_roman() no eleva la excepción OutOfRangeError que acabas de definir, puesto que aún no se lo has dicho. ¡Son

noticias excelentes! significa que es una prueba válida —falla antes de que escribas el código que hace falta para que pueda pasar.

Ahora toca escribir el código que haga pasar esta prueba satisfactoriamente.

```
1  def to_roman(n):
2      '''convert integer to Roman numeral'''
3      if n > 3999:
4          raise OutOfRangeError('number out of range (must be less than 4000)')
5
6      result = ''
7      for numeral, integer in roman_numeral_map:
8          while n >= integer:
9              result += numeral
10             n -= integer
11     return result
```

1. *Línea 4:* Es inmediato: si el valor de entrada (n) es mayor de 3999, eleva la excepción OutOfRangeError. El caso de prueba no valida la cadena de texto que contiene la excepción, aunque podrías escribir otra prueba que hiciera eso (pero ten cuidado con los problemas de internacionalización de cadenas que pueden hacer que varíen en función del idioma del usuario y de la configuración del equipo).

¿Pasará ahora la prueba? veámoslo.

```
1  you@localhost:~/diveintopython3/examples$ python3 romantest2.py -v
2  test_to_roman_known_values (__main__.KnownValues)
3  to_roman should give known result with known input ... ok
4  test_too_large (__main__.ToRomanBadInput)
5  to_roman should fail with large input ... ok
6
7  ----------------------------------------------------------------------
8  Ran 2 tests in 0.000s
9
10 OK
```

1. *Línea 5:* Bien, pasan las dos pruebas. Al haber trabajado iterativamente, yendo y viniendo entre la prueba y el código, puedes estar seguro de que las dos líneas de código que acabas de escribir son las causantes de que una de las pruebas pasara de "fallar" a "pasar". Esta clase de confianza no es gratis, pero revierte por sí misma conforme vas desarrollando más y más código.

9.4. Más paradas, más fuego

Además de probar con números que son demasiado grandes, también es necesario probar con números demasiado pequeños. Como indicamos al comienzo, en los requisitos funcionales, los números romanos no pueden representar ni el cero, ni los números negativos.

```
1 >>> import roman2
2 >>> roman2.to_roman(0)
3 ''
4 >>> roman2.to_roman(-1)
5 ''
```

No está bien, vamos a añadir pruebas para cada una de estas condiciones.

```
1  class ToRomanBadInput(unittest.TestCase):
2      def test_too_large(self):
3          '''to_roman should fail with large input'''
4          self.assertRaises(roman3.OutOfRangeError, roman3.to_roman, 4000)
5
6      def test_zero(self):
7          '''to_roman should fail with 0 input'''
8          self.assertRaises(roman3.OutOfRangeError, roman3.to_roman, 0)
9
10     def test_negative(self):
11         '''to_roman should fail with negative input'''
12         self.assertRaises(roman3.OutOfRangeError, roman3.to_roman, -1)
```

1. *Línea 4:* El método test_too_large() no ha cambiado desde el paso anterior. Se muestra aquí para enseñar el sitio en el que se incorpora el nuevo código.

2. *Línea 8:* Aquí hay una nueva prueba: el método test_zero(). Como el método anterior, indica al método assertRaises(), definido en unittest.TestCase, que llame a nuestra función to_roma() con el parámetro 0, y valida que se eleve la excepción correcta, OutOfRangeError.

3. *Línea 12:* El método test_negative() es casi idéntico, excepto que pasa -1 a la función to_roman(). Si alguna d estas pruebas no eleva una excepción OutOfRangeError (O porque la función retorna un valor o porque eleva otra excepción), se considera que la prueba ha fallado.

Vamos a comprobar que la prueba falla:

```
you@localhost:~/diveintopython3/examples$ python3 romantest3.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_negative (__main__.ToRomanBadInput)
to_roman should fail with negative input ... FAIL
test_too_large (__main__.ToRomanBadInput)
to_roman should fail with large input ... ok
test_zero (__main__.ToRomanBadInput)
to_roman should fail with 0 input ... FAIL

======================================================================
FAIL: to_roman should fail with negative input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest3.py", line 86, in test_negative
    self.assertRaises(roman3.OutOfRangeError, roman3.to_roman, -1)
AssertionError: OutOfRangeError not raised by to_roman

======================================================================
FAIL: to_roman should fail with 0 input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest3.py", line 82, in test_zero
    self.assertRaises(roman3.OutOfRangeError, roman3.to_roman, 0)
AssertionError: OutOfRangeError not raised by to_roman

----------------------------------------------------------------------
Ran 4 tests in 0.000s

FAILED (failures=2)
```

Estupendo, ambas pruebas fallan como se esperaba. Ahora vamos a volver al código a ver lo que podemos hacer para que pasen las pruebas.

```
def to_roman(n):
    '''convert integer to Roman numeral'''
    if not (0 < n < 4000):
        raise OutOfRangeError('number out of range (must be 1..3999)')

    result = ''
    for numeral, integer in roman_numeral_map:
        while n >= integer:
            result += numeral
            n -= integer
    return result
```

1. *Línea 3:* Esta es una forma muy Phytónica de hacer las cosas: dos comparaciones a la vez. Es equivalente a if not ((0¡n) and (n¡400)), pero es mucho más

fácil de leer. Esta línea de código debería capturar todas las entradas que sean demasiado grandes, negativas o cero.

2. *Línea 4:* Si cambias las condiciones, asegúrate de actualizar las cadenas de texto para que coincidan con la nueva condición. Al paquete unittest no le importará, pero será más difícil depurar el código si lanza excepciones que están descritas de forma incorrecta.

Podría mostrarte una serie completa de ejemplos sin relacionar para enseñarte cómo funcionan las comparaciones múltiples, pero en vez de eso, simplemente ejecutaré unas pruebas unitarias y lo probaré.

```
you@localhost:~/diveintopython3/examples$ python3 romantest3.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_negative (__main__.ToRomanBadInput)
to_roman should fail with negative input ... ok
test_too_large (__main__.ToRomanBadInput)
to_roman should fail with large input ... ok
test_zero (__main__.ToRomanBadInput)
to_roman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.016s

OK
```

9.5. Y una cosa más...

Había un requisito funcional más para convertir números a números romanos: tener en cuenta a los números no enteros.

```
>>> import roman3
>>> roman3.to_roman(0.5)
''
>>> roman3.to_roman(1.0)
'I'
```

1. *Línea 2:* ¡Oh! qué mal.

2. *Línea 4:* ¡Oh! incluso peor. Ambos casos deberían lanzar una excepción. En vez de ello, retornan valores falsos.

La validación sobre los no enteros no es difícil. Primero define una excepción NotIntegerError.

```
# roman4.py
class OutOfRangeError(ValueError): pass
class NotIntegerError(ValueError): pass
```

Lo siguiente es escribir un caso de prueba que compruebe si se lanza una excepción NotIntegerError.

```
class ToRomanBadInput(unittest.TestCase):
    .
    .
    .
    def test_non_integer(self):
        '''to_roman should fail with non-integer input'''
        self.assertRaises(roman4.NotIntegerError, roman4.to_roman, 0.5)
```

Ahora vamos a validar si la prueba falla apropiadamente.

```
you@localhost:~/diveintopython3/examples$ python3 romantest4.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_negative (__main__.ToRomanBadInput)
to_roman should fail with negative input ... ok
test_non_integer (__main__.ToRomanBadInput)
to_roman should fail with non-integer input ... FAIL
test_too_large (__main__.ToRomanBadInput)
to_roman should fail with large input ... ok
test_zero (__main__.ToRomanBadInput)
to_roman should fail with 0 input ... ok

======================================================================
FAIL: to_roman should fail with non-integer input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest4.py", line 90, in test_non_integer
    self.assertRaises(roman4.NotIntegerError, roman4.to_roman, 0.5)
AssertionError: NotIntegerError not raised by to_roman

----------------------------------------------------------------------
Ran 5 tests in 0.000s

FAILED (failures=1)
```

Escribe ahora el código que haga que se pase la prueba.

```
1  def to_roman(n):
2      '''convert integer to Roman numeral'''
3      if not (0 < n < 4000):
4          raise OutOfRangeError('number out of range (must be 1..3999)')
5      if not isinstance(n, int):
6          raise NotIntegerError('non-integers can not be converted')
7
8      result = ''
9      for numeral, integer in roman_numeral_map:
10         while n >= integer:
11             result += numeral
12             n -= integer
13     return result
```

1. *Línea 5:* La función interna de Python isinstance() comprueba si una variable es de un determinado tipo (o técnicamente, de cualquier tipo descendiente).

2. *Línea 6:* Si el parámetro n no es int elevará la nueva excepción NotIntegerError.

Por último, vamos a comprobar si realmente el código pasa las pruebas.

```
1  you@localhost:~/diveintopython3/examples$ python3 romantest4.py -v
2  test_to_roman_known_values (__main__.KnownValues)
3  to_roman should give known result with known input ... ok
4  test_negative (__main__.ToRomanBadInput)
5  to_roman should fail with negative input ... ok
6  test_non_integer (__main__.ToRomanBadInput)
7  to_roman should fail with non-integer input ... ok
8  test_too_large (__main__.ToRomanBadInput)
9  to_roman should fail with large input ... ok
10 test_zero (__main__.ToRomanBadInput)
11 to_roman should fail with 0 input ... ok
12
13 ----------------------------------------------------------------------
14 Ran 5 tests in 0.000s
15
16 OK
```

La función to_roman pasa todas las pruebas y no se me ocurren nuevas pruebas, por lo que es el momento de pasar a la función from_roman()

9.6. Una agradable simetría

Convertir una cadena de texto que representa un número romano a entero parece más difícil que convertir un entero en un número romano. Es cierto que existe

el tema de la validación. Es fácil validar si un número entero es mayor que cero, pero un poco más difícil comprobar si una cadena es un número romano válido. Pero ya habíamos construido una expresión regular para comprobar números romanos, por lo que esa parte está hecha.

Eso nos deja con el problema de convertir la cadena de texto. Como veremos en un minuto, gracias a la rica estructura de datos que definimos para mapear los números romanos a números enteros, el núcleo de la función from_roman() es tan simple como el de la función to_roman().

Pero primero hacemos las puertas. Necesitaremos una prueba de valores válidos conocidos para comprobar la precisión de la función. Nuestro juego de pruebas ya contiene una mapa de valores conocidos; vamos a reutilizarlos.

```
...

    def test_from_roman_known_values(self):
        '''from_roman should give known result with known input'''
        for integer, numeral in self.known_values:
            result = roman5.from_roman(numeral)
            self.assertEqual(integer, result)
...
```

Existe una agradable simetría aquí. Las funciones to_roman() y from_roman() son la inversa una de la otra. La primera convierte números enteros a cadenas formateadas que representan números romanos, la segunda convierte cadenas de texto que representan a números romanos a números enteros. En teoría deberíamos ser capaces de hacer ciclos con un número pasándolo a la función to_roman() para recuperar una cadena de texto, luego pasar esa cadena a la función from_roman() para recuperar un número entero y finalizar con el mismo número que comenzamos.

```
n = from_roman(to_roman(n)) for all values of n
```

En este caso "all values" significa cualquier número entre el 1..3999, puesto que es el rango válido de entradas a la función to_roman(). Podemos expresar esta simetría en un caso de prueba que recorrar todos los valores 1..3999, llamar a to_roman(), llamar a from_roman() y comprobar que el resultado es el mismo que la entrada original.

```
class RoundtripCheck(unittest.TestCase):
    def test_roundtrip(self):
        '''from_roman(to_roman(n))==n for all n'''
        for integer in range(1, 4000):
            numeral = roman5.to_roman(integer)
            result = roman5.from_roman(numeral)
            self.assertEqual(integer, result)
```

Estas pruebas ni siquiera fallarán. No hemos definido aún la función from_roman() por lo que únicamente se elevarán errores.

```
you@localhost:~/diveintopython3/examples$ python3 romantest5.py
E.E....
======================================================================
ERROR: test_from_roman_known_values (__main__.KnownValues)
from_roman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest5.py", line 78, in test_from_roman_known_values
    result = roman5.from_roman(numeral)
AttributeError: 'module' object has no attribute 'from_roman'

======================================================================
ERROR: test_roundtrip (__main__.RoundtripCheck)
from_roman(to_roman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest5.py", line 103, in test_roundtrip
    result = roman5.from_roman(numeral)
AttributeError: 'module' object has no attribute 'from_roman'

----------------------------------------------------------------------
Ran 7 tests in 0.019s

FAILED (errors=2)
```

Una función vacía resolverá este problema:

```
# roman5.py
def from_roman(s):
    '''convert Roman numeral to integer'''
```

(¿Te has dado cuenta? He definido una función simplemente poniendo un docstring. Esto es válido en Python. De hecho, algunos programadores lo toman al pie de la letra. "No crees una función vacía, ¡documéntala!")

Ahora los casos de prueba sí que fallarán.

```
you@localhost:~/diveintopython3/examples$ python3 romantest5.py
F.F....
======================================================================
FAIL: test_from_roman_known_values (__main__.KnownValues)
from_roman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest5.py", line 79, in test_from_roman_known_values
    self.assertEqual(integer, result)
AssertionError: 1 != None

======================================================================
FAIL: test_roundtrip (__main__.RoundtripCheck)
from_roman(to_roman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest5.py", line 104, in test_roundtrip
    self.assertEqual(integer, result)
AssertionError: 1 != None

----------------------------------------------------------------------
Ran 7 tests in 0.002s

FAILED (failures=2)
```

Ahora ya podemos escribir la función from_roman().

```
def from_roman(s):
    """convert Roman numeral to integer"""
    result = 0
    index = 0
    for numeral, integer in roman_numeral_map:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

1. *Línea 6:* El patrón aquí es el mismo que el de la función to_roman(). Iteras a través de la estructura de datos de números romanos (la tupla de tuplas), pero en lugar de encontrar el mayor número entero tantas veces como sea posible, compruebas coincidencias del carácter romano más "alto" tantas veces como sea posible.

Si no te queda claro cómo funciona from_roman() añade una sentencia print al final del bucle while:

```
def from_roman(s):
    """convert Roman numeral to integer"""
    result = 0
    index = 0
    for numeral, integer in roman_numeral_map:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
            print('found', numeral, 'of length', len(numeral), ', adding', integer)
>>> import roman5
>>> roman5.from_roman('MCMLXXII')
found M of length 1, adding 1000
found CM of length 2, adding 900
found L of length 1, adding 50
found X of length 1, adding 10
found X of length 1, adding 10
found I of length 1, adding 1
found I of length 1, adding 1
1972
```

Es el momento de volver a ejecutar las pruebas.

```
you@localhost:~/diveintopython3/examples$ python3 romantest5.py
.......
----------------------------------------------------------------------
Ran 7 tests in 0.060s

OK
```

Tenemos dos buenas noticias aquí. Por un lado que la función from_roman() pasa las pruebas ante entradas válidas, al menos para los valores conocidos. Por otro, la prueba de ida y vuelta también ha pasado. Combinada con las pruebas de valores conocidos, puedes estar razonablemente seguro de que ambas funciones to_roman() y from_roman() funcionan correctamente para todos los valores válidos (Esto no está garantizado: es teóricamente posible que la función to_roman() tuviera un fallo que produjera un valor erróneo de número romano y que la función from_roman() tuviera un fallo recíproco que produjera el mismo valor erróneo como número entero. Dependiendo de la aplicación y de los requisitos, esto puede ser más o menos preocupante; si es así, hay que escribir un conjunto de casos de prueba mayor hasta que puedas quedar razonablemente traquilo en cuanto a la fiabilidad del código desarrollado).

9.7. Más entradas erróneas

Ahora que la función from_roman() funciona correctamente con entradas válidas es el momento de poner la última pieza del puzzle: hacer que funcione correctamente con entradas incorrectas. Eso significa encontrar un modo de mirar una cadena para determinar si es un número romano correcto. Esto es inherentemente más difícil que validar las entradas numéricas de la función to_roman(), pero dispones de una poderosa herramienta: las expresiones regulares (si no estás familiarizado con ellas, ahora es un buen momento para leer el capítulo sobre las expresiones regulares).

Como viste en el caso de estudio: números romanos, existen varias reglas simples para construir un número romano utilizando las letras M, D, C, L, X, V e I. Vamos a revisar las reglas:

- Algunas veces los caracteres son aditivos, I es 1, II es 2 y III es 3. VI es 6 (literalmente 5 + 1), VII es 7 (5+1+1) y XVIII es 18 (10+5+1+1+1).
- Los caracteres que representan unidades, decenas, centenas y unidades de millar (I, X, C y M) pueden aparecer juntos hasta tres veces como máximo. Para el 4 debes restar del carácter V, L ó D (cinco, cincuenta, quinientos) que se encuentre más próximo a la derecha. No se puede representar el cuatro como IIII, en su lugar hay que poner IV (5-1). El número 40 se representa como XL (10 menos que 50: 50-10). 41 = XLI, 42 = XLII, 43 = XLIII y luego 44 = XLIV (diez menos que cincuenta más uno menos que cinco: 50-10+5-1).
- De forma similar, para el número 9, debes restar del número siguiente más próximo que represente unidades, decenas, centenas o unidades de millar (I, X, C y M). 8 = VIII, pero 9 = IX (1 menos que 10), no 9 = VIIII puesto que el carácter I no puede repetirse cuatro veces seguidas. El número 90 se representa con XC y el 900 con CM.
- Los caracteres V, L y D no pueden repetirse; el número 10 siempre se representa como X y no como VV. El número 100 siempre se representa como C y nunca como LL.
- Los números romanos siempre se escriben de los caracteres que representan valores mayores a los menores y se leen de izquierda a derecha por lo que el orden de los caracteres importa mucho. {DC es el número 600; CD otro número, el 400 (500 - 100). CI es 101, mientras que IC no es un número romano válido porque no puedes restar I del C[2].

[2]Para representar el 99 deberías usar: XCIL (100 - 10 + 10 - 1)

Así, una prueba apropiada podría ser asegurarse de que la función from_roman() falla cuando pasas una cadena que tiene demasiados caracteres romanos repetidos. Pero, ¿cuánto es "demasiados'? ...depende del carácter.

```
class FromRomanBadInput(unittest.TestCase):
    def test_too_many_repeated_numerals(self):
        '''from_roman should fail with too many repeated numerals'''
        for s in ('MMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):
            self.assertRaises(roman6.InvalidRomanNumeralError,
                              roman6.from_roman, s)
```

Otra prueba útil sería comprobar que ciertos patrones no están repetidos. Por ejemplo, IX es 9, pero IXIX no es válido nunca.

```
...
 def test_repeated_pairs(self):
        '''from_roman should fail with repeated pairs of numerals'''
        for s in ('CMCM', 'CDCD', 'XCXC', 'XLXL', 'IXIX', 'IVIV'):
            self.assertRaises(roman6.InvalidRomanNumeralError,
                              roman6.from_roman, s)
...
```

Una tercera prueba podría comprobar que los caracteres aparecen en el orden correcto, desde el mayor al menor. Por ejemplo, CL es 150, pero LC nunca es válido, porque el carácter para el 50 nunca puede aparecer antes del carácter del 100. Esta prueba incluye un conjunto no válido de conjuntos de caracteres elegidos aleatoriamente: I antes que M, V antes que X, y así sucesivamente.

```
...
def test_malformed_antecedents(self):
        '''from_roman should fail with malformed antecedents'''
        for s in ('IIMXCC', 'VX', 'DCM', 'CMM', 'IXIV',
                  'MCMC', 'XCX', 'IVI', 'LM', 'LD', 'LC'):
            self.assertRaises(roman6.InvalidRomanNumeralError,
                              roman6.from_roman, s)
...
```

Cada una de estas pruebas se basan en que la función from_roman() eleve una excepción, InvalidRomanNumeralError, que aún no hemos definido.

```
# roman6.py
class InvalidRomanNumeralError(ValueError): pass
```

Las tres pruebas deberían fallar, puesto que from_roman() aún no efectúa ningún tipo de validación de la entrada (Si no fallan ahora, ¿qué demonios están comprobando?).

```
you@localhost:~/diveintopython3/examples$ python3 romantest6.py
FFF.......
======================================================================
FAIL: test_malformed_antecedents (__main__.FromRomanBadInput)
from_roman should fail with malformed antecedents
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest6.py", line 113, in test_malformed_antecedents
    self.assertRaises(roman6.InvalidRomanNumeralError,
                      roman6.from_roman, s)
AssertionError: InvalidRomanNumeralError not raised by from_roman

======================================================================
FAIL: test_repeated_pairs (__main__.FromRomanBadInput)
from_roman should fail with repeated pairs of numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest6.py", line 107, in test_repeated_pairs
    self.assertRaises(roman6.InvalidRomanNumeralError,
                      roman6.from_roman, s)
AssertionError: InvalidRomanNumeralError not raised by from_roman

======================================================================
FAIL: test_too_many_repeated_numerals (__main__.FromRomanBadInput)
from_roman should fail with too many repeated numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest6.py", line 102, in test_too_many_repeated_numerals
    self.assertRaises(roman6.InvalidRomanNumeralError,
                      roman6.from_roman, s)
AssertionError: InvalidRomanNumeralError not raised by from_roman

----------------------------------------------------------------------
Ran 10 tests in 0.058s

FAILED (failures=3)
```

Está bien. Ahora lo que necesitamos es añadir la expresión regular para comprobar números romanos válidos en la función from_roman().

```
roman_numeral_pattern = re.compile('''
    ^                   # beginning of string
    M{0,3}              # thousands - 0 to 3 Ms
    (CM|CD|D?C{0,3})    # hundreds - 900 (CM), 400 (CD), 0-300 (0 to 3 Cs),
                        #            or 500-800 (D, followed by 0 to 3 Cs)
    (XC|XL|L?X{0,3})    # tens - 90 (XC), 40 (XL), 0-30 (0 to 3 Xs),
                        #        or 50-80 (L, followed by 0 to 3 Xs)
    (IX|IV|V?I{0,3})    # ones - 9 (IX), 4 (IV), 0-3 (0 to 3 Is),
                        #        or 5-8 (V, followed by 0 to 3 Is)
    $                   # end of string
    ''', re.VERBOSE)

def from_roman(s):
    '''convert Roman numeral to integer'''
    if not roman_numeral_pattern.search(s):
        raise InvalidRomanNumeralError(
                'Invalid Roman numeral: {0}'.format(s))

    result = 0
    index = 0
    for numeral, integer in roman_numeral_map:
        while s[index : index + len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

Y ahora a ejecutar de nuevo las pruebas...

```
you@localhost:~/diveintopython3/examples$ python3 romantest7.py
..........
----------------------------------------------------------------------
Ran 10 tests in 0.066s

OK
```

Y el premio al anticlimax del año va para... la palabra "OK", que se imprime por parte del módulo unittest cuando todas pruebas pasan correctamente.

Capítulo 10

Refactorizar

Nivel de dificultad:◆ ◆ ◆ ◆ ◇

"Despues de haber tocado una vasta cantidad de notas y más notas.
es la simplicidad la que emerge como la recompensa coronada del arte."
—Frédéric Chopin

10.1. Inmersión

A pesar tus mejores esfuerzos para escribir un conjunto amplio de pruebas uitarias, tendrás errores (bugs). ¿Qué significa que haya un error? Que nos falta un caso de prueba por escribir.

```
1 >>> import roman7
2 >>> roman7.from_roman('')
3 0
```

1. *Línea 2:* Esto es un error. Una cadena vacía debería elevar la excepción InvalidRomanNumeralError, como cualquier otra secuencia de caracteres que no represente a un número romano válido.

Después de reproducir el error, y antes de arreglarlo, deberías escribir un caso de prueba que falle para ilustrar el error.

```
class FromRomanBadInput(unittest.TestCase):
    .
    .
    .
    def testBlank(self):
        '''from_roman should fail with blank string'''
        self.assertRaises(roman6.InvalidRomanNumeralError,
            roman6.from_roman, '')
```

1. *Línea 7:* Es muy simple. La llamada a from_roman() con una cadena vacía debe asegurar que eleva la excepción InvalidRomanNumeralError. La parte más difícil fue encontrar el error; ahora que lo conoces, crear una prueba que lo refleje es lo fácil.

Puesto que el código tiene un fallo, y dispones de un caso de prueba que comprueba que existe, el caso de prueba fallará:

```
you@localhost:~/diveintopython3/examples$ python3 romantest8.py -v
from_roman should fail with blank string ... FAIL
from_roman should fail with malformed antecedents ... ok
from_roman should fail with repeated pairs of numerals ... ok
from_roman should fail with too many repeated numerals ... ok
from_roman should give known result with known input ... ok
to_roman should give known result with known input ... ok
from_roman(to_roman(n))==n for all n ... ok
to_roman should fail with negative input ... ok
to_roman should fail with non-integer input ... ok
to_roman should fail with large input ... ok
to_roman should fail with 0 input ... ok

======================================================================
FAIL: from_roman should fail with blank string
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest8.py", line 117, in test_blank
    self.assertRaises(roman8.InvalidRomanNumeralError, roman8.from_roman, '')
AssertionError: InvalidRomanNumeralError not raised by from_roman

----------------------------------------------------------------------
Ran 11 tests in 0.171s

FAILED (failures=1)
Now you can fix the bug.

skip over this code listing

[hide] [open in new window]
```

Ahora puedes arreglar el fallo.

```
1  def from_roman(s):
2      '''convert Roman numeral to integer'''
3      if not s:
4          raise InvalidRomanNumeralError('Input can not be blank')
5      if not re.search(romanNumeralPattern, s):
6          raise InvalidRomanNumeralError(
7                  'Invalid Roman numeral: {}'.format(s))
8
9      result = 0
10     index = 0
11     for numeral, integer in romanNumeralMap:
12         while s[index:index+len(numeral)] == numeral:
13             result += integer
14             index += len(numeral)
15     return result
```

1. *Línea 3:* Solamente se necesitan dos líneas de código: una comprobación explícita por la cadena de texto vacía y la sentencia raise.

2. *Línea 6:* Creo que no lo he mencionado en ninguna otra parte del libro, por lo que te puede servir como tu última lección en formateo de cadenas de texto. Desde Python 3.1 puedes dejar de poner los valors numéricos cuando utilizas índices posicionales en el especificador de formatos. En lugar de utilizar el especificador de formato {0} para indicar el primer parámetro del método format(), puedes utilizar directamente {} y Python lo sustituirá por el índice posicional apropiado. Esto funciona para cualquier número de parámetros; el primer {} equivale a {0}, el segundo {} a {1} y así sucesivamente.

```
1  you@localhost:~/diveintopython3/examples$ python3 romantest8.py -v
2  from_roman should fail with blank string ... ok
3  from_roman should fail with malformed antecedents ... ok
4  from_roman should fail with repeated pairs of numerals ... ok
5  from_roman should fail with too many repeated numerals ... ok
6  from_roman should give known result with known input ... ok
7  to_roman should give known result with known input ... ok
8  from_roman(to_roman(n))==n for all n ... ok
9  to_roman should fail with negative input ... ok
10 to_roman should fail with non-integer input ... ok
11 to_roman should fail with large input ... ok
12 to_roman should fail with 0 input ... ok
13
14 ----------------------------------------------------------------------
15 Ran 11 tests in 0.156s
16
17 OK
```

Ahora la prueba de la cadena vacía pasa sin problemas, por lo que el error está arreglado. Además, todas las demás pruebas siguen funcionando, lo que significa que la reparación del error no rompe nada de lo que ya funcionaba, para lo que disponemos de pruebas previas. Se ha acabado el trabajo.

Codificar de esta forma no hace que sea más sencillo arreglar los errores. Los errores sencillos (como este caso) requieren casos de prueba sencillos: los errores complejos requerirán casos de prueba complejos. A primera vista, puede parecer que llevará más tiempo reparar un error en un entorno de desarrollo orientado a pruebas, puesto que necesitas expresar en código aquello que refleja el error (para poder escribir el caso de prueba) y luego arreglar el error propiamente dicho. Lo luego si el caso de prueba no pasa satisfactoriamente necesitas arreglar lo que sea que esté roto o sea erróneo o, incluso, que el caso de prueba tenga un error en sí mismo. Sin embargo, a largo plazo, esta forma de trabajar entre el código y la prueba resulta rentable, porque hace más probable que los errores sean arreglados correctamente sin romper otra cosa. Una caso de prueba escrito hoy, es una prueba de regresión para el día de mañana.

10.2. Gestionar requisitos cambiantes

A pesar de tus mayores esfuerzos para "clavar" a la tierra a tus clientes e identificar los requisitos exactos a fuerza de hacerles cosas horribles con tijeras y cera caliente, los requisitos cambiarán. La mayoría de los clientes no saben lo que quieren hasta que lo ven, e incluso aunque lo supieran, no es fácil para ellos articular de forma precisa lo que quieren de forma que nos sea útil a los desarrolladores. E incluso en ese caso, querrán más cosas para la siguiente versión del proyecto. Por eso, prepárate para actualizar los casos de prueba según van cambiando los requisitos.

Imagina, por ejemplo, que querías ampliar el rango de las funciones de conversión de números romanos. Normalmente, ningún carácter de un número romano se puede repetir más de tres veces seguidas. Pero los romano estaban deseando hacer una excepción a a la regla para poder reflejar el número `4000` con cuatro `M` seguidas. Si haces este cambio, será posible ampliar el rango de valores válidos en romano del `1..3999` al `1..4999`. Pero primero, necesitas modificar los casos de prueba.

```
1  class KnownValues(unittest.TestCase):
2      known_values = ( (1, 'I'),
3                       .
4                       .
5                       .
6                      (3999, 'MMMCMXCIX'),
7                      (4000, 'MMMM'),
8                      (4500, 'MMMMD'),
9                      (4888, 'MMMMDCCCLXXXVIII'),
10                     (4999, 'MMMMCMXCIX') )
11
12 class ToRomanBadInput(unittest.TestCase):
13     def test_too_large(self):
14         '''to_roman should fail with large input'''
15         self.assertRaises(roman8.OutOfRangeError, roman8.to_roman, 5000)
16
17     .
18     .
19     .
20
21 class FromRomanBadInput(unittest.TestCase):
22     def test_too_many_repeated_numerals(self):
23         '''from_roman should fail with too many repeated numerals'''
24         for s in ('MMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):
25             self.assertRaises(roman8.InvalidRomanNumeralError,
26                               roman8.from_roman, s)
27
28     .
29     .
30     .
31
32 class RoundtripCheck(unittest.TestCase):
33     def test_roundtrip(self):
34         '''from_roman(to_roman(n))==n for all n'''
35         for integer in range(1, 5000):
36             numeral = roman8.to_roman(integer)
37             result = roman8.from_roman(numeral)
38             self.assertEqual(integer, result)
```

1. *Línea 7:* Los valores existentes no cambian (aún son valores razonables para probar), pero necesitas añadir unos cuantos en el rango de 4000. He incluido el 4000 (el más corto), 4500 (el segundo más corto), 4888 (el más largo) y el 4999 (el mayor).

2. *Línea 15:* La definición de la "entrada mayor" ha cambiado. Esta prueba se usaba para probar la función to_roman() con el número 4000 y esperar un error;

ahora que los números del rango 4000-4999 son válidos, necesitamos subir el valor de la prueba al 5000.

3. *Línea 24:* La definición de "demasiados caracteres repetidos" también ha cambiado. Esta prueba llamaba a la función from_roman() con la cadena "M" y esperaba un error; ahora que MMMM se considera un número romano válido, necesitamos elevar el valor a "MMMMM".

4. *Línea 35:* El ciclo de prueba a través del rango completo de valores del número 1 al 3999 también hay que cambiarlo. Puesto que el rango se ha expandido es necesario actualizar el bucle para que llegue hasta el 4999.

Ahora los casos de prueba están actualizados con los nuevos requisitos; pero el código no lo está, por lo que hay que esperar que los casos de prueba fallen.

```
you@localhost:~/diveintopython3/examples$ python3 romantest9.py -v
from_roman should fail with blank string ... ok
from_roman should fail with malformed antecedents ... ok
from_roman should fail with non-string input ... ok
from_roman should fail with repeated pairs of numerals ... ok
from_roman should fail with too many repeated numerals ... ok
from_roman should give known result with known input ... ERROR
to_roman should give known result with known input ... ERROR
from_roman(to_roman(n))==n for all n ... ERROR
to_roman should fail with negative input ... ok
to_roman should fail with non-integer input ... ok
to_roman should fail with large input ... ok
to_roman should fail with 0 input ... ok

======================================================================
ERROR: from_roman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest9.py", line 82, in test_from_roman_known_values
    result = roman9.from_roman(numeral)
  File "C:\home\diveintopython3\examples\roman9.py", line 60, in from_roman
    raise InvalidRomanNumeralError('Invalid Roman numeral: {0}'.format(s))
roman9.InvalidRomanNumeralError: Invalid Roman numeral: MMMM

======================================================================
ERROR: to_roman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest9.py", line 76, in test_to_roman_known_values
    result = roman9.to_roman(integer)
  File "C:\home\diveintopython3\examples\roman9.py", line 42, in to_roman
    raise OutOfRangeError('number out of range (must be 0..3999)')
roman9.OutOfRangeError: number out of range (must be 0..3999)

======================================================================
ERROR: from_roman(to_roman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest9.py", line 131, in testSanity
    numeral = roman9.to_roman(integer)
  File "C:\home\diveintopython3\examples\roman9.py", line 42, in to_roman
    raise OutOfRangeError('number out of range (must be 0..3999)')
roman9.OutOfRangeError: number out of range (must be 0..3999)

----------------------------------------------------------------------
Ran 12 tests in 0.171s

FAILED (errors=3)
```

1. *Línea 6:* La prueba de from_roman() sobre valores conocidos falla en cuanto encuentra MMMM, puesto que from_roman() aún cree que este número no es válido.

2. *Línea 7:* La prueba de valores conocidos de to_roman() falla en cuanto encuentra 4000, puesto que to_roman() aún piensa que este valor está fuera de rango.

3. *Línea 8:* La prueba completa de "ida y vuelta" también fallará en cuanto encuentre el valor 4000, porque to_roman() aún piensa que está fuera de rango.

Ahora que tienes casos de prueba que fallan debido a los nuevos requisitos, puedes abordar la modificación del código para incorporar los mismos de forma que se superen los casos de prueba (Cuando comienzas a trabajar de forma orientada a pruebas, puede parecer extraño al principio que el código que se va a probar nunca va "por delante" de los casos de prueba. Mientras está "por detrás" te queda trabajo por hacer, en cuanto el código alcanza a los casos de prueba, has terminado de codificar. Una vez te acostumbres, de preguntarás cómo es posible que hayas estado programando en el pasado sin pruebas).

```
1  roman_numeral_pattern = re.compile('''
2      ^                   # beginning of string
3      M{0,4}              # thousands - 0 to 4 Ms
4      (CM|CD|D?C{0,3})    # hundreds - 900 (CM), 400 (CD), 0-300 (0 to 3 Cs),
5                          #            or 500-800 (D, followed by 0 to 3 Cs)
6      (XC|XL|L?X{0,3})    # tens - 90 (XC), 40 (XL), 0-30 (0 to 3 Xs),
7                          #        or 50-80 (L, followed by 0 to 3 Xs)
8      (IX|IV|V?I{0,3})    # ones - 9 (IX), 4 (IV), 0-3 (0 to 3 Is),
9                          #        or 5-8 (V, followed by 0 to 3 Is)
10     $                   # end of string
11     ''', re.VERBOSE)
12
13 def to_roman(n):
14     '''convert integer to Roman numeral'''
15     if not (0 < n < 5000):
16         raise OutOfRangeError('number out of range (must be 1..4999)')
17     if not isinstance(n, int):
18         raise NotIntegerError('non-integers can not be converted')
19
20     result = ''
21     for numeral, integer in roman_numeral_map:
22         while n >= integer:
23             result += numeral
24             n -= integer
25     return result
26
27 def from_roman(s):
28     .
29     .
30     .
```

1. *Línea 3:* No necesitas hacer ningún cambio a la función from_roman(). Lo único que hay que modificar es roman_numeral_pattern. Si observas detenidamente, lo primero que notarás es que he cambiado el valor máximo de caracteres M opcionales, para poner 4 donde antes había 3. Esto permitirá la existencia de valores romanos de 4999. La función from_roman() es totalmente genérica. simplemente busca caracteres romanos y los va acumulando, sin preocuparse sobre las veces que se repite. La única razón por la que no permitía MMMM es que lo impedíamos expresamente en la comprobación contra la expresión regular.

2. *Línea 15:* La función to_roman() solamente necesita un pequeño cambio en el rango de validación. Donde se comprobaba que el valor se encontrase en 0 ¡n ¡4000, ahora se comprueba que se cumpla 0 ¡n ¡5000. Y se modifica el mensaje de error que se eleva para reflejar el nuevo rango (1...4999 en lugar de 1...3999). No necesitas realizar más cambios a la función (Es capaz de

añadir una M por cada millar que encuentra. La única razón por la que antes no funcionaba con 4000 es que la validación del rango de valores válidos lo impedía explícitamente).

Puede ser que estés algo escéptico sobre que estos dos pequeños cambios sean todo lo que necesitas. Vale, no me creas, obsérvalo por ti mismo.

```
you@localhost:~/diveintopython3/examples$ python3 romantest9.py -v
from_roman should fail with blank string ... ok
from_roman should fail with malformed antecedents ... ok
from_roman should fail with non-string input ... ok
from_roman should fail with repeated pairs of numerals ... ok
from_roman should fail with too many repeated numerals ... ok
from_roman should give known result with known input ... ok
to_roman should give known result with known input ... ok
from_roman(to_roman(n))==n for all n ... ok
to_roman should fail with negative input ... ok
to_roman should fail with non-integer input ... ok
to_roman should fail with large input ... ok
to_roman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 12 tests in 0.203s

OK
```

Pasan todas las pruebas, paramos de codificar.

Un conjunto amplio y exhaustivo de pruebas significa que nunca tienes que depender de un programador que dice "Confía en mi".

10.3. Rectorización

Lo mejor de disponer de un conjunto de pruebas exhaustivo no es la sensación agradable que se obtiene cuando todas las pruebas pasan satisfactoriamente,ni la sensación de éxito cuando alguien te culpa de romper su código y *pruebas* que no has sido tú. Lo mejor de las pruebas unitarias es que te dan la libertad de refactorizar el código sin "piedad".

La refactorización es el proceso por el que se toma código que funciona correctamente y se le hace funcionar mejor. Normalmente "mejor" significa "más rápido", aunque también puede significar "usando menos memoria" o "usando menos disco" o simplemente "de forma más elegante". Independientemente de lo que signifique para ti en tu entorno, refactorizar es importante para la salud a largo plazo de un programa.

Aquí "mejor" significa dos cosas a la vez: "más rápido" y "más fácil de mantener". Específicamente, la función from_roman() es más lenta y compleja de lo que me gustaría debido a la fea expresión regular que se utiliza para validar los números romanos. Podrías pensar "de acuerdo, la expresión regular es grande y compleja, pero cómo si no voy a validar que una cadena arbitraria es un número romano válido.

Respuesta, únicamente existen 5000, ¿porqué no construir una tabla de búsqueda? Esta idea se hace cada vez más evidente cuando se observa que así no hay que utilizar expresiones regulares. Según se construye una tabla de búsqueda de números romanos para convertir enteros en romanos, se puede construir una tabla inversa para convertir de romanos a enteros. Así, cuando tengas que comprobar si una cadena de texto es un número romano válido ya dispondrás de los números válidos y "validar" queda reducido a mirar en un diccionario de búsqueda.

Y lo mejor de todo es que ya dispones de un buen conjunto de pruebas unitarias. Puedes modificar la mitad del código del módulo, pero las pruebas seguirán siendo las mismas. Esto significa que puedes comprobar —a ti mismo y a los demás— que el nuevo código funciona tan bien como el original.

```
class OutOfRangeError(ValueError): pass
class NotIntegerError(ValueError): pass
class InvalidRomanNumeralError(ValueError): pass

roman_numeral_map = (('M',  1000),
                     ('CM', 900),
                     ('D',  500),
                     ('CD', 400),
                     ('C',  100),
                     ('XC', 90),
                     ('L',  50),
                     ('XL', 40),
                     ('X',  10),
                     ('IX', 9),
                     ('V',  5),
                     ('IV', 4),
                     ('I',  1))

to_roman_table = [ None ]
from_roman_table = {}

def to_roman(n):
    '''convert integer to Roman numeral'''
    if not (0 < n < 5000):
        raise OutOfRangeError('number out of range (must be 1..4999)')
    if int(n) != n:
        raise NotIntegerError('non-integers can not be converted')
    return to_roman_table[n]

def from_roman(s):
    '''convert Roman numeral to integer'''
    if not isinstance(s, str):
        raise InvalidRomanNumeralError('Input must be a string')
    if not s:
        raise InvalidRomanNumeralError('Input can not be blank')
    if s not in from_roman_table:
        raise InvalidRomanNumeralError(
                    'Invalid Roman numeral: {0}'.format(s))
    return from_roman_table[s]

def build_lookup_tables():
    def to_roman(n):
        result = ''
        for numeral, integer in roman_numeral_map:
            if n >= integer:
                result = numeral
                n -= integer
                break
        if n > 0:
            result += to_roman_table[n]
        return result

    for integer in range(1, 5000):
        roman_numeral = to_roman(integer)
        to_roman_table.append(roman_numeral)
        from_roman_table[roman_numeral] = integer

build_lookup_tables()
```

Vamos a trocear el código anterior en partes fáciles de digerir. La línea más importante es la última.

```
1 build_lookup_tables()
```

Se trata de una llamada a función pero no está rodeada de una sentencia if. No es un bloque if __name__ == "__main__", esta función se llama cuando el módulo se importa (Es importante conocer que los módulos únicamente se importan una vez, cuando se cargan en memoria la primera vez que se usan. Si importas de nuevo un módulo ya cargado, Python no hace nada. Por eso esta función únicamente se ejecuta la primera vez que importas este módulo.

¿Qué es lo que hace la función build_lookup_tables()? me alegor de que me lo preguntes.

```
1  to_roman_table = [ None ]
2  from_roman_table = {}
3  .
4  .
5  .
6  def build_lookup_tables():
7      def to_roman(n):
8          result = ''
9          for numeral, integer in roman_numeral_map:
10             if n >= integer:
11                 result = numeral
12                 n -= integer
13                 break
14         if n > 0:
15             result += to_roman_table[n]
16         return result
17
18     for integer in range(1, 5000):
19         roman_numeral = to_roman(integer)
20         to_roman_table.append(roman_numeral)
21         from_roman_table[roman_numeral] = integer
```

1. *Línea 7:* Este es un trozo muy inteligente de programa, tal vez demasiado. La función to_roman() se define en primer lugar; busca valores en la tabla de búsqueda y las retorna. Pero la función build_lookup_tables() redefine la función para que haga algo (como en el ejemplo anterior, antes de que añadieras una tabla de búsqueda). Dentro de la función build_lookup_tables() se llama a la función to_roman(9 redefinida en la función. Una vez la función build_lookup_tables() finaliza, la versión redefinida desaparece —solamente se define en el ámbito local de la función build_lookup_tables().

2. *Línea 19:* Esta línea de código llamará a la función to_roman() redefinida, la que realmente calcula el número romano.

3. *Línea 20:* Una vez dispones del resultado (de la función to_roman() redefinida), añaders el valor entero y su equivalente romano a las dos tablas de búsqueda.

Una vez construidas ambas tablas, el resto del código es simple y rápido.

```
1  def to_roman(n):
2      '''convert integer to Roman numeral'''
3      if not (0 < n < 5000):
4          raise OutOfRangeError('number out of range (must be 1..4999)')
5      if int(n) != n:
6          raise NotIntegerError('non-integers can not be converted')
7      return to_roman_table[n]
8
9  def from_roman(s):
10     '''convert Roman numeral to integer'''
11     if not isinstance(s, str):
12         raise InvalidRomanNumeralError('Input must be a string')
13     if not s:
14         raise InvalidRomanNumeralError('Input can not be blank')
15     if s not in from_roman_table:
16         raise InvalidRomanNumeralError(
17                 'Invalid Roman numeral: {0}'.format(s))
18     return from_roman_table[s]
```

1. *Línea 7:* Después de efectuar las validaciones de rango, la función to_roman() únicamente tiene que encontrar el valor apropiado en la tabla de búsqueda y devolverlo.

2. *Línea 17:* De forma similar, la función from_roman() queda reducida a algunas validaciones de límites y una línea de código. No hay expresiones regulares. No hay bucles. Solamente la conversión desde y hacia números romanos.

Pero ¿funciona?, sí, sí, funciona. Y puedo probarlo.

```
you@localhost:~/diveintopython3/examples$ python3 romantest10.py -v
from_roman should fail with blank string ... ok
from_roman should fail with malformed antecedents ... ok
from_roman should fail with non-string input ... ok
from_roman should fail with repeated pairs of numerals ... ok
from_roman should fail with too many repeated numerals ... ok
from_roman should give known result with known input ... ok
to_roman should give known result with known input ... ok
from_roman(to_roman(n))==n for all n ... ok
to_roman should fail with negative input ... ok
to_roman should fail with non-integer input ... ok
to_roman should fail with large input ... ok
to_roman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 12 tests in 0.031s

OK
```

1. *Línea 16:* Sé que no lo has preguntado pero, ¡también es rápido! Como diez veces más que antes. Por supuesto, no es una comparación totalmente justa, puesto que esta versión tarda más en importarse (cuando construye las tablas de búsqueda). Pero como el import se realiza una única vez, el coste de inicio se amortiza en las llamadas a las funciones from_roman() y to_roman(). Puesto que las pruebas hacen varios miles de llamadas a las funciones (la prueba de ciclo hace diez mil) se compensa enseguida.

¿La moraleja del cuento?

- La simplicidad es una virtud.
- Especialmente cuando se trata de expresiones regulares.
- Las pruebas unitarias te dan la suficiente confianza como para hacer modificaciones a gran escala en el código.

10.4. Sumario

La prueba unitaria es un concepto muy valioso que, si se implementa de forma adecuada, puede reducir los costes de mantenimiento e incrementar la flexibilidad a largo plazo de cualquier proyecto. También es importante comprender que las pruebas unitarias no son la panacea, ni un solucionador mágico de problemas, ni

un bala de plata. Escribir casos de prueba útiles es una tarea dura y mantenerlos actualizados requiere disciplina (especialmente cuando los clientes están reclamando las correcciones de fallos críticos). Las pruebas unitarias no sustituyen otras formas de prueba, incluídas las pruebas funcionales, de integración y de aceptación por parte del usuario. Pero son prácticas y funcionan y, una vez las hayas utilizado, te preguntarás cómo has podido trabajar alguna vez sin ellas.

Estos pocos capítulos han cubierto muchos aspectos, gran parte de ellos ni siquiera son específicos de Python. Existen marcos de trabajo para pruebas unitarias en muchos lenguajes de programación; todos requieren que comprendas los mismos conceptos básicos:

- Diseño de casos de prueba que sean específicos, automatizados e independientes entre sí.
- Escribir los casos de prueba *antes* del código que se está probando.
- Escribir casos de prueba que tengan entradas válidas y comprobar que se produce el resultado esperado.
- Escribir casos de prueba con entradas erróneas y comprobar que se produce el fallo esperado.
- Escribir y actualizar los casos de prueba para reflejar nuevos requisitos.
- Refactorizar "sin piedad" para mejorar el rendimiento, escalabilidad, legibilidad, mantenibilidad o cualquier otra "bilidad" que eches de menos.

Capítulo 11

Ficheros

Nivel de dificultad:◆ ◆ ◆ ◇ ◇

"Una caminata de nueve millas no es ninguna broma,
especialmente si se hace bajo la lluvia."
—Harry Kemelman, La caminata de nueve millas

11.1. Inmersión

Mi portátil con Windows tenía 38.493 ficheros antes de instalar ninguna aplicación. Al instalar Python se añadieron unos 3.000 más. El fichero es el paradigma principal de almacenamiento de cualquie sistema operativo importante; este concepto está tan arraigado que la mayoría de la gente tendría problemas imaginando una alternativa. Metafóricamente hablando, tu ordenador se está "ahogándose" en ficheros.

11.2. Leer contenido de ficheros de texto

Antes de que puedas leer un fichero, necesitas abrirlo. En Python, esto no podría ser más sencillo:

```
1 a_file = open('examples/chinese.txt', encoding='utf-8')
```

Python dispone de la función interna `open()`, que toma el nombre de un fichero como parámetro. En este ejemplo, el nombre del fichero es "examples/chinese.txt". Hay cinco cosas interesantes que resaltar en este nombre de fichero:

- No es únicamente un nombre de fichero; es una combinación de un camino a través de un directorio y un nombre de fichero. Podríamos pensar en un función hipotética que podría tomar dos parámetros —un camino a un directorio y un nombre de fichero— pero la función open() únicamente toma uno. En Python, siempre que se necesita un "nombre de fichero" puedes utilizar parte o todo el camino al directorio en el que se encuentre.

- El camino al directorio utiliza la barra inclinada hacia adelante, independientemente del sistema operativo que se esté utilizando. Windows utiliza barras inclinadas invertidas para desplazarse entre subdirectorios, mientras que Mac OS X y Linux utilizan barras inclinadas hacia adelante. Pero en Python, las barras inclinadas hacia adelante siempre funcionan, incluso en Windows.

- El camino al directorio no comienza con barra inclinada o letra de unidad, por lo que se trata de un *camino relativo*. ¿Relativo a qué? podrías preguntar... paciencia, mi pequeño saltamontes.

- Es una cadena. Todos los sistemas operativos modernos (¡incluso Windows!) utilizan Unicode para almacenar los nombres de los ficheros y directorios. Python 3 permite trabajar con caminos y ficheros no ASCII.

- No necesita encontrase en un disco local. Podría tratarse de una unidad de red montada en tu ordenador. Este fichero podría ser un elemento de un sistema de ficheros virtual. Si tu ordenador (sistema operativo) lo considera un fichero y puede acceder a él como tal, Python puede abrirlo.

Pero esta llamada a la función open() no se acaba con el nombre del fichero. Existe otro parámetro, denominado encoding. ¡Oh! esto está resultando *terriblemente* familiar.

11.2.1. La codificación de caracteres enseña su fea cara

Los bytes son bytes; los caracteres son una abstracción. Una cadena es una secuencia de caracteres Unicode. Pero un fichero en disco no es una secuencia de caracteres Unicode; un fichero en disco es una secuencia de caracteres. Por eso si lees un fichero de "texto" del disco, ¿Cómo convierte Python la secuencia de bytes en una secuencia de caracteres? Decodifica los bytes de acuerdo a un algoritmo específica de codificación de caracteres y retorna una secuenca de caracteres Unicode (también conocida como una cadena de texto).

```
1  # Este ejemplo se ha creado en Windows. Otras plataformas pueden
2  # comportarse de forma diferente, por las razones descritas más abajo.
3  >>> file = open('examples/chinese.txt')
4  >>> a_string = file.read()
5  Traceback (most recent call last):
6    File "<stdin>", line 1, in <module>
7    File "C:\Python31\lib\encodings\cp1252.py", line 23, in decode
8      return codecs.charmap_decode(input,self.errors,decoding_table)[0]
9  UnicodeDecodeError: 'charmap' codec can't decode byte 0x8f in
10 position 28: character maps to <undefined>
11 >>>
```

¿Qué ha sucedido? No especificamos ninguna codificación de caracteres, por eso Python se ve forzado a utilizar la codificación de caracteres por defecto. ¿Y cuál es esta codificación? Si observas la traza del error observarás que está fallando en cp1252.py, lo que significa que Python está utilizando en este caso CP-1252 como la codificación de caracteres por defecto. El conjunto de caracteres CP-1252 no soporta los caracteres que se encuentran en este fichero, por eso la lectura falla con un feo UnicodeDecodeError.

Pero espera, ¡es peor que eso! La codificación de caracteres por defecto es dependiente de la plataforma, por eso este código podría funcionar en tu ordenador (siempre que tu codificación de caracteres por defecto fuera UTF-8), pero entonces podría fallar cuando lo distribuyeras a alguna otra persona (si su codificación de caracteres por defecto fuera otra, como por ejemplo CP-1252).

> La codificación de caracteres por defecto es dependiente de la plataforma.

> Si necesitas conocer la codificación de caracteres por defecto, importa el módulo locale y llama a la función locale.getpreferredencoding(). En mi portátile Windows, devuelve 'cp1252', pero en mi máquina Linux retorna 'UTF8', lo mismo que en mi MacBookPro. No puedo mantener la consistencia ni ¡en mi propia casa! En cada caso pueden obtenerse diferentes resultados (incluso en Windows) dependiendo de la versión del sistema operativo instalado y la configuración del idioma y de los parámetros regionales. Por eso es muy importante especificar la codificación de caracteres siempre que se abre un fichero "de texto".

11.2.2. Objetos de flujo (streams)

Hasta el momento, todo lo que hemos aprendido es que Python dispone de una función interna denominada open(). Esta función devuelve un objeto de flujo,

que tiene métodos y atributos para recuperar la información y manipular un flujo de caracteres.

```
1 >>> a_file = open('examples/chinese.txt', encoding='utf-8')
2 >>> a_file.name
3 'examples/chinese.txt'
4 >>> a_file.encoding
5 'utf-8'
6 >>> a_file.mode
7 'r'
```

1. *Línea 2:* El atributo name es el nombre que se pasó como parámetro a la función open() cuando se abrió el fichero. No está normalizado a un camino absoluto.

2. *Línea 4:* De igual forma, el atributo encoding refleja la codificación de caracteres que se pasó como parámetro a la función open(). Si no se especifica la codificación de caracteres cuando abriste el fichero (¡mal desarrollador!) entonces el atributo reflejará la función locale.getpreferredencoding().

3. *Línea 6:* El atributo mode refleja el modo en el que se abrió el fichero. Se puede pasar un parámetro opcional mode a la función open(). Si no especificaste el modo al abrir el fichero Python utiliza por defecto 'r', lo que significa que se abra "solamente para lectura, en modo texto". Como se verá más tarde en este capítulo, el modo del fichero sirve para varios propósitos: escribir, añadir o abrir un fichero en el modo binario (en el que se trata el contenido como bytes en lugar de cadenas).

   La documentación de la función open muestra todos los modos válidos.

11.2.3. Leer datos de un fichero de texto

Después de abrir un fichero para lectura problemente querrás leer su contenido.

```
1 >>> a_file = open('examples/chinese.txt', encoding='utf-8')
2 >>> a_file.read()
3 'Dive Into Python 是为有经验的程序员编写的一本 Python 书。\\n'
4 >>> a_file.read()
5 ''
```

1. *Línea 2:* Una vez has abierto el fichero (con la codificación de caracteres correcta) para leer de él hay que llamar al método read() del objeto de flujo. El resultado es una cadena de texto.

2. *Linea 4:* Aunque pueda ser sorprendente, leer de nuevo del fichero no eleva una excepción. Python non considera que leer más allá del final del fichero sea un error; simplemente retorna una cadena vacía.

> Especifica siempre el parámetro encoding cuando abras un fichero.

¿Cómo habría que hacer si quieres releer el fichero?

```
1  # continuación del ejemplo anterior
2  >>> a_file.read()
3  ''
4  >>> a_file.seek(0)
5  0
6  >>> a_file.read(16)
7  'Dive Into Python'
8  >>> a_file.read(1)
9  ' '
10 >>> a_file.read(1)
11 '是'
12 >>> a_file.tell()
13 20
```

1. *Línea 2:* Puesto que estás aún al final del fichero, más llamadas al método read() simplemente retornarán la cadena vacía.
2. *Línea 4:* El método seek() mueve el objeto de flujo a la posición de un byte concreto del fichero.
3. *Línea 6:* El método read() puede tomar un parámetro opcional, el número de caracteres a leer.
4. *Línea 8:* Si quieres, puedes leer un carácter cada vez.
5. *Línea 12:* ¿16 + 1 + 1 = ... 20?

Vamos a intentarlo de nuevo.

```
1  # sigue del ejemplo anterior
2  >>> a_file.seek(17)
3  17
4  >>> a_file.read(1)
5  '是'
6  >>> a_file.tell()
7  20
```

1. *Línea 2:* Se mueve al byte 17.

2. *Línea 4:* Lee un carácter.

3. *Línea 6:* Ahora estás en el byte 20.

¿Lo ves ahora? Los métodos seek() y tell() siempre cuentan en *bytes*, pero como el fichero está abierto como texto, el método read() cuenta *caracteres*. Los caracteres chinos necesitan varios bytes para codificarse en UTF8. Los caracteres ingleses únicamente requieren un byte, por eso puedes llegar a pensar que los métodos seek() y read() cuentan la misma cosa. Pero esto únicamente es cierto para algunos caracteres.

Pero espera, ¡que es peor!

```
1  >>> a_file.seek(18)
2  18
3  >>> a_file.read(1)
4  Traceback (most recent call last):
5    File "<pyshell#12>", line 1, in <module>
6      a_file.read(1)
7    File "C:\Python31\lib\codecs.py", line 300, in decode
8      (result, consumed) = self._buffer_decode(data, self.errors, final)
9  UnicodeDecodeError: 'utf8' codec can't decode byte 0x98 in
10 position 0: unexpected code byte
```

1. *Línea 1:* Muévete al byte 18 e intenta leer un carácter.

2. *Línea 3:* ¿Porqué falla? Porque no existe un carácter en el byte 18. El carácter más cercano comienza en el byte 17 (y ocupa tres bytes). Intentar leer un carácter en la mitad dará un error UnicodeDecodeError.

11.2.4. Cerrar ficheros

Los ficheros abiertos consumen recursos del sistema, y dependiendo del modo de apertura, puede que otros programas no puedan acceder a ellos. Es importante que se cierren los ficheros tan pronto como se haya terminado de trabajar con ellos.

```
1  # sigue del ejemplo anterior
2  >>> a_file.close()
```

Bueno *eso* ha sido algo anticlimático.

El objeto a_file[1] aún existe; el llamar a su método close() no destruye el objeto, pero éste deja de ser útil.

[1] que es un objeto de flujo o *stream*

```
1  # sigue del ejemplo anterior
2  >>> a_file.read()
3  Traceback (most recent call last):
4    File "<pyshell#24>", line 1, in <module>
5      a_file.read()
6  ValueError: I/O operation on closed file.
7  >>> a_file.seek(0)
8  Traceback (most recent call last):
9    File "<pyshell#25>", line 1, in <module>
10     a_file.seek(0)
11 ValueError: I/O operation on closed file.
12 >>> a_file.tell()
13 Traceback (most recent call last):
14   File "<pyshell#26>", line 1, in <module>
15     a_file.tell()
16 ValueError: I/O operation on closed file.
17 >>> a_file.close()
18 >>> a_file.closed
19 True
```

1. *Línea 2:* No puedes leer de un fichero cerrado; se eleva una excepción IOError.
2. *Línea 7:* Tampoco puedes moverte en un fichero cerrado.
3. *Línea 12:* No existe una posición activa si el fichero está cerrado, por eso también falla el método tell().
4. *Línea 17:* Sorprendentemente, tal vez, llamar de nuevo al método close() sobre un objeto ya cerrado *no* eleva una excepción. Simplemente no hace nada.
5. *Línea 18:* Los objetos de flujos cerrados tienen un atributo que sí es útil: el atributo closed, que sirve para confirmar que el fichero está cerrado.

11.2.5. Cerrar ficheros de forma automática

Los objetos de flujo (streams) tiene un método close() para cerrar explícitamente el flujo. Pero, ¿qué sucede si tu código tiene un error y falla antes de que llames al método close()? En teoría, este fichero quedaría abierto permanentemente. Mientras estás depurando un nuevo código en tu ordenador personal, no sería un gran problema. En un servidor de producción sí que puede serlo.

Python 2 tenía una solución para ello: el bloque try...finally. Esta solución aún funciona en Python 3, y puede encontrarse en el código preexistente o en el otras personas. Pero Python 2.5 introdujo una solución más limpia, que es la preferida en Python 3: la sentencia with.

```
1 with open('examples/chinese.txt', encoding='utf-8') as a_file:
2     a_file.seek(17)
3     a_character = a_file.read(1)
4     print(a_character)
```

Este código llama a open() pero no a a_file.close(). La sentencia with inicia un bloque de código, como el de una sentencia if o un bucle for. Dentro del bloque puedes utilizar la variable a_file que contiene el objeto de flujo devuelto por la función open(). Lógicamente, están disponibles todos los métodos del objeto de flujo —seek(), read() o lo que necesites. Cuando el bloque with finaliza, Python *llama automáticamente* a a_file.close().

Lo importante es: no importa cómo o cuándo finalices el bloque with, Python cerrará el fichero... incluso si la salida se produce a causa de una excepción sin manejar. Sí, lo oyes bien, incluso si el código eleva una excepción y el programa finaliza, el fichero se cerrará. Garantizado.

> En términos técnicos, la sentencia with crea un contexto de ejecución. En estos ejemplo, el objeto de flujo actúa como gestor del contexto. Python crea el objeto de flujo a_file y le dice que está entrando en un contexto de ejecución. Cuando el bloque with finaliza, Python le dice al objeto de flujo que está saliendo del contexto de ejecución y el objeto de flujo llama por sí mismo al método close(). Para conocer más detalles puedes ver el Apéndice B, "Clases que se pueden utilizar en un bloque with".

No hay nada específico relacionado con los ficheros en una sentencia with, es un marco general para crear contextos de ejecución y decirle a los objetos que están entrando y saliendo de él. Si el objeto en cuestión es un objeto de flujo (streams), entonces hace cosas útiles relacionadas con los ficheros (como cerrar el fichero automáticamente). Pero el comportamiento lo define el propio objeto de flujo, no la sentencia with. Existen otras muchas formas de utilizar gestores de contexto que no tienen nada que ver con los ficheros. Puedes crear los tuyos propios, como verás más tarde neeste mismo capítulo.

11.2.6. Leer los datos línea a línea

Una "línea" en un ficheo de texto es lo que te puedes imaginar — tecleas unas cuantas palabras y pulsas INTRO y ya estás en una nueva línea. Una línea de texto es una secuencia de caracteres que está delimitada por... ¿qué cosa exactamente? Bueno, es complicado de decir, porque los ficheros de texto pueden utilizar diferentes caracteres para marcar el final de una línea. Cada sistema operativo tiene su propia

convención. En algunos se utiliza el carácter de "retorno de carro", otros utilizan el caracter de "salto de línea" y algunos utilizan ambos al final de cada línea.

Ahora respira con expresión de relajación, porque Python *controla estos tipos diferentes de fin de línea de forma automática* por defecto. Si le dices que "quieres leer un fichero de texto de línea en línea", Python descubrirá por su cuenta la clase de fin de línea que el fichero de texto utiliza y simplemente funcionará como esperas.

> Si necesitas un control más fino sobre lo que debe considerar Python como fin de línea deberás pasar el parámetro opcional newline a la función open(). Mira la documentación de la función open() para ver los detalles necesarios.

Bueno, ¿cómo hay que hacer para leer una línea cada vez? Es simple, es bello.

```
1 line_number = 0
2 with open('examples/favorite-people.txt', encoding='utf-8') as a_file:
3     for a_line in a_file:
4         line_number += 1
5         print('{:>4} {}'.format(line_number, a_line.rstrip()))
```

1. *Línea 2:* Utilizando el patrón with abres el fichero de forma segura y dejas a Python que lo cierre por ti.

2. *Línea 3:* Para leer un fichero línea a línea lo mejor es utilizar el bucle for. Además de tener métodos explícitos como el read(), un objeto de flujoo también es un iterador que retorna una línea del fichero cada vez que le pides un valor.

3. *Línea 5:* Si utilizas el método format puedes ir imprimendo el número de línea y la propia línea. El especificador de formaot {:¿4} significa que imprima el parámetro justificado a la derecha dentro de cuatro espacios. La variable a_line contiene una línea completa, incluyendo el retorno de carro. El método rstrip() de las cadenas de texto elimina los espacios en blanco del final de una cadena, incluyendo los caracteres de retorno de carro y salto de línea.

```
you@localhost:~/diveintopython3$ python3 examples/oneline.py
   1 Dora
   2 Ethan
   3 Wesley
   4 John
   5 Anne
   6 Mike
   7 Chris
   8 Sarah
   9 Alex
  10 Lizzie
```

¿Falló con este error?

```
you@localhost:~/diveintopython3$ python3 examples/oneline.py
Traceback (most recent call last):
  File "examples/oneline.py", line 4, in <module>
    print('{:>4} {}'.format(line_number, a_line.rstrip()))
ValueError: zero length field name in format
```

Si fue así es que probablemente estés utilizando Python 3.0. Deberías actualizarte a Python 3.1.

Python 3.0 soportaba el formateo de cadenas, pero únicamente con especificadores de formato numerados explícitamente. Python 3.1 te permite omitir el índice del parámetro en los especificadores de formato. Esta sería la versión compatible en Python 3.0, para que puedas compararla con la anterior.

```
print('{0:>4} {1}'.format(line_number, a_line.rstrip()))
```

11.3. Escribir en ficheros de texto

Se puede escribir en ficheros de forma parecida a como se lee de ellos. Primero se abre el fichero y se obtiene el objeto de flujo, luego se utilizan los métodos del objeto de flujo que sirven para escribir datos en el fichero, para terminar se debe cerrar el fichero.

Para abrir un fichero para escribir se debe utilizar la función open() y especificar el modo de escritura. Existen dos modos de escritura:

Simplemente abre el fichero y comienza a escribir.

- Modo de "escritura" que sobreescribe el fichero. Se debe pasar el parámetro mode='w' a la función open().

- Modo de "anexación" que añade datos al final del fichero, conservando los datos que existieran anteriormente. Se debe pasar el parámetro mode='a' a la función open().

Cualquiera de los dos modos creará el fichero automáticamente si no existiera ya, por lo que no hay necesidad de ningún tipo de código que compruebe si "el fichero aún no existe, crea un nuevo fichero vacío para que lo pueda abrir después por primera vez". Simplemente abre el fichero y comienza a escribir.

Siempre deberías cerrar el fichero cuando hayas terminado de escribir con el fin de liberar al manejador del fichero y asegurar que los datos realmente se han escrito en el disco. Como cuando se leen datos de un fichero, se puede utilizar el método close() o puedes utilizar la sentencia with y dejar a Python que cierre el fichero por ti. Apuesto a que puedes adivinar la técnica que te recomiendo.

```
1 >>> with open('test.log', mode='w', encoding='utf-8') as a_file:
2 ...     a_file.write('test succeeded')
3 >>> with open('test.log', encoding='utf-8') as a_file:
4 ...     print(a_file.read())
5 test succeeded
6 >>> with open('test.log', mode='a', encoding='utf-8') as a_file:
7 ...     a_file.write('and again')
8 >>> with open('test.log', encoding='utf-8') as a_file:
9 ...     print(a_file.read())
10 test succeededand again
```

1. *Línea 1:* Comienzas creando un nuevo fichero test.log (o sobreescribiendo el fichero existente) y abriendo el fichero en modo escritura. El parámetro mode='w' significa que abres el fichero para escribir en él. Si, tan peligroso como suena. Espero que no te importase el contenido previo que el fichero pudiera tener (si tenía alguno), porque todos esos datos ya han desaparecido.

2. *Línea 2:* Puedes añadir datos al fichero recién abierto utilizando el método write() del objeto de flujo devuelto por la función open(). Después el bloque with termina y Python cierra automáticamente el fichero.

3. *Línea 6:* Estuvo bien, vamos a hacerlo de nuevo. Pero esta vez con mode='a' para añadir al final del fichero en lugar de sobreescribirlo. Añadir *nunca* dañará el contenido preexistente del fichero.

4. *Línea 10:* Ahora el fichero contiene tanto la línea original como la segunda línea que añadiste a test.log. Ten en cuenta que no se incluyen retornos de caro. Puesto que no los incluiste explícitamente al fichero ninguna de las veces,

el fichero no los contiene. Puedes escribir un retorno de carro utilizando el carácter '
n'. Al no haberlo incluido, todo lo que escribiste al fichero acaba en una única línea.

11.3.1. Codificación de caracteres, otra vez

¿Observaste el parámetro encoding que se pasaba a la función open() mientras abrías el fichero para escritura? Es importante, ¡nunca lo omitas! Como has visto al comienzo de este capítulo, los ficheros no contienen *cadenas de texto*, contienen *bytes*. La lectura de una "cadena de texto" de un fichero de texto funciona porque le dices a Python la codificación de caracteres que tiene que utilizar para leer el flujo de bytes y convertirla a una cadena de texto. Al escribir texto a un fichero ocurre el mismo problema pero en sentido inverso. No puedes escribir caracteres a un fichero; los caracteres son una abstracción. Para escribir a un fichero, Python necesita saber cómo convertir la cadena de texto en una secuencia de bytes. La única forma de estar seguro de ejecutar la conversión correcta es especificar el parámetro encoding cuando abres el fichero en modo escritura.

11.4. Ficheros binarios

No todos los ficheros contienen texto. Algunos contienen fotos de mi perro.

```
1 >>> an_image = open('examples/beauregard.jpg', mode='rb')
2 >>> an_image.mode
3 'rb'
4 >>> an_image.name
5 'examples/beauregard.jpg'
6 >>> an_image.encoding
7 Traceback (most recent call last):
8   File "<stdin>", line 1, in <module>
9 AttributeError: '_io.BufferedReader' object has no attribute 'encoding'
```

1. *Línea 1:* Abrir un fichero en modo binario es simple pero sutil. La única diferencia respecto de abrirlo en modo de texto es que el parámetro mode contiene un carácter 'b' adicional.

2. *Línea 2:* El objeto de flujo que obtienes cuando se abre un fichero en modo binario tiene muchos de los atributos que ya se han visto, incluido mode, que refleja el parámetro mode que se pasó a la función open().

3. *Línea 4:* Los objetos de flujo binarios también tienen el atributo name, como pasa con los objetos de flujo de texto.

4. *Línea 6:* Aquí hay una diferencia: un objeto de flujo binario no tiene el atributo encoding. Tiene sentido ¿verdad? Estás leyendo (o escribiendo) bytes, no cadenas de texto, por lo que no hay ninguna conversión que hacer. Lo que obtienes de un fichero binario es exactamente lo que pones en él, no hay necesidad de ninguna conversión.

   ¿He mencionado que estabas leyendo bytes? ¡Oh! pues estás leyendo bytes.

```
1  # sigue del ejemplo anterior
2  >>> an_image.tell()
3  0
4  >>> data = an_image.read(3)
5  >>> data
6  b'\xff\xd8\xff'
7  >>> type(data)
8  <class 'bytes'>
9  >>> an_image.tell()
10 3
11 >>> an_image.seek(0)
12 0
13 >>> data = an_image.read()
14 >>> len(data)
15 3150
```

1. *Línea 4:* Como con los ficheros de texto, los ficheros binarios se pueden leer poco a poco. Pero hay una diferencia crucial...

2. *Línea 7:* ...estás leyendo bytes, no cadenas de texto. Puesto que el fichero se ha abierto en modo binario, el método read() toma como parámetro el *número de bytes que se desea leer*, no el número de caracteres.

3. *Línea 9:* Lo que signific que nunca hay diferencia entre el número que le pasas como parámetro al método read(9 y el índice que devuelve el método tell(). El método read() lee bytes, y los métodos seek() y tell() cuentan el número de bytes leídos. Siempre coinciden en el caso de los ficheros binarios.

11.5. Objetos de flujo obtenidos de fuentes que no son ficheros

Imagina que estás escribiendo una librería, y una de las funciones de ésta lee algunos datos de un fichero. La función podría tomar como parámetro el nombre del fichero en formato cadena de texto, abriría el fichero para lectura, leería de él y lo cerraría antes de terminar. Por no deberías hacer esto. En vez de esto, tu API debería tomar como parámetro un objeto de flujo cualquiera.

En el caso más simple, un objeto de flujo es cualquier objeto que tenga un método read() con un parámetro opcional size para pasarle el tamaño a leer y que devuelve una cadena de texto. Cuando se le llama sin el parámetro size, el métod read() debería leer todo lo que falta por leer para devolver todos los datos como una única cadena de texto. Cuando se llama con el parámetro size, lee esa cantidad desde la entrada devolviendo una cadena de texto con estos datos. Cuando se le llama de nuevo, continúa por donde quedó y devuelve el siguiente trozo de los datos de entrada.

> Para leer de un fichero ficticio, simplemente utiliza read().

Este comportamiento es idéntico a los objetos de flujo que obtienes cuando abres un fichero real. La diferencia es que *no te estás limitando a ficheros reales*. La fuente de etrada que se está leyendo puede ser cualquier cosa: una página web, una cadena en memoria o, incluso, la salida de otro programa. Siempre que tus funciones tomen como parámetro un objeto de flujo y llamen al método read() podrás manejar cualquier fuente de entrada que se comporte como un fichero, sin que tengas que escribir código que maneje cada tipo específico de entrada.

```
1 >>> a_string = 'PapayaWhip is the new black.'
2 >>> import io
3 >>> a_file = io.StringIO(a_string)
4 >>> a_file.read()
5 'PapayaWhip is the new black.'
6 >>> a_file.read()
7 ''
8 >>> a_file.seek(0)
9 0
10 >>> a_file.read(10)
11 'PapayaWhip'
12 >>> a_file.tell()
13 10
14 >>> a_file.seek(18)
15 18
16 >>> a_file.read()
17 'new black.'
```

1. *Línea 2:* El módulo io define la clase StringIO que puedes utilizar para tratar a las cadenas de texto en memoria como si fuesen un fichero.

2. *Línea 3:* Para crear un objeto de flujo a partir de una cadena de texto debes crear una instancia de la clase io.StringIO() y pasarle la cadena de texto que quieres recorrer como si fuesen datos de un fichero. Ahora tienes un objeto de flujo y puedes hacer todo lo que puedes hacer con los objetos de flujo.

3. *Línea 4:* Al llamar al método read() se lee el "fichero" completo, lo que en el caso de un objeto StringIO es tan simple como obtener la cadena de texto original.

4. *Línea 6:* Como pasa con un fichero real, el llamar de nuevo a read() devuelve una cadena vacía.

5. *Línea 8:* Puedes buscar explícitamente el comienzo de la cadena, como en un fichero real, mediante el uso del método seek() del objeto StringIO.

6. *Línea 10:* También puedes leer la cadena a trozos pasándole el parámetro size al método read().

   io.StringIO te permite manipular una cadena de texto como si fuese un fichero de texto. También existe una clase io.BytesIO que te permite tratar un array de bytes como un fichero binario.

11.5.1. Manipular ficheros comprimidos

La librería estándar de Python contiene módulos que permiten leer y escribir ficheros comprimidos. Existen diversos sistemas de compresión; los dos más populares en sistemas no windows son gzip y bzip2 (También te puedes encontrar archivos PKZIP y GNU Tar. Python también dispone de módulos para ellos).

El módulo gzip te permite crear un objeto de flujo para leer y escribir ficheros comprimidos con el formato gzip. Este objeto dispone del método read() (si lo abriste para lectura) y del método write() (si lo abriste de escritura). Esto significa que puedes utilizar los métodos que ya has aprendido para leer o escribir directamente ficheros comprimidos en formato gzip sin tener que crear un fichero temporal con los datos sin comprimir.

Como un bonus añadido, también permite el uso de la sentencia with por lo que puedes dejar a Python que cierre el fichero comprimido de forma automática cuando hayas terminado de trabajar con él.

```
1  you@localhost:~$ python3
2  >>> import gzip
3  >>> with gzip.open('out.log.gz', mode='wb') as z_file:
4  ...     z_file.write(
5  ...     'A nine mile walk is no joke, especially in the rain.'.encode(
6  ...     'utf-8'))
7  ...
8  >>> exit()
9
10 you@localhost:~$ ls -l out.log.gz
11 -rw-r--r--  1 mark mark    79 2009-07-19 14:29 out.log.gz
12 you@localhost:~$ gunzip out.log.gz
13 you@localhost:~$ cat out.log
14 A nine mile walk is no joke, especially in the rain.
```

1. *Línea 3:* Los ficheros comprimidos se deben abrir siempre en modo binario (Observa el carácter 'b' en el parámetro mode).

2. *Línea 8:* Este ejemplo lo construí en Linux. Si no estás familiarizado con la línea de comando, este comando te muestra un "listado largo" del fichero comprimido que acabas de crear con la consola de Python.

3. *Línea 10:* El comando gunzip descomprime el fichero y almacena el contenido en un nuevo fichero con el mismo nombre que el original pero sin la extensión .gz.

4. *Línea 11:* El comando cat muestra el contenido de un fichero. Este fichero con-

tiene la cadena de texto que escribiste directamente en el fichero comprimido out.log.gz desde la consola de Python.

¿Te pasó este error?

```
>>> with gzip.open('out.log.gz', mode='wb') as z_file:
...         z_file.write(
...         'A nine mile walk is no joke, especially in the rain.'.encode(
...         'utf-8'))
...
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
AttributeError: 'GzipFile' object has no attribute '__exit__'
```

Si fue así, posiblemente estés utilizando Python 3.0. Deberías actualizarte a Python 3.1. Python 3.0 tenía un módulo gzip pero no soportaba el uso de los objetos de flujo de ficheros comprimidos como parte de un gestor de contexto. Python 3.1 añade esta funcionalidad que permite utilizar estos objetos como parte de la sentencia with.

11.6. Flujos de entrada, salida y error estándares

Los gurús de la línea de comando están familiarizados con el concepto de entrada estándar, salida estándar y error estándar. Esta sección es para el resto de vosotros.

sys.stdin, sys.stdout, sys.stderr.

La salida estándar y la salida de error estándar (comunmente abreviadas como stdout y stderr) son "tuberías" (pipes) que vienen en cualquier sistema de tipo UNIX, incluyendo Mac OS X y Linux. Cuando llamas a la función print(), lo que se imprime se envía a la tubería de salida stdout. Cuando tu programa falla e imprime la traza de error, la salida va a la tubería stderr. Por defecto, ambas tuberías están conectadas directamenten a la ventana del terminal en el que estás trabajando; cuando tu programa imprime algo, ves la salida en la ventana del terminal; y cuando el programa falla, también ves la traza de error en la misma ventana del terminal. En la consola gráfica de Python, las tuberías stdout y stderr están conectadas por defecto a la "ventana interactiva" en la que te encuentras.

```
1  >>> for i in range(3):
2  ...     print('PapayaWhip')
3  PapayaWhip
4  PapayaWhip
5  PapayaWhip
6  >>> import sys
7  >>> for i in range(3):
8  ...     sys.stdout.write('is the')
9  is theis theis the
10 >>> for i in range(3):
11 ...     sys.stderr.write('new black')
12 new blacknew blacknew black
```

1. *Línea 2:* La función print() en un bucle. Nada sorprendente en este trozo de código.

2. *Línea 8:* stdout está definida en el módulo sys y es un objeto de flujo. Si se llama a su función write() se imprimirá la cadena de texto que le pases como parámetro. De hecho, es lo que hace la función print: añade un retorno de carro a la cadena que quieres imprimir y llama a sys.stdout.write.

3. *Línea 11:* En el caso más simple, sys.stdout y sys.stderr envían su salida al mismo sitio: en entorno integrado de desarrollo de Python (si te encuentras en uno) o el terminal (si estás ejecutando Python desde la línea de comando). Como en el caso de la salida estándar, la salida de error tampoco añade retornos de carro por ti. Si quieres retornos de carro, tienes que añadirlos.

sys.stdout y sys.stderr son objetos de flujo, abiertos como de escritura únicamente. Si se intenta llamar a sus métodos read() se elevará el error IOError.

```
1 >>> import sys
2 >>> sys.stdout.read()
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in <module>
5 IOError: not readable
```

11.6.1. Redirección de la salida estándar

sys.stdout y sys.stderr son objetos de flujo, aunque únicamente soportan la escritura. Pero no son constantes, son variables. Esto significa que puedes asignarles un nuevo valor —cualquier otro objeto de flujo— para redirigir su salida.

```
import sys

class RedirectStdoutTo:
    def __init__(self, out_new):
        self.out_new = out_new

    def __enter__(self):
        self.out_old = sys.stdout
        sys.stdout = self.out_new

    def __exit__(self, *args):
        sys.stdout = self.out_old

print('A')
with open('out.log', mode='w', encoding='utf-8') \
    as a_file, RedirectStdoutTo(a_file):
    print('B')
print('C')
```

Prueba esto:

```
you@localhost:~/diveintopython3/examples$ python3 stdout.py
A
C
you@localhost:~/diveintopython3/examples$ cat out.log
B
```

¿Te pasó este error?

```
you@localhost:~/diveintopython3/examples$ python3 stdout.py
  File "stdout.py", line 15
    with open('out.log', mode='w', encoding='utf-8') \
         as a_file, RedirectStdoutTo(a_file):
                  ^
SyntaxError: invalid syntax
```

Si es así, probablemente estés utilizando Python 3.0. Posiblemente deberías actualizarte a Python 3.1.

Python 3.0 soporta la sentencia with, pero cada sentencia puede utilizar únicamente un gestor de contexto. Python 3.1 te permite encadenar varios gestores de contexto en una única sentencia with.

Vamos a ver la última parte primero.

```
1 print('A')
2 with open('out.log', mode='w', encoding='utf-8') \
3     as a_file, RedirectStdoutTo(a_file):
4     print('B')
5 print('C')
```

Se trata de una sentencia with compleja. Déjame que la reescriba como algo más reconocible.

```
1 with open('out.log', mode='w', encoding='utf-8') as a_file:
2     with RedirectStdoutTo(a_file):
3         print('B')
```

Como muestra esta reescritura, en realidad se trata de dos sentencias with, una de ellas anidada en el ámbito de la otra. La sentencia with "exterior" debería ya serte familiar: abre un fichero de texto codificado en UTF8 denominado out.log, para escritura; y lo asigna a la variable denominada a_file. Pero no es lo único extraño aquí.

```
1 with RedirectStdoutTo(a_file)
```

¿Dónde está la claúsula as aquí? La sentencia with no la requiere. Al igual que puedes llamar a una función e ignorar su valor de retorno, puedes crear una sentencia with sin asignar el contexto resultante a una variable. En este caso solamente estás interesado en el efecto "lateral" del contexto RedirectStdoutTo.

¿Cuál es el efecto lateral de este contexto? Echa un vistazo a la clase RedirectStdoutTo. Esta clase es un gestor de contexto a medida. Cualquier clase puede serlo si define dos métodos especiales: __enter__() y __exit__().

```
1  class RedirectStdoutTo:
2      def __init__(self, out_new):
3          self.out_new = out_new
4
5      def __enter__(self):
6          self.out_old = sys.stdout
7          sys.stdout = self.out_new
8
9      def __exit__(self, *args):
10         sys.stdout = self.out_old
```

1. *Línea 2:* El método __init__() se llama inmediatamente después de que la instancia se crea. Toma un parámetro, el objeto de flujo que quieres utilizar como salida estándar durante la vida del contexto. Este método simplemente almacena el objeto de flujo en una variable para que los otros métodos lo puedan usar más tarde.

2. *Línea 5:* El método __enter__() es un método especial de la clase; Python lo llama cuando se entra en un contexto (por ejemplo: al comienzo de una bloque with). Este método almacena el valor actual de sys.stdout en self.out_old, luego redirige la salida estándar asignando self.out_new a sys.stdout.

3. *Línea 9:* El método __exit__() es otro método especial de la clase; Python lo llama cuando sale del contexto (por ejemplo: al final del bloque with). Este método restablee la salida estandar a su valor original asignando el valor almacenado self.old_value a sys.stdout.

Al juntarlo todo:

```
1 print('A')
2 with open('out.log', mode='w', encoding='utf-8') as a_file, \
3      RedirectStdoutTo(a_file):
4     print('B')
5 print('C')
```

1. *Línea 1:* Imprimirá en la ventana interativa (del entorno integrado de desarrollo o el terminal, dependiendo de cómo estés ejecutando este programa).

2. *Línea 2:* La sentencia with recibe una lista de contextos separada por comas. Esta lista actúa como una serie de bloques with anidados. El primer contexto es el bloque externo, el último es el interno. El primer contexto abre un fichero, el segundo redirige sys.stdout al objeto de flujo que se creó en el primer contexto.

3. *Línea 3:* Debido a que la función print() se ejecutó en el contexto creado por la sentencia with, no se imprimirá a la pantalla, se imprimirá en el fichero out.log.

4. *Línea 4:* El bloque with se ha acabado. Python le ha dicho a cada gestor de contexto que hagan lo que tengan que hacer para salir del contexto. Los gestores de contexto forman una pila LIFO (Last-in-first-out: Primero en entrar, último en salir). En la salida, el segundo contexto vuelve a poner el valor original de sys.stdout, luego el primer contexto cierra el fichero out.log. Puesto que la salida estándar ha vuelto a su valor inicial, la llamada a la función print() de nuevo imprimirá en la pantalla.

La redirección de la salida de error estándar funciona de igual manera, simplemente cambiando sys.stdout por sys.stderr.

11.7. Lecturas recomendadas

- Lectura y escritura de ficheros en el tutorial de Python:
  http://docs.python.org/tutorial/inputoutput.html#reading-and-writing-files
- El módulo io:
  http://docs.python.org/3.1/library/io.html
- Objetos de flujo (streams):
  http://docs.python.org/3.1/library/stdtypes.html#file-objects
- Tipos de gestores de contexto:
  http://docs.python.org/3.1/library/stdtypes.html#context-manager-types
- sys.stdout y sys.stderr:
  http://docs.python.org/3.1/library/sys.html#sys.stdout
- FUSE en la Wikipedia:
  http://en.wikipedia.org/wiki/Filesystem_in_Userspace

Capítulo 12

XML

Nivel de dificultad:◆ ◆ ◆ ◆ ◇

"En el gobierno de Aristemo,
Draco aplicó sus ordenanzas."
—Aristóteles

12.1. Inmersión

La mayoría de los capítulos de este libro se han desarrollado alrededor de un código de ejemplo. Pero XML no trata sobre código, trata sobre datos. Un uso muy común para XML es la "provisión de contenidos sindicados" que lista los últimos artículos de un blog, foro u otro sitio web con actualizaciones frecuentes. El software para blogs más popular puede generar fuentes de información y actualizarlas cada vez que hay nuevos artículos, hilos de discusión o nuevas entradas en un blog. Puedes seguir un blog "sscribiéndote" a su canal (feed), y puedes seguir diversos blogs mediante un agregador de canales como el lector de Google.

Aquí están los datos de XML que utilizaremos en este capítulo. Es un canal —específicamente, una fuente de información sindicada **Atom**.

```
<?xml version='1.0' encoding='utf-8'?>
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'>
  <title>dive into mark</title>
  <subtitle>currently between addictions</subtitle>
  <id>tag:diveintomark.org,2001-07-29:/</id>
  <updated>2009-03-27T21:56:07Z</updated>
  <link rel='alternate' type='text/html'
        href='http://diveintomark.org/'/>
  <link rel='self' type='application/atom+xml'
        href='http://diveintomark.org/feed/'/>
  <entry>
    <author>
      <name>Mark</name>
      <uri>http://diveintomark.org/</uri>
    </author>
    <title>Dive into history, 2009 edition</title>
    <link rel='alternate' type='text/html'
      href='http://diveintomark.org/archives/2009/03/27/ (sigue abajo)
      dive-into-history-2009-edition'/>
    <id>tag:diveintomark.org,2009-03-27:/archives/20090327172042</id>
    <updated>2009-03-27T21:56:07Z</updated>
    <published>2009-03-27T17:20:42Z</published>
    <category scheme='http://diveintomark.org' term='diveintopython'/>
    <category scheme='http://diveintomark.org' term='docbook'/>
    <category scheme='http://diveintomark.org' term='html'/>
  <summary type='html'>Putting an entire chapter on one page sounds
    bloated, but consider this &amp;mdash; my longest chapter so far
    would be 75 printed pages, and it loads in under 5
    seconds&amp;hellip; On dialup.</summary>
  </entry>
```

```
  <entry>
    <author>
      <name>Mark</name>
      <uri>http://diveintomark.org/</uri>
    </author>
    <title>Accessibility is a harsh mistress</title>
    <link rel='alternate' type='text/html'
      href='http://diveintomark.org/archives/2009/03/21/ (sigue)
      accessibility-is-a-harsh-mistress'/>
    <id>tag:diveintomark.org,2009-03-21:/archives/20090321200928</id>
    <updated>2009-03-22T01:05:37Z</updated>
    <published>2009-03-21T20:09:28Z</published>
    <category scheme='http://diveintomark.org' term='accessibility'/>
    <summary type='html'>The accessibility orthodoxy does not permit
      people to question the value of features that are rarely
      useful and rarely used.</summary>
  </entry>
  <entry>
    <author>
      <name>Mark</name>
    </author>
    <title>A gentle introduction to video encoding, part 1:
          container formats</title>
    <link rel='alternate' type='text/html'
      href='http://diveintomark.org/archives/2008/12/18/ (sigue)
           give-part-1-container-formats'/>
    <id>tag:diveintomark.org,2008-12-18:/archives/20081218155422</id>
    <updated>2009-01-11T19:39:22Z</updated>
    <published>2008-12-18T15:54:22Z</published>
    <category scheme='http://diveintomark.org' term='asf'/>
    <category scheme='http://diveintomark.org' term='avi'/>
    <category scheme='http://diveintomark.org' term='encoding'/>
    <category scheme='http://diveintomark.org' term='flv'/>
    <category scheme='http://diveintomark.org' term='GIVE'/>
    <category scheme='http://diveintomark.org' term='mp4'/>
    <category scheme='http://diveintomark.org' term='ogg'/>
    <category scheme='http://diveintomark.org' term='video'/>
    <summary type='html'>These notes will eventually become part of a
      tech talk on video encoding.</summary>
  </entry>
</feed>
```

12.2. Curso rápido de 5 minutos sobre XML

Si conoces ya XML puedes saltarte esta sección.

XML es una forma generalizada de describir una estructura de datos jerárquica.

Un *documento* XML contiene uno o más *elementos*, que están delimitados por *etiquetas* de inicio y fin. Lo siguiente es un documento XML completo (aunque bastante aburrido).

```
1 <foo>
2 </foo>
```

1. *Línea 1:* Esta es la etiqueta de inicio del elemento foo.
2. *Línea 2:* Esta es la etiqueta de fin del elemento foo, que es pareja de la anterior. Como los paréntesis en la escritura, matemáticas o código, toda etiqueta de inicio debe *cerrase* con una etiqueta de fin.

Los elementos se pueden *anidar* a cualquier profundidad. Si un elemento bar se encuentra dentro de un elemento foo, se dice que bar es un *subelemento* o *hijo* de foo.

```
1 <foo>
2   <bar></bar>
3 </foo>
```

Al primer elemento de un documento XML se le llama el *elemento raíz*. Un documento XML únicamente puede tener un elemento raíz. Lo siguiente **no es un documento XML** porque tiene dos elementos raíz:

```
1 <foo></foo>
2 <bar></bar>
```

Los elementos pueden tener *atributos*, que son parejas de nombres con valores. Los atributos se deben incluir dentro de la etiqueta de inicio del elemento y deben estar separados por un espacio en blanco. Los *nombres de atributo* no se pueden repetir dentro de un elemento. Los valores de los atributos deben ir entre comillas. Es posible utilizar tanto comillas simples como dobles.

```
1 <foo lang='en'>
2   <bar id='papayawhip' lang="fr"></bar>
3 </foo>
```

1. *Línea 1:* El elemento foo tiene un atributo denominado lang. El valor del atributo lang es en.
2. *Línea 2:* El elemento bar tiene dos atributos. El valor del atributo lang es fr. Esto no entra en conflicto con el elemento foo, cada elemento tiene su propio conjunto de atributos.

Si un elemento tiene más de un atributo, el orden de los mismos no es significativo. Los atributos de un elemento forman un conjunto desordenado de claves y valores, como en un diccionario de Python. No exite límite en el número de atributos que puedes definir para cada elemento.

Los elementos pueden contener texto.

```
1 <foo lang='en'>
2   <bar lang='fr'>PapayaWhip</bar>
3 </foo>
```

Existe una forma de escribir elementos vacíos de forma compacta. Colocando un carácter / al final de la etiqueta de inicio se puede evitar tener que escribir la etiqueta de fin. El documento XML del ejemplo anterior se puede escribir de esta otra forma:

```
1 <foo/>
```

Como pasa con las funciones de Python que se pueden declarar en diferentes *módulos*, los elementos XML se pueden declarar en diferentes espacios de nombre. Los espacios de nombre se suelen representar como URLs. Se puede utilizar una declaración xmlns para definir un espacio de nombres por defecto. Una declaración de un espacio de nombres es parecida a un atributo, pero tiene un significado y propósito diferente.

```
1 <feed xmlns='http://www.w3.org/2005/Atom'>
2   <title>dive into mark</title>
3 </feed>
```

1. *Línea 1:* El elemento feed se encuentra en el espacio de nombres http://www.w3.org/2005/Ato
2. *Línea 2:* El elemento title se encuentra también en el espacio de nombres http://www.w3.org/2005/Atom. La declaración del espacio de nombres afecta al elemento en el que está declarado y a todos los elementos hijo.

```
1 <atom:feed xmlns:atom='http://www.w3.org/2005/Atom'>
2   <atom:title>dive into mark</atom:title>
3 </atom:feed>
```

1. *Línea 1:* El elemento feed se encuentra en el espacio de nombres http://www.w3.org/2005/Ato
2. *Línea 2:* El elemento title también se encuentra en el espacio de nombres http://www.w3.org/2005/Atom.

En lo que concierne al analizador de XML, los dos documentos anteriores son *idénticos*. Espacio de nombres + nombre de elemento = identidad en XML. Los prefijos existen únicamente para referirse a los espacios de nombres, por lo que el prefijo utilizado en la práctica (atom:) es irrelevante. Los espacios de nombre coinciden, los nombres de elemento coinciden, los atributos (o falta de ellos) coinciden y cada contenido de texto coincide, por lo que estos dos documentos XML son el idénticos a efectos prácticos.

Finalmente, los documentos XML pueden contener en la primera línea información sobre la codificación de caracteres, antes del elemento raíz. Si tienes curiosidad sobre cómo un documento puede contener información que necesita conocerse antes de que el documento pueda analizarse consulta la Sección F de la especificación XML (http://www.w3.org/TR/REC-xml/#sec-guessing-no-ext-info) para ver los detalles sobre cómo resolver este problema.

```
1 <?xml version='1.0' encoding='utf-8'?>
```

Y con esto ya conoces suficiente XML como para ¡ser peligroso!

12.3. La estructura de una fuente de información Atom

Piensa en un blog o en cualquier sitio web que tenga contenido frecuentemente actualizado como CNN.com. El propio sitio dispone de un título (CNN.com), un subtítulo (Breaking News, U.S., World, Weather, Entertaintment y Video News), una fecha de última actualización (actualizado a 12:43 p.m. EDT, Sat May 16, 2009) y una lista de artículos publicados en diferente momentos. Cada artículo, a su vez, tiene título, una fecha de primera publicación (y posiblemente una fecha de última actualización, si se publicó una corrección) y una URL única.

El formato de sindicación de contenidos Atom está diseñado para capturar toda esta información en un formato estándar. Mi blog y CNN.com son muy diferentes en diseño, ámbito y audiencia; pero ambos tienen la misma estructura básica. CNN.com tiene un título, mi blog tiene un título. CNN.com publica artículos, yo publico artículos.

En el nivel más alto existe el *elemento raíz*, que toda fuente Atom comparte: el elemento feed del espacio de nombres http://www.w3.org/2005/Atom.

```
1 <feed xmlns='http://www.w3.org/2005/Atom'
2       xml:lang='en'>
```

1. *Línea 1:* El espacio de nombres de Atom es http://www.w3.org/2005/Atom.

2. *Línea 2:* Cualquier elemento puede contener un atributo xml:lang que sirve para declarar el idioma del elemento y de sus hijos. En este caso, el atributo xml:lang se declara una única vez en el elemento raíz, lo que significa que toda la fuente se encuentra en inglés.

Una fuente Atom contiene diversas partes de información sobre la propia fuente. Se declaran como hijas del elemento raíz feed.

```
1 <feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'>
2   <title>dive into mark</title>
3   <subtitle>currently between addictions</subtitle>
4   <id>tag:diveintomark.org,2001-07-29:/</id>
5   <updated>2009-03-27T21:56:07Z</updated>
6   <link rel='alternate' type='text/html' href='http://diveintomark.org/'/>
```

1. *Línea 2:* El título de esta fuente es dive into mark.

2. *Línea 3:* El subtítulo es currently between addictions.

3. *Línea 4:* Toda fuente necesita un identificador único global. Hay que mirar la RFC 4151[1] para ver cómo crear uno.

4. *Línea 5:* Esta fuente fue actualizada por última vez el 27 de marzo de 2009 a las 21:56 GMT. Normalmente es equivalente a la fecha de última modificación del artículo más reciente.

5. *Línea 6:* Ahora las cosas comienzan a ponerse interesantes. Esteelemento link no tiene contenido de texto, pero tiene tres atributos: rel, type y href. El valor de rel indica la clase de enlace que es; rel='alternate' significa que es un enlace a una representación alternativa de esta fuente. El atributo type='text/html' significa que es un enlace a una página HTML. Por último, el destino del enlace se indica en el atributo href.

Ahora ya conocemos que esta fuente lo es de un sitio denominado "dive into mark" que está disponible en http://diveintomark.org y que fue actualizada por última vez el 27 de marzo de 2009.

Aunque el orden de los elementos puede ser relevante en algunos documentos XML, no es relevante en una fuente Atom.

[1] http://www.ietf.org/rfc/rfc4151.txt

Después de los metadatos de la fuente se encuentra una lista con los artículos más recientes. Un artículo se representa así:

```
1  <entry>
2    <author>
3      <name>Mark</name>
4      <uri>http://diveintomark.org/</uri>
5    </author>
6    <title>Dive into history, 2009 edition</title>
7    <link rel='alternate' type='text/html'
8       href='http://diveintomark.org/archives/2009/03/27/
9  dive-into-history-2009-edition'/>
10   <id>tag:diveintomark.org,2009-03-27:/archives/20090327172042</id>
11   <updated>2009-03-27T21:56:07Z</updated>
12   <published>2009-03-27T17:20:42Z</published>
13   <category scheme='http://diveintomark.org' term='diveintopython'/>
14   <category scheme='http://diveintomark.org' term='docbook'/>
15   <category scheme='http://diveintomark.org' term='html'/>
16   <summary type='html'>Putting an entire chapter on one page sounds
17     bloated, but consider this &amp;amp;mdash; my longest chapter so far
18     would be 75 printed pages, and it loads in under 5 seconds&amp;amp;hellip;
19     On dialup.</summary>
20 </entry>
```

1. *Línea 2:* El elemento author indica quién escribió este artículo: un individuo llamado Mark, a quién puedes encontrar en http://diveintomark.org/ (Es el mismo sitio que el enlace alternativo para la fuente, pero no tiene porqué serlo. Muchos blogs tienen varios autores, cada uno con su propio sitio web personal).

2. *Línea 6:* El elemento title indica el título del artículo. "Dive into history, 2009 edition".

3. *Línea 7:* Como con el enlace alternativo en el nivel de la fuente, este elemento link indica la dirección de la versión HTML de este artículo.

4. *Línea 10:* Las entradas, como la fuente, necesitan un identificador único.

5. *Línea 11:* Las entradas tienen dos fechas: la fecha de primera publicación (published) y la fecha de última modificación (updated).

6. *Línea 13:* Las entradas pueden tener un número arbitrario de categorías. Este artículo está archivado bajo las categorías diveintopython, docbook y html.

7. *Línea 16:* El elemento summary ofrece un breve resumen del artículo (Existe también un elemento content, que no se muestra aquí, por si quieres incluir el texto completo del artículo en tu fuente). Este resumen tiene el atributo

específico de Atom type='html' que indica que este resumen está escrito en HTML, no es texto plano. Esto es importante puesto que existen entidades específicas de HTML e el texto (— y …) que se deben mostrar como "—" y "..." en lugar de que se muestre el texto directamente.

8. *Línea 20:* Por último, la etiqueta de cierre del elemento entry, que señala el final de los metadatos de este artículo.

12.4. Análisis de XML

Python puede analizar documentos XML de diversas formas. Dispone de analizadores DOM y SAX como otros lenguajes, pero me centraré en una librería diferente denominada ElementTree.

```
1 >>> import xml.etree.ElementTree as etree
2 >>> tree = etree.parse('examples/feed.xml')
3 >>> root = tree.getroot()
4 >>> root
5 <Element {http://www.w3.org/2005/Atom}feed at cd1eb0>
```

1. *Línea 1:* La librería ElementTree forma parte de la librería estándar de Python, se encuentra en xml.etree.ElementTree.

2. *Línea 2:* El punto de entrada primario de la librería es la función parse() que puede tomar como parámetro el nombre de un fichero o un objeto de flujo. Esta función analiza el documento entero de una vez. Si la memoria es escasa, existen formas para analiar un documento XML de forma incremental[2].

3. *Línea 3:* La función parse() devuelve un objeto que representa al documento completo. *No* es el elemento raíz. Para obtener una referencia al elemento raíz, debes llamar al método getroot().

4. *Línea 4:* Como cabría esperar, el elemento raíz es el elemento feed del espacio de nombres http://www.w3.org/2005/Atom. La representación en cadena de texto de este elemento incide en un punto importante: un elemento XML es una combinación de su espacio de nombres y la etiqueta de su nombre (también de nominado el *nombre local*). todo elemento de este documento se encuentra en el espacio de nombres Atom, por lo que el elemento raíz se representa como {http://www.w3.org/2005/Atom}feed.

[2] http://effbot.org/zone/element-iterparse.htm

ElementTree represena a los elementos XML como {espacio_de_nombres}nombre_local. Verás y utilizarás este formato en muchos lugares de la API de ElementTree.

12.4.1. Los elementos son listas

En la API de ElementTree los eleemntos se comportan como listas. Los elementos de la lista son los hijos del elemento.

```
1  # sigue del ejemplo anterior
2  >>> root.tag
3  '{http://www.w3.org/2005/Atom}feed'
4  >>> len(root)
5  8
6  >>> for child in root:
7  ...     print(child)
8  ...
9  <Element {http://www.w3.org/2005/Atom}title at e2b5d0>
10 <Element {http://www.w3.org/2005/Atom}subtitle at e2b4e0>
11 <Element {http://www.w3.org/2005/Atom}id at e2b6c0>
12 <Element {http://www.w3.org/2005/Atom}updated at e2b6f0>
13 <Element {http://www.w3.org/2005/Atom}link at e2b4b0>
14 <Element {http://www.w3.org/2005/Atom}entry at e2b720>
15 <Element {http://www.w3.org/2005/Atom}entry at e2b510>
16 <Element {http://www.w3.org/2005/Atom}entry at e2b750>
```

1. *Línea 2:* Continuando con el ejemplo anterior, el elemento raíz es {http://www.w3.org/2005/Ator
2. *Línea 4:* La "longitud" del elemento raíz es el número de elementos hijo.
3. *Línea 6:* Puedes utilizar el elemento como iterador para recorrer todos los elementos hijo.
4. *Línea 7:* Como ves por la salida, existen ocho elementos hijos: todos los metadatos de la fuente (title. subtitle, id, updated y link) seguidos por los tres elementos entry.

Puede que ya te hayas dado cuenta, pero quiero dejarlo explícito: la lista de los elementos hijo, únicamente incluye los hijos *directos*. cada uno de los elementos entry tiene sus propios hijos, pero no se muestran en esta lista. Estarán incluidos en la lista de hijos del elemento entry, pero no se encuentran en la lista de feed. Existen formas de encontrar elementos independientemente de los profundamente anidados que se encuentren; lo veremos más adelante en este mismo capítulo.

12.4.2. Los atributos son diccionarios

XML no solamente es una colección de elementos; cada elemento puede tener también su propio conjunto de atributos. Una vez tienes la referencia a un elemento específico puedes recuperar fácilmente sus atributos utilizando un diccionario de Python.

```
1  # sigue del ejemplo anterior
2  >>> root.attrib
3  {'{http://www.w3.org/XML/1998/namespace}lang': 'en'}
4  >>> root[4]
5  <Element {http://www.w3.org/2005/Atom}link at e181b0>
6  >>> root[4].attrib
7  {'href': 'http://diveintomark.org/',
8   'type': 'text/html',
9   'rel': 'alternate'}
10 >>> root[3]
11 <Element {http://www.w3.org/2005/Atom}updated at e2b4e0>
12 >>> root[3].attrib
```

1. *Línea 2:* La propiedad attrib es un diccionario que contiene los atributos del elemento. El texto XML original era ¡feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'¿. El prefijo xml: se refiere a un espacio de nombres interno que todo documento XML puede utilizar sin necesidad de declararlo.

2. *Línea 4:* El quinto hijo —[4] en una lista cuyo primer elemento se cuenta como cero— es el elemento link.

3. *Línea 6:* El elemento link tiene tres atributos: href, type y rel.

4. *Línea 10:* El cuarto hijo —[3]— es elemento updated.

5. *Línea 12:* El elemento updated no tiene atributos por lo que .attrib es un diccionario vacío.

12.5. Búsqueda de nodos en un documento XML

Hasta ahora hemos trabajado con este documento XML de "arriba hacia abajo", comenzando por el elemento raíz, recuperando sus hijos y luego los nietos, etc. Pero muchas aplicaciones de XML necesitan encontrar elementos específicos. ElementTree puede hacer esto también.

```
1  >>> import xml.etree.ElementTree as etree
2  >>> tree = etree.parse('examples/feed.xml')
3  >>> root = tree.getroot()
4  >>> root.findall('{http://www.w3.org/2005/Atom}entry')
5  [<Element {http://www.w3.org/2005/Atom}entry at e2b4e0>,
6   <Element {http://www.w3.org/2005/Atom}entry at e2b510>,
7   <Element {http://www.w3.org/2005/Atom}entry at e2b540>]
8  >>> root.tag
9  '{http://www.w3.org/2005/Atom}feed'
10 >>> root.findall('{http://www.w3.org/2005/Atom}feed')
11 []
12 >>> root.findall('{http://www.w3.org/2005/Atom}author')
13 []
```

1. *Línea 4:* El método findall() encuentra todos los elementos hijo que coinciden con una consulta específica (En breve veremos los formatos posibles de la consulta).

2. *Línea 10:* Cada elemento —incluido el elemento raíz, pero también sus hijos— tiene un método findall(). Permite encontrar todos los elementos que coinciden entre sus hijos. Pero ¿porqué no devuelve esta consulta ningún resultado? Aunque no sea obvio, esta consulta particular únicamente busca entre los hijos del elemento. Puesto que el elemento raíz feed no tiene ningún hijo denominado feed, esta consulta devuelve una lista vacía.

3. *Línea 12:* También te puede sorprender este resultado. Existe un elemento author en este documento; de hecho hay tres (uno en cada entry). Pero estos elementos author no son *hijos directos* el elemento raíz; son "nietos" (literalmente, un elemento hijo de otro elemento hijo). Si quieres buscar elementos author en cualquier nivel de profundidad puedes hacerlo, pero el formato de la consulta es algo distinto.

```
1  >>> tree.findall('{http://www.w3.org/2005/Atom}entry')
2  [<Element {http://www.w3.org/2005/Atom}entry at e2b4e0>,
3   <Element {http://www.w3.org/2005/Atom}entry at e2b510>,
4   <Element {http://www.w3.org/2005/Atom}entry at e2b540>]
5  >>> tree.findall('{http://www.w3.org/2005/Atom}author')
6  []
```

1. *Línea 1:* Por comodidad, el objeto tree (devuelto por la función etree.parse() tiene varios métodos que replican aquellos disponibles en el elemento raíz. Los resultados son idénticos a los que se obtienen si se llamase a tree.getroot().findall().

2. *Linea 5:* Tal vez te pueda sorprender, pero esta consulta no encuentra a los elementos author del documento. ¿Porqué no? porque es simplemente una forma de llamar a tree.getroot().findall('{http://www.w3.org/2005/Atom}author'), lo que significa "encuentra todos los elementos author que sean hijos directos del elemento raíz". Los elementos author no son hijos del elemento raíz; son hijos de los elementos entry. Por eso la consulta no retorna ninguna coincidencia.

También hay un método find() que retorna el primer elemento que coincide. Es útil para aquellas situaciones en las que únicamente esperas una coincidencia, o cuando haya varias pero solamente te importa la primera de ellas.

```
1  >>> entries = tree.findall('{http://www.w3.org/2005/Atom}entry')
2  >>> len(entries)
3  3
4  >>> title_element = entries[0].find('{http://www.w3.org/2005/Atom}title')
5  >>> title_element.text
6  'Dive into history, 2009 edition'
7  >>> foo_element = entries[0].find('{http://www.w3.org/2005/Atom}foo')
8  >>> foo_element
9  >>> type(foo_element)
10 <class 'NoneType'>
```

1. *Línea 1:* Viste esto en el ejemplo anterior. Encuentra todos los elementos atom:entry.

2. *Línea 4:* El método find() toma una consulta y retorna el primer elemento que coincide.

3. *Línea 7:* No existen elementos denominados foo por lo que retorna None.

   Hay una complicación en el método find() que te pasará en algún momento. En un contexto booleano los objetos elemento de ElementTree se evalúan a False si no tienen hijos (si len(element) es cero). Esto significa que if element.find('...') no está comprobando si el método find() encontró un elemento coincidente; está comprobando si ¡el elemento coincidente tiene algún elemento hijo! Para comprobar si el método find() retornó algún elemento debes utilizar if element.find('...') is not None.

Existe una forma de buscar entre los elementos *descendientes*: hijos, nietos y niveles más profundos de anidamiento.

```
1  >>> all_links = tree.findall('//{http://www.w3.org/2005/Atom}link')
2  >>> all_links
3  [<Element {http://www.w3.org/2005/Atom}link at e181b0>,
4   <Element {http://www.w3.org/2005/Atom}link at e2b570>,
5   <Element {http://www.w3.org/2005/Atom}link at e2b480>,
6   <Element {http://www.w3.org/2005/Atom}link at e2b5a0>]
7  >>> all_links[0].attrib
8  {'href': 'http://diveintomark.org/',
9   'type': 'text/html',
10  'rel': 'alternate'}
11 >>> all_links[1].attrib
12 {'href': 'http://diveintomark.org/archives/2009/03/27/
13    dive-into-history-2009-edition',
14  'type': 'text/html',
15  'rel': 'alternate'}
16 >>> all_links[2].attrib
17 {'href': 'http://diveintomark.org/archives/2009/03/21/
18    accessibility-is-a-harsh-mistress',
19  'type': 'text/html',
20  'rel': 'alternate'}
21 >>> all_links[3].attrib
22 {'href': 'http://diveintomark.org/archives/2008/12/18/
23    give-part-1-container-formats',
24  'type': 'text/html',
25  'rel': 'alternate'}
```

1. *Línea 1:* Esta consulta —//http://www.w3.org/2005/Atomlink— es muy similar a las anteriores, excepto por las dos barras inclinadas al comienzo de la consulta. Estas dos barras significan que "no se busque únicamente entre los hijos directos; quiero cualquier elemento que coincida *independientemente* del nivel de anidamiento". Por eso el resultado es una lista de cuatro elementos link, no únicamente uno.

2. *Línea 7:* El primer resultado es hijo directo del elemento raíz. Como puedes observar por sus atributos, es el enlace alternativo que apunta a la versión HTML del sitio web que esta fuente describe.

3. *Línea 11:* Los otros tres resultados son cada uno de los enlaces alternativos de cada entrada. cada entry tiene un único elemento hijo link. Debido a la doble barra inclinada al comienzo de la consulta, se encuentran todos estos enlaces.

En general, el método findall() de ElementTree es una característica muy potente, pero el lenguaje de consulta puede ser un poco sorprendente. Está descrito oficialmente en http://effbot.org/zone/element-xpath.htm(Soporte limitado a expresiones XPath). XPath es un estándar del W3C para consultar documentos XML.

El lenguaje de consulta de ElementTree es suficientemente parecido a XPath para poder hacer búsquedas básicas, pero también suficientemente diferente como para desconcertarte si ya conoces XPath.

Ahora vamos a ver una librería de terceros que extiende la API de ElementTree para proporcionar un soporte completo de XPath.

12.6. Ir más allá con LXML

lxml[3] es una librería de terceros de código abierto que se desarrolla sobre el popular analizador libxml2[4]. Proporciona una API que es 100 % compatible con ElementTree, y la extiende con soporte completo a Xpath 1.0 y otras cuantas bondades. Existe un instalador disponible para Windows[5]; los usuarios de Linux siempre deberían intentar usar las herramientas específicas de la distribución como yum o apt-get para instalar los binarios precompilados desde sus repositorios. En otro caso, necesitarás instalar los binarios manualmente[6].

```
1 >>> from lxml import etree
2 >>> tree = etree.parse('examples/feed.xml')
3 >>> root = tree.getroot()
4 >>> root.findall('{http://www.w3.org/2005/Atom}entry')
5 [<Element {http://www.w3.org/2005/Atom}entry at e2b4e0>,
6  <Element {http://www.w3.org/2005/Atom}entry at e2b510>,
7  <Element {http://www.w3.org/2005/Atom}entry at e2b540>]
```

1. *Línea 1:* Una vez importado, lxml proporciona la misma API que la librería estándar ElementTree.
2. *Línea 2:* La función parse(), igual que en ElementTree.
3. *Línea 3:* El método getroot(), también igual.
4. *Línea 4:* El método findall(), exactamente igual.

Para documentos XML grandes, lxml es significativamente más rápido que la librería ElementTree. Si solamente estás utilizando la API ElementTree y quieres usar la implementación más rápida existente, puedes intentar importar lxml y de no estar disponible, usar como segunda opción ElementTree.

[3] http://codespeak.net/lxml/
[4] http://www.xmlsoft.org/
[5] http://pypi.python.org/pypi/lxml/
[6] http://codespeak.net/lxml/installation.html

```
1 try:
2     from lxml import etree
3 except ImportError:
4     import xml.etree.ElementTree as etree
```

Pero lxml proporciona algo más que el ser más rápido que ElementTree. Su método findall() incluye el soporte de expresiones más complicadas.

```
1 >>> import lxml.etree
2 >>> tree = lxml.etree.parse('examples/feed.xml')
3 >>> tree.findall('//{http://www.w3.org/2005/Atom}*[@href]')
4 [<Element {http://www.w3.org/2005/Atom}link at eeb8a0>,
5  <Element {http://www.w3.org/2005/Atom}link at eeb990>,
6  <Element {http://www.w3.org/2005/Atom}link at eeb960>,
7  <Element {http://www.w3.org/2005/Atom}link at eeb9c0>]
8 >>> tree.findall("//{http://www.w3.org/2005/Atom}*" \
9                  "[@href='http://diveintomark.org/']")
10 [<Element {http://www.w3.org/2005/Atom}link at eeb930>]
11 >>> NS = '{http://www.w3.org/2005/Atom}'
12 >>> tree.findall('//{NS}author[{NS}uri]'.format(NS=NS))
13 [<Element {http://www.w3.org/2005/Atom}author at eeba80>,
14  <Element {http://www.w3.org/2005/Atom}author at eebba0>]
```

1. *Línea 1:* En este ejemplo voy a importar lxml.tree en lugar de utilizar from lxml import etree, para destacar que estas características son específicas de lxml.

2. *Línea 3:* Esta consulta encuentra todos los elementos del espacio de nombres Atom, en cualquier sitio del documento, que contengan el atributo href. Las // al comienzo de la consulta significa "elementos en cualquier parte del documento (no únicamente los hijos del elemento raíz)". {http://www.w3.org/2005/Atom} significa "únicamente los elementos en el espacio de nombres de Atom". * significa "elementos con cualquier nombre local" y @href significa "tiene un atributo href".

3. *Línea 8:* La consulta encuentra todos los elementos Atom con el atributo href cuyo valor sea http://diveintomark.org/.

4. *Línea 11:* Después de hacer un rápido formateo de las cadenas de texto (porque de otro modo estas consultas compuestas se vuelven ridículamente largas), esta consulta busca los elementos author que tienen un elemento uri como hijo. Solamente retorna dos elementos author, los de la primera y segunda entry. El author del último entry contiene únicamente el elemento name, no uri.

¿No es suficiente para tí? lxml tampoco integra soporte de expresiones XPath 1.0. No voy a entrar en profundidad en la sintaxis de XPath; se podría escribir un libro entero sobre ello. Pero te mostraré cómo se integra en lxml.

```
1  >>> import lxml.etree
2  >>> tree = lxml.etree.parse('examples/feed.xml')
3  >>> NSMAP = {'atom': 'http://www.w3.org/2005/Atom'}
4  >>> entries = tree.xpath("//atom:category[@term='accessibility']/..",
5  ...     namespaces=NSMAP)
6  >>> entries
7  [<Element {http://www.w3.org/2005/Atom}entry at e2b630>]
8  >>> entry = entries[0]
9  >>> entry.xpath('./atom:title/text()', namespaces=NSMAP)
10 ['Accessibility is a harsh mistress']
```

1. *Línea 3:* Para realizar consultas XPath en elementos de un espacio de nombres, necesitas definir dichos espacios de nombre con el mapeo a sus alias. Esto se realiza con un diccionario de Python.

2. *Línea 4:* Esto es una consulta XPath. La expresión XPath busca elementos category (en el espacio de nombres de Atom) que contengan el atributo term con el valor accesibility. Pero ése no es el resultado real de la consulta. Observa el final de la cadena de texto de la consulta; ¿observaste el trozo /..? Significa que "devuelve el elemento padre del elemento category que se acaba de encontrar". Así esta consulta XPath encontrará todas las entradas que tengan un hijo ¡category term='accessibility'¿.

3. *Línea 6:* La función xpath() devuelve una lista de objetos ElementTree. En este documento, únicamente hay una entrada con un elemento category cuyo term sea accesibility.

4. *Línea 9:* Las expresiones XPath no siempre devuelven una lista de elementos. Técnicamente, el modelo DOM de un documento XML no contiene elementos, contiene *nodos*. Dependiendo de su tipo, los nodos pueden ser elementos, atributos o incluso contenido de texto. El resultado de una consulta XPath es una lista de nodos. Esta consulta retorna una lista de nodos de texto: el contenido de texto (text()) del elemento title (atom:title) que sea hijo del elemento actual (./).

12.7. Generación de XML

El soporte a XML de Python no está limitado al análisis de documentos existentes. Puedes crear también documentos XML desde cero.

```
1 >>> import xml.etree.ElementTree as etree
2 >>> new_feed = etree.Element('{http://www.w3.org/2005/Atom}feed',
3 ...       attrib={'{http://www.w3.org/XML/1998/namespace}lang': 'en'})
4 >>> print(etree.tostring(new_feed))
5 <ns0:feed xmlns:ns0='http://www.w3.org/2005/Atom' xml:lang='en'/>
```

1. *Línea 2:* Para crear un elemento nuevo, se debe instanciar un objeto de la clase Element. Se le pasa el nombre del elemento (espacio de nombres + nombre local) como primer parámetro. Esta sentencia crear un elemento feed en el espacio de nombres Atom. Esta será nuestro elemento raíz del nuevo documento.

2. *Línea 3:* Para añadir atributos al elemento, se puede pasar un diccionario de nombres y valores de atributos en el parámetro attrib. Observa que el nombre del atributo debe estar en el formato estándar de ElementTree, {espacio_de_nombres}nombre_loca

3. *Línea 4:* En cualquier momento puedes serializar cualquier elemento (y sus hijos) con la función tostring() de ElementTree.

¿Te ha sorprendido el resultado de la serialización? La forma en la que ElementTree serializa los elementos con espacios de nombre XML es técnicamente precisa pero no óptima. El documento XML de ejemplo al comienzo del capítulo definió un espacio de nombres por defecto (xmlns='http://www.w3.org/2005/Atom'). La definición de un espacio de nombres por defecto es útil para documentos —como las fuentes Atom— en los que todos, o la mayoría de, los elementos pertenecen al mismo espacio de nombres, porque puedes declarar el espacio de nombres una única vez y declarar cada elemento únicamente con su nombre local (¡feed¿, ¡link¿, ¡entry¿). No hay necesidad de utilizar prefijos a menos que quieras declarar elementos de otro espacio de nombres.

Un analizador XML no verá ninguna diferencia entre un documento XML con un espacio de nombres por defecto y un documento XML con un espacio de nombres con prefijo. El DOM resultante de esta serialización:

```
1 <ns0:feed xmlns:ns0='http://www.w3.org/2005/Atom' xml:lang='en'/>
```

es idéntico al DOM de esta otra:

```
1 <feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'/>
```

La única diferencia práctica es que la segunda serialización es varios caracteres más corta. Si tuvieramos que modificar nuestro ejemplo para añadirle el prefijo ns0: en cada etiqueta de inicio y fin, serían 4 caracteres por cada etiqueta de inicio x 79 etiquetas + 4 caracteres por la propia declaración del espacio de nombres, en total son 320 caracteres más. En el caso de que asumamos una codificación de caracteres

UTF8 se trata de 320 bytes extras (después de comprimir la diferencia se reduce a 21 bytes). Puede que no te importe mucho, pero para una fuente Atom, que puede descargarse miles de veces cada vez que cambia, una diferencia de unos cuantos bytes por petición puede suponer una cierta diferencia.

La librería ElementTree no ofrece un control fino sobre la serialización de los elementos con espacios de nombres, pero lxml sí:

```
1 >>> import lxml.etree
2 >>> NSMAP = {None: 'http://www.w3.org/2005/Atom'}
3 >>> new_feed = lxml.etree.Element('feed', nsmap=NSMAP)
4 >>> print(lxml.etree.tounicode(new_feed))
5 <feed xmlns='http://www.w3.org/2005/Atom'/>
6 >>> new_feed.set('{http://www.w3.org/XML/1998/namespace}lang', 'en')
7 >>> print(lxml.etree.tounicode(new_feed))
8 <feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'/>
```

1. *Línea 2:* Para comenzar, se define el mapeo de los espacios de nombre como un diccionario. Los valores del diccionario son espacios de nombres; las claves son el prefijo deseao. Utilizar None como prefijo, sirve para declarar el espacio de nombres por defecto.

2. *Línea 3:* Ahora puedes pasar el parámetro nsmap, que es específico de lxml, cuando vayas a crear un elemento, y lxml respectará los prefijos que hayas definido.

3. *Línea 4:* Como se esperaba, esta serialización define el espacio de nombres Atom como el espacio de nombres por defecto y declara el elemento feed sin prefijo.

4. *Línea 6:* ¡Ups! Olvidamos añadir el atributo xml:lang. Siempre puedes añadir atributos a cualquier elemento con el método set(). Toma dos parámetros, el nombre del atributo en formato estándar de ElementTree y el valor del atributo. Este método no es específico de lxml, lo único específico de lxml en este ejemplo es la parte del parámetro nsmap para controlar los prefijos de la salida serializada.

¿Están los documentos XML limitados a un elemento por documento? Por supuesto que no. Puedes crear hijos de forma fácil.

```
1  >>> title = lxml.etree.SubElement(new_feed, 'title',
2  ...      attrib={'type':'html'})
3  >>> print(lxml.etree.tounicode(new_feed))
4  <feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'>
5  <title type='html'/></feed>
6  >>> title.text = 'dive into &hellip;'
7  >>> print(lxml.etree.tounicode(new_feed))
8  <feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'>
9  <title type='html'>dive into &amp;hellip;</title></feed>
10 >>> print(lxml.etree.tounicode(new_feed, pretty_print=True))
11 <feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'>
12 <title type='html'>dive into&amp;hellip;</title>
13 </feed>
```

1. *Línea 1:* Para crear elementos hijo de un elemento existente, instancia objetos de la clase SubElement. Los parámetros necesarios son el elemento padre (new_feed en este caso) y el nombre del nuevo elemento. Puesto que los elementos hijo heredan el espacio de nombres de sus padres, no hay necesidad de redeclarar el espacio de nombres o sus prefijos.

2. *Línea 2:* Puedes pasarle un diccionario de atributos. Las claves son los nombres de los atributos y los valores son los valores de los atributos.

3. *Línea 3:* Como esperabas, el nuevo elemento title se ha creado en el espacio de nombres Atom y fue insertado como hijo del elemento feed. Puesto que el elemento title no tiene contenido de texto y no tiene hijos por sí mismo, lxml lo serializa como un elemento vacío (con /¿).

4. *Línea 6:* Para establecer el contenido de texto de un elemento basta con asignarle valor a la propiedad .text.

5. *Línea 7:* Ahora el elemento title se serializa con su contenido de texto. Cualquier contenido de texto que contenga símbolos 'menor que' o ampersands necesitan 'escaparse' al serializarse. lxml hace estas conversiones de forma automática.

6. *Línea 10:* Puedes aplicar una impresión formateada a la serialización, lo que inserta los saltos de línea correspondientes al cambiar las etiquetas. En términos técnicos, lxml añade espacios en blanco no significativos para hacer más legible la salida resultante.

   Podrías querer echarle un vistazo a xmlwitch[7], otra librería de terceros para generar XML. Hace uso extensivo de la sentencia with para hacer la generación de código XML más legible.

[7] http://github.com/galvez/xmlwitch/tree/master

12.8. Análisis de XML "estropeado"

La especificación XML obliga a que todos los analizadores XML empleen un manejo de errores "draconiano". Esto es, deben parar tan pronto como detecten cualquier clase de error de "malformado" del documento. Errores de mala formación del documento son: que las etiquetas de inicio y fin no se encuentren bien balanceadas, entidades sin definir, caracteres unicode ilegales y otro número de reglas esotéricas. Esto es un contraste importante con otros formatos habituales como HTML —tu navegador no para de mostrar una página web si se te olvida cerrar una etiqueta HTML o aparece un escape o ampersand en el valor de un atributo (Es un concepto erróneo bastante extendido que HTML no tiene definida una forma de hacer manejo de errores. Sí que está bien definido, pero es significativamente más complejo que "párate ante el primer error que encuentres".

Algunas personas (yo mismo incluido) creen que fue un error para los inventores del XML obligar a este manejo de errores "draconianos". No me malinterpretes; puedo comprender el encanto de la simplificación de las reglas de manejo de errores. Pero en la práctica, el concepto de "bien formado" es más complejo de lo que suena, especialmente para aquellos documentos XML (como los documentos Atom) se publican en la web mediante un servidor HTTP. A pesar de la madurez de XML, cuyo manejo estandarizado de errores es de 1997, las encuestas muestran continuamente que una significativa fracción de fuentes Atom de la web están plagadas con errores de "buena formación".

Por eso, tengo razones teóricas y prácticas para analizar documentos XML a "cualquier precio", esto es, para *no* parar ante el primer error de formación. Si te encuentras tú mismo en esta situación, lxml puede ayudar.

Aquí hay un fragmento de un documento XML mal formado. El ampersand debería estar "escapado".

```
1 <?xml version='1.0' encoding='utf-8'?>
2 <feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'>
3   <title>dive into &hellip;</title>
4 ...
5 </feed>
```

Eso es un error, porque la entidad … no está definida en XML (está definida en HTML). Si intentas analizar este documento XML con los valores por defecto, lxml parará en la entidad sin definir.

```
>>> import lxml.etree
>>> tree = lxml.etree.parse('examples/feed-broken.xml')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "lxml.etree.pyx", line 2693,
       in lxml.etree.parse (src/lxml/lxml.etree.c:52591)
  File "parser.pxi", line 1478,
       in lxml.etree._parseDocument (src/lxml/lxml.etree.c:75665)
  File "parser.pxi", line 1507,
       in lxml.etree._parseDocumentFromURL (src/lxml/lxml.etree.c:75993)
  File "parser.pxi", line 1407,
       in lxml.etree._parseDocFromFile (src/lxml/lxml.etree.c:75002)
  File "parser.pxi", line 965,
       in lxml.etree._BaseParser._parseDocFromFile
       (src/lxml/lxml.etree.c:72023)
  File "parser.pxi", line 539,
       in lxml.etree._ParserContext._handleParseResultDoc
       (src/lxml/lxml.etree.c:67830)
  File "parser.pxi", line 625,
       in lxml.etree._handleParseResult (src/lxml/lxml.etree.c:68877)
  File "parser.pxi", line 565,
       in lxml.etree._raiseParseError (src/lxml/lxml.etree.c:68125)
lxml.etree.XMLSyntaxError:
       Entity 'hellip' not defined, line 3, column 28
```

Para analizar este documento, a pesar de su error de buena formación, necesitas crear un analizador XML específico.

```
>>> parser = lxml.etree.XMLParser(recover=True)
>>> tree = lxml.etree.parse('examples/feed-broken.xml', parser)
>>> parser.error_log
examples/feed-broken.xml:3:28:FATAL:PARSER:ERR_UNDECLARED_ENTITY:
    Entity 'hellip' not defined
>>> tree.findall('{http://www.w3.org/2005/Atom}title')
[<Element {http://www.w3.org/2005/Atom}title at ead510>]
>>> title = tree.findall('{http://www.w3.org/2005/Atom}title')[0]
>>> title.text
'dive into '
>>> print(lxml.etree.tounicode(tree.getroot()))
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'>
  <title>dive into </title>
.
. [resto de la serialización suprimido por brevedad]
.
```

1. *Línea 1:* Para crear un analizador específico, se debe instanciar la clase lxml.etree.XMLParser. Puede recibir un número diferente de parámetros. Nos interesa ahora el paráme-

tro recover. Cuando se establece a True, el analizador XML intentará "recuperarse" de este tipo de errores.

2. *Línea 2:* Para analizar un documento XML con este analizador, basta con pasar este objeto parser como segundo parámetro de la función parse(). Observa que lxml no eleva ninguna excepción sobre la entidad no definida ….

3. *Línea 3:* Aún así, el analizador mantiene un registro de los errores de formación que ha encontrado (Esto siempre es cierto independientemente de que esté activado para recuperarse de esos errores o no).

4. *Línea 9:* Puesto que no supo que hacer con la entidad sin definir …, el analizador simplemente la descarta silenciosamente. El contenido de texto del elemento title se convierte en 'dive into '.

5. *Línea 11:* Como puedes ver en la serialización, la entidad … ha sido suprimida.

Es importante reiterar que **no existe garantía de interoperabilidad** entre analizadores XML que se recuperan de los errores. Una analizador diferente podría decidir que reconoce la entidad … de HTML y reemplazarla por …? ¿Es esto mejor? Puede ser. ¿Es más correcto? No, ambas soluciones son igualmente erróneas. El comportamiento correcto (de acuerdo a la especificación XML) es pararse y elevar el error. Si has decidido que no es lo que quieres hacer, lo haces bajo tu propia responsabilidad.

12.9. Lecturas recomendadas

- XML en la wikipedia.org:
  http://en.wikipedia.org/wiki/XML
- La API de ElementTree:
- Elementos y árboles de elementos
  http://effbot.org/zone/element.htm
- Soporte de XPath en ElementTree
  http://effbot.org/zone/element-xpath.htm
- La función iterparse de ElementTree
  http://effbot.org/zone/element-iterparse.htm

- lxml
  http://codespeak.net/lxml/
- Análisis de XML y HTML con lxml
  http://codespeak.net/lxml/1.3/parsing.html
- XPath y XSLT con lxml
  http://codespeak.net/lxml/1.3/xpathxslt.html
- xmlwitch
  http://github.com/galvez/xmlwitch/tree/master

Capítulo 13

Serialización de Objetos en Python

Nivel de dificultad:◆◆◆◆◇

"Desde que vivimos en este apartamento, cada sábado me he levantado a las 6:15, me he preparado un tazón de cereales con leche, me he sentado en este lado de este sofá, he puesto la BBC America, y he visto Doctor Who."
—Sheldon, La teoría del Big Bang.[1]

13.1. Inmersión

El concepto de la serialización es simple. Tienes una estructura de datos en memoria que quieres grabar, reutilizar o enviar a alguien. ¿Cómo lo haces? Bueno, eso depende de lo que quieras grabar, de cómo lo quieras reutilizar y a quién se lo envías. Muchos juegos te permiten grabar el avance cuando sales de ellos, y continuar en donde lo dejaste cuando los vuelves a cargar (En realidad, esto también lo hacen las aplicaciones que no son de juegos). En estos casos, se necesita almacenar en disco una estructura de datos que almacena "tu grado de avance hasta el momento", cuando los juegos se reinician, es necesario volver a cargar estas estructuras de datos. Los datos, en estos casos, sólo se utilizan por el mismo programa que los creó, no se envían por la red ni se leen por nadie más que por el programa que los creó. Por ello, los posibles problemas de interoperabilidad quedan reducidos a asegurar

[1] http://en.wikiquote.org/wiki/The_Big_Bang_Theory#The_Dumpling_Paradox_.5B1.07.5D

que versiones posteriores del mismo programa pueden leer los datos escritos por versiones previas.

Para casos como estos, el módulo pickle es ideal. Forma parte de la librería estándar de Python, por lo que siempre está disponible. Es rápido, la mayor parte está escrito en C, como el propio intérprete de Python. Puede almacenar estructuras de datos de Python todo lo complejas que se necesite.

¿Qué puede almacenar el módulo pickle?

- Todos los *tipos de datos* nativos que Python soporta: booleanos, enteros, números de coma flotante, números complejos, cadenas, objetos bytes, arrays de byte y None.
- Listas, tuplas, diccionarios y conjuntos que contengan cualquier combinación de tipos de dato nativos.
- Listas, tuplas, diccionarios y conjuntos de datos que contengan cualquier combinación de listas, tiplas, diccionarios y conjuntos conteniendo cualquier combinación de tipos de datos nativos (y así sucesivamente, hasta alcanzar un máximo nivel de anidamiento[2]).
- Funciones, clases e instancias de clases (con ciertas limitaciones).

Si no es suficiente para ti, el módulo pickle se puede extender. Si estás interesado en la extensibilidad, revisa los enlaces de la sección de *Lecturas recomendadas* al final de este capítulo.

13.1.1. Una nota breve sobre los ejemplos de este capítulo

Este capítulo cuenta una historia con dos consolas de Python. Todos los ejemplos de este capítulo son parte de una única historia. Se te pedirá que vayas pasando de una consola a otra de Python para demostrar el funcionamiento de los módulos pickle y json.

Para ayudarte a mantener las cosas claras, abre la consola de Python y define la siguiente variable:

```
1 >>> shell = 1
```

Mantén la ventana abierta. Ahora abre otra consola de Python y define la siguiente variable:

[2] http://docs.python.org/3.1/library/sys.html#sys.getrecursionlimit

```
1 >>> shell = 2
```

A lo largo de este capítulo, utilizaré la variable shell para inidcar en qué consola de Python se ejecuta cada ejemplo.

13.2. Almacenamiento de datos a un fichero "pickle"

El módulo pickle funciona con estructuras de datos. Vamos a construir una.

```
1  >>> shell
2  1
3  >>> entry = {}
4  >>> entry['title'] = 'Dive into history, 2009 edition'
5  >>> entry['article_link'] = 'http://diveintomark.org/' + \
6          'archives/2009/03/27/dive-into-history-2009-edition'
7  >>> entry['comments_link'] = None
8  >>> entry['internal_id'] = b'\xDE\xD5\xB4\xF8'
9  >>> entry['tags'] = ('diveintopython', 'docbook', 'html')
10 >>> entry['published'] = True
11 >>> import time
12 >>> entry['published_date'] = \
13         time.strptime('Fri Mar 27 22:20:42 2009')
14 >>> entry['published_date']
15 time.struct_time(tm_year=2009, tm_mon=3, tm_mday=27,
16                   tm_hour=22, tm_min=20, tm_sec=42,
17                   tm_wday=4, tm_yday=86, tm_isdst=-1)
```

1. *Línea 1:* Tecléalo en la consolo #1.
2. *Línea 3:* La idea aquí es construir un diccionario de Python que pueda representar algo que sea útil, como una *entrada de una fuente Atom*. Pero también quiero asegurarme de que contiene diferentes tipos de datos para mostrar el funcionamiento del módulo pickle. No entres demasiado en los valores concretos.
3. *Línea 13:* El módulo time contiene una estructura de datos (time_struct) que representa un punto en el tiempo (con una precisión de milisegundo) y funcionees que sirven para manipular estructuras de este tipo. La función strptime() recibe una cadena formateada y la convierte en una estructura time_struct. Esta cadena se encuentra en el formato por defecto, pero puedes controlarlo con los códigos de formato. Para más detalles consulta el módulo time[3].

[3] http://docs.python.org/3.1/library/time.html

Bueno, ya tenemos un estupendo diccionario de Python. Vamos a salvarlo en un fichero.

```
1 >>> shell
2 1
3 >>> import pickle
4 >>> with open('entry.pickle', 'wb') as f:
5 ...     pickle.dump(entry, f)
6 ...
```

1. *Línea 1:* Seguimos en la consola #1.
2. *Línea 4:* Utiliza la función open() para abrir un fichero. El modo de apertura es 'wb', de escritura y en binario. Lo envolvemos en una sentencia with para asegurar que el fichero se cierra automáticamente al finalizar su uso.
3. *Línea 5:* La función dump() del módulo pickle toma una estructura de datos serializable de Python y la serializa a un formato binario, específico de Python, y almacena el resultado en el fichero abierto.

Esa última sentencia era muy importante.

- El módulo pickle toma una estructura de datos Python y la salva a un fichero.
- Para hacer esto, serializa la estructura de datos utilizando un formato de datos denominado "el protocolo pickle".
- Este protocolo es específico de Python; no existe ninguna garantía de compatibilidad entre lenguajes de programación. Probablemente no puedas abrir el fichero entry.pickle con Perl, PHP, Java u otro lenguaje.
- No todas las estructuras de datos de Python se pueden serializar con el módulo pickle. El protocolo pickle ha cambiado varias veces para acomodar nuevos tipos de datos que se han ido añadiendo a Python, pero aún tiene limitaciones.
- Como resultado de estos cambios, no existe garantía de compatibilidad entre diferentes versiones de Python. Las versiones nuevas de Python soportan los formatos antiguos de serialización, pero las versiones viejas de Python no soportan los formatos nuevos (puesto que no soportan los tipos de datos nuevos).
- A menos que especifiques otra cosa, las funciones del módulo pickle utilizarán la última versión del protocolo pickle. Esto asegura que dispones de la máxima flexibilidad en los tipos de datos que puedes serializar, pero también significa que el fichero resultante no podrá leerse en versiones de Python más antiguas que no soporten la última versión del protocolo.

- La última versión del protocolo pickle es un formato binario. Asegúrate de abrir el fichero en modo binario o los datos se corromperán durante la escritura.

13.3. Carga de datos de un fichero "pickle"

Ahora cambia a la segunda consola de Python —la otra, la que no utilizaste para crear el diccionario entry.

```
1  >>> shell
2  2
3  >>> entry
4  Traceback (most recent call last):
5    File "<stdin>", line 1, in <module>
6  NameError: name 'entry' is not defined
7  >>> import pickle
8  >>> with open('entry.pickle', 'rb') as f:
9  ...     entry = pickle.load(f)
10 ...
11 >>> entry
12 {'comments_link': None,
13  'internal_id': b'\xDE\xD5\xB4\xF8',
14  'title': 'Dive into history, 2009 edition',
15  'tags': ('diveintopython', 'docbook', 'html'),
16  'article_link':
17  'http://diveintomark.org/archives/2009/03/27/
18   dive-into-history-2009-edition',
19  'published_date': time.struct_time(tm_year=2009, tm_mon=3,
20  tm_mday=27, tm_hour=22, tm_min=20, tm_sec=42, tm_wday=4,
21  tm_yday=86, tm_isdst=-1),
22  'published': True}
```

1. *Línea 1:* Consola #2.

2. *Línea 3:* La variable entry no está definida en esta consola. La definimos en la consola #1, que se trata de un entorno totalmente separado de éste y tiene su propio estado.

3. *Línea 8:* Se abre el fichero entry.pickle que creamos con la consola #1. El módulo pickle utiliza un formato binario, por lo que siempre hay que abrir los ficheros de este tipo en modo binario.

4. *Línea 9:* La función pickle.load() toma un objeto stream, lee los datos serializados del stream, crea un nuevo objeto Python, recrea los datos serializados en el nuevo objeto Python y devuelve el objeto.

5. *Línea 11:* Ahora la variable entry contiene un diccionario con las claves y valores que nos son familiares de la otra consola.

El ciclo pickle.dump() / pickle.load() da como resultado una estructura de datos nueva que es igual a la original.

```
1 >>> shell
2 1
3 >>> with open('entry.pickle', 'rb') as f:
4 ...     entry2 = pickle.load(f)
5 ...
6 >>> entry2 == entry
7 True
8 >>> entry2 is entry
9 False
10 >>> entry2['tags']
11 ('diveintopython', 'docbook', 'html')
12 >>> entry2['internal_id']
13 b'\xDE\xD5\xB4\xF8'
```

1. *Línea 1:* Volvemos a la consola #1.

2. *Línea 3:* Abrimos el fichero entry.pickle.

3. *Línea 4:* Cargamos los datos serializados en la nueva variable entry2.

4. *Línea 6:* Python confirma que los dos diccionarios, entry y entry2, son iguales. En esta consola, construimos el diccionario almacenado en entry desde cero, creando un diccionario vacío y añadiéndole valores poco a poco. Serializamos el diccionario y lo almacenamos en el fichero entry.pickle. Ahora hemos recuperado los datos serializados desde el fichero y hemos creado una réplica perfecta de la estructura de datos original.

5. *Línea 8:* Igualdad no es lo mismo que identidad. Como he dicho, hemos creado una *réplica perfecta* de los datos originales. Pero son una copia.

6. *Línea 10:* Por razones que aclararé más tarde en el capítulo, he querido mostrar que el valor de la clave 'tags' es una tupla, y el valor de la clave 'internal_id' es un objeto bytes.

13.4. Serialización con "pickle" sin pasar por un fichero

Los ejemplos de la sección anterior te mostraron cómo serializar un objeto Python directamente a un fichero en disco. Pero ¿qué sucede si no necesitas un fichero? Puedes serializar a un objeto bytes en memoria.

```
1 >>> shell
2 1
3 >>> b = pickle.dumps(entry)
4 >>> type(b)
5 <class 'bytes'>
6 >>> entry3 = pickle.loads(b)
7 >>> entry3 == entry
8 True
```

1. *Línea 3:* La función pickle.dumps() (observa la 's' al final del nombre de la función) realiza la misma serialización que la función pickle.dump(). Pero en lugar de tomar como parámetro un objeto stream y serializar sobre él, simplemente retorna los datos serializados.

2. *Línea 4:* Puesto que el protocolo pickle utiliza un formato de datos binario, la función pickle.dumps() retorna un objeto bytes.

3. *Línea 6:* La función pickle.loads() (de nuevo, observa la 's' al final del nombre de la función) realiza la misma operación que la función pickle.load(). Pero en lugar de tomar como parámetro un objeto stream y leer de él los datos, toma un objeto bytes que contenga datos serializados.

4. *Línea 7:* El resultado final es el mismo: una réplica perfecta del diccionario original.

13.5. Los bytes y las cadenas de nuevo vuelven sus feas cabezas

El protocolo pickle existe desde hace muchos años, y ha madurado a la par que lo ha hecho Python. Por ello, actualmente existen cuatro versiones diferentes del protocolo.

- Python 1.x tenía dos protocolos, un formato basado en texto ("versión 0") y un formato binario ("versión 1").

- Python 2.3 introdujo un protocolo nuevo ("versión 2") para tener en cuenta la nueva funcionalidad de clases y objetos en Python. Es un formato binario.
- Python 3.0 introdujo otro protocolo ("versión 3") que soporta explícitamente los objetos bytes y arrays de byte. Es un formato binario.

Como puedes observar, la diferencia existente entre cadenas de texto y bytes vuelve a aparecer (si te sorprende es que no has estado poniendo atención). En la práctica, significa que mientras que Python 3 puede leer datos almacenados con el protocolo versión 2, Python 2 no puede leer datos almacenados con el protocolo versión 3.

13.6. Depuración de ficheros "pickle"

¿A qué se parece el protocolo "pickle"? Vamos a salir un momento de la consola de Python y echarle un vistazo al fichero entry.pickle que hemos creado.

```
you@localhost:~/diveintopython3/examples$ ls -l entry.pickle
-rw-r--r-- 1 you  you  358 Aug  3 13:34 entry.pickle
you@localhost:~/diveintopython3/examples$ cat entry.pickle
comments_linkqNXtagsqXdiveintopythonqXdocbookqXhtmlq?qX publishedq?
XlinkXJhttp://diveintomark.org/archives/2009/03/27/dive-into-history-
2009-edition
q    Xpublished_dateq
ctime
struct_time
?qRqXtitleqXDive into history, 2009 editionqu.
```

No ha sido muy útil. Puedes ver las cadenas de texto, pero los otros tipos de dato salen como caracteres ilegibles. Los campos no están delimitados por tabuladores ni espacios. No se trata de un formato que quieras depurar por ti mismo.

```
>>> shell
1
>>> import pickletools
>>> with open('entry.pickle', 'rb') as f:
...     pickletools.dis(f)
    0: \x80 PROTO      3
    2: }    EMPTY_DICT
    3: q    BINPUT     0
    5: (    MARK
    6: X        BINUNICODE 'published_date'
   25: q        BINPUT     1
   27: c        GLOBAL     'time struct_time'
   45: q        BINPUT     2
   47: (        MARK
   48: M            BININT2    2009
   51: K            BININT1    3
   53: K            BININT1    27
   55: K            BININT1    22
   57: K            BININT1    20
   59: K            BININT1    42
   61: K            BININT1    4
   63: K            BININT1    86
   65: J            BININT     -1
   70: t            TUPLE      (MARK at 47)
   71: q        BINPUT     3
   73: }        EMPTY_DICT
   74: q        BINPUT     4
   76: \x86     TUPLE2
   77: q        BINPUT     5
   79: R        REDUCE
   80: q        BINPUT     6
   82: X        BINUNICODE 'comments_link'
  100: q        BINPUT     7
  102: N        NONE
  103: X        BINUNICODE 'internal_id'
  119: q        BINPUT     8
  121: C        SHORT_BINBYTES 'xxxx'
  127: q        BINPUT     9
  129: X        BINUNICODE 'tags'
  138: q        BINPUT     10
  140: X        BINUNICODE 'diveintopython'
  159: q        BINPUT     11
  161: X        BINUNICODE 'docbook'
  173: q        BINPUT     12
  175: X        BINUNICODE 'html'
  184: q        BINPUT     13
  186: \x87     TUPLE3
  187: q        BINPUT     14
  189: X        BINUNICODE 'title'
  199: q        BINPUT     15
  201: X        BINUNICODE 'Dive into history, 2009 edition'
  237: q        BINPUT     16
  239: X        BINUNICODE 'article_link'
```

```
  256: q          BINPUT     17
  258: X          BINUNICODE 'http://diveintomark.org/archives/2009/
                              03/27/dive-into-history-2009-edition'
  337: q          BINPUT     18
  339: X          BINUNICODE 'published'
  353: q          BINPUT     19
  355: \x88       NEWTRUE
  356: u          SETITEMS   (MARK at 5)
  357: .      STOP
highest protocol among opcodes = 3
```

La información más interesante de este volcado es la que aparece en la última línea, ya que muestra la versión del protocolo de "pickle" con la que el fichero se grabó. No existe un marcador de versión explício en el protocolo de "pickle". Para determinar la versión del protocolo, se observan los marcadores (códigos de operación - "opcodes") existentes en los datos almacenados y se utiliza el conocimiento expreso de qué códigos fueron introducidos en cada versión del protocolo "pickle". La función pickle.dis() hace exactamente eso e imprime el resultado en la última línea del volcado de salida.

La siguiente es una función que simplemente devuelve el número de versión sin imprimir nada:

```
import pickletools

def protocol_version(file_object):
    maxproto = -1
    for opcode, arg, pos in pickletools.genops(file_object):
        maxproto = max(maxproto, opcode.proto)
    return maxproto
```

Y aquí la vemos en acción:

```
>>> import pickleversion
>>> with open('entry.pickle', 'rb') as f:
...     v = pickleversion.protocol_version(f)
>>> v
3
```

13.7. Serialización de objetos Python para cargarlos en otros lenguajes

El formato utilizado por el módulo pickle es específico de Python. No intenta ser compatible con otros lenguajes de programación. Si la compatibilidad entre

lenguajes es un requisito, necesitas utilizar otros formatos de serialización. Uno de ellos es JSON[4]. "JSON" significa "JavaScript Object Notation - Notación de Objetos JavaScript", pero no dejes que el nombre te engañe —JSON está diseñado explícitamente para permitir su uso en diferentes lenguajes de programación.

Python 3 inclyye un módulo json en su librería estándar. Como el módulo pickle, el módulo json dispone de funciones para la serialización de estructuras de datos, almacenamiento de los datos serializados en disco, carga de los mismos y deserialización en un nuevo objeto Python. Pero también tiene importantes diferencias. La primera es que JSON es un formato de datos textual, no binario. La especificación RFC 4627[5] define el formato y cómo se codifican los diferentes tipos de datos de forma textual. Por ejemplo, un valor booleano se almacena como la cadena texto de cinco caracteres "false" o como la cadena de texto de cuatro caracters "true". Todos los valores de JSON tienen en cuenta las mayúsculas y minúsculas.

Segundo, como cualquier formato basado en texto, existe el problema de los espacios en blanco. JSON permite el uso de un número arbitrario de espacios en blanco (espacios, tabuladores, retornos de carro y saltos de línea) entre los valores. Estos espacios en blanco son "no significativos", lo que significa que los codificadores de JSON pueden añadir tantos como deseen, y los decodificadores de JSON están obligados a ignorarlos siempre que se encuentren entre dos valores. Esto permite que los datos de un fichero JSON se puedan imprimir bien formateados, anidando de forma clara los valores que se encuentran dentro de otros para que puedas verlos bien en un editor o visor de texto estándar. El módulo json de Python dispone de opciones para codificar la salida con formato apropiado para la lectura.

Tercero, existe el problema perenne de la codificación de caracteres. Puesto que JSON codifica los valores como texto plano, pero como ya sabes, no existe tal "texto plano". JSON debe almacenarse con caracteres Unicode (UTF-32, UTF-16 o, por defecto, UTF-8), y la sección 3 de la RFC-4627[6], define cómo indicar qué codificación se está utilizando.

13.8. Almacenamiento de datos en un fichero JSON

JSON se parece mucho a una estructura de datos que pudieras definir en JavaScript. No es casualidad, en realidad, puedes utilizar la función eval() de JavaScript para "decodificar" los datos serializados en JSON. Lo fundamental es conocer que

[4] http://json.org/
[5] http://www.ietf.org/rfc/rfc4627.txt
[6] http://www.ietf.org/rfc/rfc4627.txt

JSON forma parte del propio lenguaje JavaScript. Como tal, JSON puede que ya te sea familiar.

```
1 >>> shell
2 1
3 >>> basic_entry = {}
4 >>> basic_entry['id'] = 256
5 >>> basic_entry['title'] = 'Dive into history, 2009 edition'
6 >>> basic_entry['tags'] = ('diveintopython', 'docbook', 'html')
7 >>> basic_entry['published'] = True
8 >>> basic_entry['comments_link'] = None
9 >>> import json
10 >>> with open('basic.json', mode='w', encoding='utf-8') as f:
11 ...     json.dump(basic_entry, f)
```

1. *Línea 3:* Vamos a crear una nueva estructura de datos, en lugar de reutilizar la estructura de datos entry preexistente. Después veremos qué sucede cuando intentamos codificar en JSON la otra estructura de datos más compleja.

2. *Línea 10:* JSON está basado en texto, lo que significa que es necesario abrir el fichero en modo texto y especificar una codificación de caracteres. Nunca te equivocarás utilizando UTF-8.

3. *Línea 11:* Como con el módulo pickle, el módulo json define la función dump() que toma una estructura de datos Python y un objeto de flujo (stream) con permisos de escritura. La función dump() serializa la estructura de datos de Python y escribe el resultado en el objeto de flujo. Al hacerlo dentro de una sentencia with nos aseguramos de que el fichero quede cerrado correctamente cuando hayamos terminado.

¿Cómo queda el resultado serializado?

```
1 you@localhost:~/diveintopython3/examples$ cat basic.json
2 {"published": true, "tags": ["diveintopython", "docbook", "html"],
3 "comments_link": null, "id": 256,
4 "title": "Dive into history, 2009 edition"}
```

Es más legible que el fichero en formato de pickle. Pero como JSON puede contener tantos espacios en blanco como se desee entre diferentes valores, y el módulo json proporciona una forma sencilla de utilizar esta capacidad, podemos crear ficheros JSON aún más legibles.

```
1 >>> shell
2 1
3 >>> with open('basic-pretty.json', mode='w', encoding='utf-8') as f:
4 ...     json.dump(basic_entry, f, indent=2)
```

Si se pasa el parámetro indent a la función json.dump() el fichero JSON resultante será más legible aún. A costa de un fichero de tamaño mayor. El parámetro indent es un valor entero en el que 0 significa "pon cada valor en su propia línea" y un número mayor que cero significa "pon cada valor en su propia línea, y utiliza este número de espacios para indentar las estructuras de datos anidadas".

Por lo que éste es el resultado:

```
you@localhost:~/diveintopython3/examples$ cat basic-pretty.json
{
  "published": true,
  "tags": [
    "diveintopython",
    "docbook",
    "html"
  ],
  "comments_link": null,
  "id": 256,
  "title": "Dive into history, 2009 edition"
}
```

13.9. Mapeo de los tipos de datos de Python a JSON

Puesto que JSON no es específico de Python, existen algunas diferencias en su cobertura de los tipos de dato de Python. Algunas de ellas son simplemente de denominación, pero existen dos tipos de dato importantes de Python que no existen en JSON. Observa esta tabla a ver si los echas de menos:

Notas	JSON	Python 3
	object	dictionary
	array	list
	string	string
	integer	integer
	real number	float
*	true	True
*	false	False
*	null	None
* Las mayúsculas y minúsculas en los valores JSON son significativas.		

¿Te has dado cuenta de lo que falta? ¡Tuplas y bytes! JSON tiene un tipo de datos array, al que se mapean las listas de Python, pero no tiene un tipo de datos

separado para los "arrays congelados" (tuplas). Y aunque JSON soporta cadenas de texto, no tiene soporte para los objetos bytes o arrays de bytes.

13.10. Serialización de tipos no soportados en JSON

Incluso aunque JSON no tiene soporte intrínseco de bytes, es posible serializar objetos bytes. El módulo json proporciona unos puntos de extensibilidad para codificar y decodificar tipos de dato desconocidos (Por desconocido se entiende en este contexto a aquellos tipos de datos que no están definidos en la especificación de JSON). Si quieres codificar bytes u otros tipos de datos que JSON no soporte de forma nativa, necesitas proporcionar codificadores de decodificadores a medida para esos tipos de dato.

```
>>> shell
1
>>> entry
{'comments_link': None,
 'internal_id': b'\xDE\xD5\xB4\xF8',
 'title': 'Dive into history, 2009 edition',
 'tags': ('diveintopython', 'docbook', 'html'),
 'article_link': 'http://diveintomark.org/archives/2009/03
                  /27/dive-into-history-2009-edition',
 'published_date': time.struct_time(tm_year=2009, tm_mon=3,
                   tm_mday=27, tm_hour=22, tm_min=20, tm_sec=42,
                   tm_wday=4, tm_yday=86, tm_isdst=-1),
 'published': True}
>>> import json
>>> with open('entry.json', 'w', encoding='utf-8') as f:
...     json.dump(entry, f)
...
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
  File "C:\Python31\lib\json\__init__.py", line 178, in dump
    for chunk in iterable:
  File "C:\Python31\lib\json\encoder.py", line 408, in _iterencode
    for chunk in _iterencode_dict(o, _current_indent_level):
  File "C:\Python31\lib\json\encoder.py", line 382, in _iterencode_dict
    for chunk in chunks:
  File "C:\Python31\lib\json\encoder.py", line 416, in _iterencode
    o = _default(o)
  File "C:\Python31\lib\json\encoder.py", line 170, in default
    raise TypeError(repr(o) + " is not JSON serializable")
TypeError: b'\xDE\xD5\xB4\xF8' is not JSON serializable
```

1. *Línea 3:* Ok, es el momento de volver a la estructura de datos entry. Tiene de

todo: un valor booleano, un None, una cadena de texto, una tupla de cadenas de texto, un objeto bytes y una estructura time.

2. *Línea 15:* Sé lo que he dicho antes, pero vamos a repetirlo: JSON es un formato de texto. Siempre se deben abrir los ficheros JSON en modo texto con la codificación de caracteres UTF-8.

3. *Línea 18:* ¡Error! ¿Qué ha pasado?

Lo que ha pasado es que la función json.dump() intentó serializar el objeto bytes pero falló porque JSON no dispone de soporte de objetos bytes. Sin embargo, si es importante almacenar bytes en este formato, puedes definir tu propio "formato de serialización".

```
1 def to_json(python_object):
2     if isinstance(python_object, bytes):
3         return {'__class__': 'bytes',
4                 '__value__': list(python_object)}
5     raise TypeError(repr(python_object) + ' is not JSON serializable')
```

1. *Línea 1:* Para definir un formato de serialización para un tipo de datos que JSON no soporte de forma nativa, simplemente define una función que tome un objeto Python como parámetro. Este objeto será el que la función json.dump() sea incapaz de serializar de forma nativa —en este caso el objeto bytes.

2. *Línea 2:* La función debe validar el tipo de datos que recibe. No es estrictamente necesario pero así queda totalmente claro que casos cubre esta función, y hace más sencillo ampliarla más tarde.

3. *Línea 4:* En este caso, he elegido convertir el objeto bytes en un diccionario. La clave __class__ guardará el tipo de datos original (como una cadena, "bytes"), y la clave __value__ guardará el valor real. Como hay que convertirlo a algo que pueda serializarse en JSON, no se puede guardar directamente el objeto bytes. Como un objeto bytes es una secuencia de números entereos; con cada entero entre el 0 y el 255, podemos utilizar la función list() para convertir el objeto bytes en una lista de enteros. De forma que el objeto b'\xDE\xD5\x84\xF8' se convierte en [222, 213, 180, 248]. Por ejemplo, el byte \xDE en hexadecimal, se convierte en 222 en decimal, \xD5 es 213 y así cada uno de ellos.

4. *Línea 5:* Esta línea es importante. La estructura de datos que estás serializando puede contener tipos de dato que ni el serializador interno del módulo de Python ni el tuyo puedan manejar. En este caso, tu serializador debe elevar una excepción TypeError para que la función json.dump() sepa que tu serializador no reconoció el tipo de dato del objeto.

Y eso es todo, no necesitas hacer nada más. En particular, esta función a medida retorna un *un diccionario de Python*, no una cadena. No estás haciendo la serialización a JSON completa por ti mismo; solamente la parte correspondiente a un tipo de datos que no está soportado de forma nativa. La función json.dump() hará el resto.

```
>>> shell
1
>>> import customserializer
>>> with open('entry.json', 'w', encoding='utf-8') as f:
...     json.dump(entry, f, default=customserializer.to_json)
...
Traceback (most recent call last):
  File "<stdin>", line 9, in <module>
    json.dump(entry, f, default=customserializer.to_json)
  File "C:\Python31\lib\json\__init__.py", line 178, in dump
    for chunk in iterable:
  File "C:\Python31\lib\json\encoder.py", line 408, in _iterencode
    for chunk in _iterencode_dict(o, _current_indent_level):
  File "C:\Python31\lib\json\encoder.py", line 382, in _iterencode_dict
    for chunk in chunks:
  File "C:\Python31\lib\json\encoder.py", line 416, in _iterencode
    o = _default(o)
  File "/Users/pilgrim/diveintopython3/examples/customserializer.py",
  line 12, in to_json
    raise TypeError(repr(python_object) + ' is not JSON serializable')
TypeError: time.struct_time(tm_year=2009, tm_mon=3, tm_mday=27, tm_hour=22,
 tm_min=20, tm_sec=42, tm_wday=4, tm_yday=86, tm_isdst=-1)
 is not JSON serializable
```

1. *Línea 3:* El módulo customserializer es el lugar en el que has definido la función to_json() del ejemplo anterior.

2. *Línea 4:* El modo texto, la codificación UTF-8, etc. ¡Lo olvidarás! ¡a veces lo olvidarás! Y todo funcionará hasta el momento en que falle, y cuando falle, lo hará de forma espectacular.

3. *Línea 5:* Este es el trozo importante: asignar una función de conversión ad-hoc en la función json.dump(), hay que pasar tu función a la función json.dump() en el parámetro default.

4. *Línea 20:* Ok, realmente no ha funcionado. Pero observa la excepción. La función json.dump() ya no se queja más sobre el objeto de tipo bytes. Ahora se está quejando sobre un objeto totalmente diferente, el objeto time.struct_time.

Aunque obtener una excepción diferente podría no parecer mucho progreso ¡lo es! Haremos una modificación más para superar este error:

```
1  import time
2
3  def to_json(python_object):
4      if isinstance(python_object, time.struct_time):
5          return {'__class__': 'time.asctime',
6                  '__value__': time.asctime(python_object)}
7      if isinstance(python_object, bytes):
8          return {'__class__': 'bytes',
9                  '__value__': list(python_object)}
10     raise TypeError(repr(python_object) + ' is not JSON serializable')
```

1. *Línea 4:* Añadimos código a nuestra función customserializer.to_json(), necesitamos validar que el objeto Python sea time.struct_time (aquél con el que la función json.dump() está teniendo problemas).

2. *Línea 6:* Haremos una conversión parecida a la que hicimos con el objeto bytes: convertir el objeto time.struct_time en un diccionario que solamente contenga valores serializables en JSON. En este caso, la forma más sencilla de convertir una fecha/hora a JSON es convertirlo en una cadena con la función time.asctime(). La función time.asctime() convertirá la estructura en la cadena 'Fri Mar 27 22:20:42 2009'.

Con estas dos conversiones a medida, la estructura completa de datos entry debería serializarse a JSON sin más problemas.

```
1  >>> shell
2  1
3  >>> with open('entry.json', 'w', encoding='utf-8') as f:
4  ...     json.dump(entry, f, default=customserializer.to_json)
5  ...
```

```
1  you@localhost:~/diveintopython3/examples$ ls -l example.json
2  -rw-r--r-- 1 you  you  391 Aug  3 13:34 entry.json
3  you@localhost:~/diveintopython3/examples$ cat example.json
4  {"published_date": {"__class__": "time.asctime",
5  "__value__": "Fri Mar 27 22:20:42 2009"},
6  "comments_link": null, "internal_id": {"__class__": "bytes",
7  "__value__": [222, 213, 180, 248]},
8  "tags": ["diveintopython", "docbook", "html"],
9  "title": "Dive into history, 2009 edition",
10 "article_link": "http://diveintomark.org/archives/
11 2009/03/27/dive-into-history-2009-edition",
12 "published": true}
```

13.11. Carga de datos desde un fichero JSON

Como el módulo pickle, el módulo json tiene una función load() que toma un objeto de flujo de datos y lee la información formateada en JSON y crea un objeto Python que es idéntico a la estructura de datos JSON.

```
1 >>> shell
2 2
3 >>> del entry
4 >>> entry
5 Traceback (most recent call last):
6   File "<stdin>", line 1, in <module>
7 NameError: name 'entry' is not defined
8 >>> import json
9 >>> with open('entry.json', 'r', encoding='utf-8') as f:
10 ...     entry = json.load(f)
11 ...
12 >>> entry
13 {'comments_link': None,
14  'internal_id': {'__class__': 'bytes',
15  '__value__': [222, 213, 180, 248]},
16  'title': 'Dive into history, 2009 edition',
17  'tags': ['diveintopython', 'docbook', 'html'],
18  'article_link': 'http://diveintomark.org/archives/
19 2009/03/27/dive-into-history-2009-edition',
20  'published_date': {'__class__': 'time.asctime',
21 '__value__': 'Fri Mar 27 22:20:42 2009'},
22  'published': True}
```

1. *Línea 3:* Con fines demostrativos, pasamos a la consola #2 y borramos la estructura de datos entry que habíamos creado antes con el módulo pickle.

2. *Línea 10:* En el caso más simple, la función json.load() funciona de la misma forma que la función pickle.load(). Le pasamos un flujo de datos y devuelve un objeto Python nuevo.

3. *Línea 12:* Tengo buenas y malas noticias. Las buenas primero: la función json.load() carga satisfactoriamente el ficheroentry.json que has creado en la consola #1 y crea un nuevo objeto Python que contiene la información. Ahora las malas noticias: No recrea la estructura de datos entry original. Los dos valores 'internal_id' y 'published_date' se han recreado como diccionarios — específicamente, los diccionarios con valores compatibles JSON que creamos en la función de conversión to_json().

La función json.load() no sabe nada sobre ninguna función de conversión que puedas haber pasado a la función json.dump(). Lo que se necesita es la función

opuesta a to_json() —una función que tomará un objeto JSON convertido a medida y convertirá de nuevo a Python el tipo de datos original.

```
1 # add this to customserializer.py
2 def from_json(json_object):
3     if '__class__' in json_object:
4         if json_object['__class__'] == 'time.asctime':
5             return time.strptime(json_object['__value__'])
6         if json_object['__class__'] == 'bytes':
7             return bytes(json_object['__value__'])
8     return json_object
```

1. *Línea 2:* Esta función de conversión también toma un parámetro y devuelve un valor. Pero el parámetro que toma no es una cadena, es un objeto Python —el resultado de deserializar la cadena JSON en un objeto Python.

2. *Línea 3:* Lo único que hay que hacer es validar si el objeto contiene la clave '__class__' que creó la función to_json(). Si es así, el valor de la clave '__class__' te dirá cómo decodificar el valor en su tipo de datos original de Python.

3. *Línea 5:* Para decodificar la cadena de texto que que devolvió la función time.asctime(), utilizamos la función time.strptime(). Esta función toma una cadena de texto con formato de fecha y hora (en un formato que se puede adaptar, pero que tiene el formato por defecto de la función time.asctime()) y devuelve un objeto time.struct_time.

4. *Línea 7:* Para convertir de nuevo la lista de enteros a un objeto bytes puedes utilizar la función bytes().

Eso es todo. Solamente se manejaban dos tipos de dato en la función to_json(), y ahora son esos dos tipos de dato los que se manejan en la función from_json(). Este es el resultado:

```
>>> shell
2
>>> import customserializer
>>> with open('entry.json', 'r', encoding='utf-8') as f:
...     entry = json.load(f, object_hook=customserializer.from_json)
...
>>> entry
{'comments_link': None,
 'internal_id': b'\xDE\xD5\xB4\xF8',
 'title': 'Dive into history, 2009 edition',
 'tags': ['diveintopython', 'docbook', 'html'],
 'article_link': 'http://diveintomark.org/archives/
2009/03/27/dive-into-history-2009-edition',
 'published_date': time.struct_time(tm_year=2009, tm_mon=3,
tm_mday=27, tm_hour=22, tm_min=20, tm_sec=42, tm_wday=4,
tm_yday=86, tm_isdst=-1),
 'published': True}
```

1. *Línea 5:* Para utilizar la función from_json() durante el proceso de deserialización, hay que pasarla en el parámetro object_hook a la función json.load(). Una función que toma como parámetro a otra función ¡es muy útil!

2. *Línea 7:* La estructura de datos entry ahora contiene uan clave 'internal_id' que tiene como valor a un objeto bytes. Y también contiene una clave 'published_date' cuyo valor es un objeto time.struct_time.

Sin embargo, aún queda un pequeño tema por tratar.

```
>>> shell
1
>>> import customserializer
>>> with open('entry.json', 'r', encoding='utf-8') as f:
...     entry2 = json.load(f, object_hook=customserializer.from_json)
...
>>> entry2 == entry
False
>>> entry['tags']
('diveintopython', 'docbook', 'html')
>>> entry2['tags']
['diveintopython', 'docbook', 'html']
```

1. *Línea 7:* Incluso después de utilizar la función to_json() en la serialización y la función from_json() en la deserialización, aún no hemos recreado la réplica perfecta de la estructura original ¿porqué no?

2. *Línea 9:* En la estructura de datos original el valor de la clave 'tags' era una tupla de tres cadenas.

3. *Línea 11:* Pero en la estructura entry2 el valor de la clave 'tags' es una *lista* de tres cadenas. JSON no distingue entre tuplas y listas; solamente tiene un tipo de datos parecido a la lista, el array, y el módulo json de Python convierte calladamente ambos tipos, listas y tuplas, en arrays de JSON durante la serialización. Para la mayoría de usos, es posible ignorar esta diferencia, pero conviene saberlo cuando se utiliza este módulo json.

13.12. Lecturas recomendadas

Muchos artículos del módulo pickle hacen referencia a cPickle. En Python 2 existen dos implementaciones del módulo pickle, uno escrito en Python puro y otro escrito en C (pero que se puede llamar desde Python). En Python 3 se han consolidado ambos módulos, por lo que siempre deberías utilizar import pickle.

Sobre el módulo pickle:

- el módulo pickle:
  http://docs.python.org/3.1/library/pickle.html
- pickle y cPickle —serialización de objetos en Python:
  http://www.doughellmann.com/PyMOTW/pickle/
- Utilización de pickle:
  http://wiki.python.org/moin/UsingPickle
- Gestión de la persistencia en Python:
  http://www.ibm.com/developerworks/library/l-pypers.html

Sobre el módulo json:

- json — Serializador de la notación de objetos de JavaScript:
  http://www.doughellmann.com/PyMOTW/json/
- Codificación y decodificación JSON en Python utilizando objetos a medida:
  http://blog.quaternio.net/2009/07/16/json-encoding-and-decoding-with-custom-objects-in-python/

Sobre la extensibilidad de pickle:

- Sobre el almacenamiento de instancias de clase:
  http://docs.python.org/3.1/library/pickle.html#pickling-class-instances
- Persistencia de objetos externos:
  http://docs.python.org/3.1/library/pickle.html#persistence-of-external-objects
- Manejo de objetos con estado:
  http://docs.python.org/3.1/library/pickle.html#handling-stateful-objects

Capítulo 14

Servicios Web HTTP

Nivel de dificultad:◆ ◆ ◆ ◆ ◇

"Una mente revuelta hace la almohada incómoda."
—Charlotte Brontë

14.1. Inmersión

Los servicios web HTTP permiten el envío y recepción de información desde servidores remotos, sin utilizar nada más que operaciones HTTP. Si quieres recuperar información desde un servidor, utiliza HTTP GET; si quieres enviar información al servidor, utiliza HTTP POST. Los servicios web HTTP más avanzados disponen de APIs avanzadas que permiten crear, modifiar y borrar información, utilizando HTTP PUT y HTTP DELETE. En otras palabras, los "verbos" construidos en el protocolo HTTP (GET, POST, PUT y DELETE) pueden mapearse directamente en operaciones de nivel de aplicación para recuperar, crear, modificar y borrar datos.

La principal ventaja de esta aproximación es la simplicidad, y esta simplicidad se ha demostrado que es muy popular. La información —normalmente en XML o JSON— puede estar construida y almacenada estáticamente, o generada dinámicamente por un programa del servidor y todos los lenguajes de programación importantes (incluido Python ¡desde luego!) incluyen una librería HTTP para descargarla. La depuración también es más sencilla, porque cada recurso de un servicio web HTTP dispone de una dirección única (en forma de URL), puedes cargarla en tu navegador web e inmediatamente ver la información obtenida.

Son ejemplos de servicios web HTTP:

1. Las APIs de datos de Google[1] te permiten interactuar con una amplia variedad de servicios de Google, incluidos Blogger[2] y YouTube[3].

2. Los servicios de Flickr[4] te permiten cargar y descargar fotos de Flickr.

3. La API de Twitter[5] te permite publicar actualizaciones de estado en Twitter.

4. ...y muchos más[6].

Python 3 dispone de dos librerías diferentes para interactuar con los servicios web HTTP:

1. http.client[7] es una librería de bajo nivel que implementa el protocolo HTTP[8].

2. urllib.request[9] es una capa de abstracción construida sobre http.client. Proporciona una API estándar para acceder a servidores HTTP y FTP, sigue automáticamente redirecciones HTTP y maneja algunas formas comunes de autenticación HTTP.

¿Cuál deberiamos usar? Ninguna de ellas. En su lugar deberías utilizar httplib2[10], una librería de código abierto de terceros que implementa HTTP de forma más completa que http.client y proporciona una mejor abstracción que urllib.request.

Para comprender porqué httplib2 es la elección correcta necesitas comprender primero HTTP.

14.2. Características de HTTP

Hay cinco características importantes que todos los clientes HTTP deberían soportar.

---

[1] http://code.google.com/apis/gdata/
[2] http://www.blogger.com/
[3] http://www.youtube.com/
[4] http://www.flickr.com/services/api/
[5] http://apiwiki.twitter.com/
[6] http://www.programmableweb.com/apis/directory/1?sort=mashups
[7] http://docs.python.org/3.1/library/http.client.html
[8] RFC 2616:http://www.w3.org/Protocols/rfc2616/rfc2616.html
[9] http://docs.python.org/3.1/library/urllib.request.html
[10] http://code.google.com/p/httplib2/

14.2.1. Caché

Lo más importante que se debe comprender para usar cualquier servicio web es que el acceso a través de la web es increiblemente costoso. No quiero decir que cueste en "euros y céntimos" (aunque el ancho de banda no sea gratis). Quiero decir que lleva mucho tiempo abrir una conexión, enviar una petición y recuperar una respuesta de un servidor remoto. Incluso en la conexión más rápida existente, la *latencia* (el tiempo que tarda desde el envío de una petición hasta que se inicia la recogida de los datos en la respuesta) puede ser mayor de lo que puedas esperar. Un router puede tener un malfuncionamiento, un paquete se pierde, un proxy está bajo ataque —nunca existe un momento de aburrimiento en la red de Internet y no puedes hacer nada contra esto.

HTTP está diseñado con la posibilidad de cacheo en mente. Existe toda una clase de dispositivos (llamados "proxys caché") cuyo único trabajo consiste en interponerse entre ti y el resto del mundo para minimizar el acceso a la red. Tu empresa o tu proveedor de servicios de Internet es muy probable que tengan proxys de este tipo, incluso aunque no seas consciente de ello. Funcionan gracias al sistema de caché construido en el protocolo HTTP.

Cache-Control: max-age significa "no me moleste hasta dentro de una semana"

Veamos un ejemplo concreto sobre cómo funciona la caché. Visita diveintomark.org en tu navegador. Esta página incluye una imagen de fondo wearehugh.com/m.jpg. Cuando tu navegador descarga esa imagen, el servidor incluye las siguientes cabeceras HTTP:

```
1  HTTP/1.1 200 OK
2  Date: Sun, 31 May 2009 17:14:04 GMT
3  Server: Apache
4  Last-Modified: Fri, 22 Aug 2008 04:28:16 GMT
5  ETag: "3075-ddc8d800"
6  Accept-Ranges: bytes
7  Content-Length: 12405
8  Cache-Control: max-age=31536000, public
9  Expires: Mon, 31 May 2010 17:14:04 GMT
10 Connection: close
11 Content-Type: image/jpeg
```

Las cabeceras Cache-Control y Expires le dicen a tu navegador (y a cualquier proxy con caché entre tú y el servidor) que esta imagen puede cachearse durante un año. *¡un año!* y si, durante el próximo año, visitas alguna página que incluya un enlace a esta imagen, tu navegador cargará la imagen desde su caché *sin generar ninguna actividad en la red.*

Pero espera, que es aún mejor. Digamos que tu navegador borra la imagen de su caché local por alguna razón. Puede que se le acabe el espacio que tiene reservado en el disco, puede que tú borres manualmente la caché. Lo que sea. Pero las cabeceras HTTP dicen que esta información se puede almacenar en cualquier caché pública[11]. Los proxies con caché están diseñados para disponer de "toneladas" de espacio de almacenamiento, probablemete mucho más del que dispone tu navegador.

Si tu compañía o proveedor de Interne disponen de un proxy con caché, el proxy puede que tenga todavía la información disponible en la caché. Cuando visites de nuevo diveintomark.org tu navegador buscará la imagen en la caché local, si no la encuentra hará una petición a la red para intentar descargarla del servidor remoto. Pero si el proxy aún dispone de una copia de la imagen, interceptará la petición y servirá la imagen desde la caché. Esto significa que tu petición nunca alcanzará el servidor remoto. De hecho, nunca saldrá de la red de tu compañía. Esto hace que la descarga sea más rápida (menos saltos en la red) y ahorra dinero a tu compañía (menos información descargándose del mundo exterior).

El sistema de caché de HTTP únicamente funciona cuando todas las partes hacen su trabajo. Por una parte, los servidores deben enviar las cabeceras correctas en su respuesta. Por otra parte, los clientes deben comprender y respetar las cabeceras antes de solicitar la misma información dos veces. Los proxys que se encuentren en medio del camino no son la panacea; dependen de los servidores y de los clientes.

Las librerías de Python de HTTP no soportan la caché, pero la librería httplib2 sí.

14.2.2. Comprobación de la última vez que se modificó una página

Alguna información nunca cambia, mientras que otra cambia constantemente. Entre ambos extremos existe un amplio campo de datos que *podría* haber cambiado, pero no lo ha hecho. El flujo de información de la CNN se actualiza cada pocos minutos, pero mi blog puede que no cambie en días o semanas. En el último caso, no quiero decirle a los clientes que cacheen las páginas durante semanas, porque cuando realmente pongo una nueva entrada, la gente no la leería hasta pasadas unas semanas (porque estarían respetando mis cabeceras de caché que dirían "no te preocupes de validar durante semanas"). Por otra parte, no quiero que los clientes se estén descargando mi flujo completo una vez cada hora si no ha cambiado.

[11] Técnicamente lo importante es que la cabecera Cache-Control no tiene la clave private, por lo que esta información se puede almacenar en una caché por defecto.

304: Not Modified significa "la misma mierda en distinto día".

El protocolo HTTP tiene una solución para esto. Cuando solicitas datos por primera vez, el servidor puede enviar una cabecera denominada Last-Modified. Esta cabecera indica lo que dice: la fecha en la que la información fue modificada. La imagen de fondo de diveintomark.org incluía una cabecera Last-Modified

```
1  HTTP/1.1 200 OK
2  Date: Sun, 31 May 2009 17:14:04 GMT
3  Server: Apache
4  Last-Modified: Fri, 22 Aug 2008 04:28:16 GMT
5  ETag: "3075-ddc8d800"
6  Accept-Ranges: bytes
7  Content-Length: 12405
8  Cache-Control: max-age=31536000, public
9  Expires: Mon, 31 May 2010 17:14:04 GMT
10 Connection: close
11 Content-Type: image/jpeg
```

Cuando solicitas la misma información una segunda vez (o tercera o cuarta), puedes enviar una cabecera If-Modified-Since con la petición, con la fecha que recuperaste desde el servidor la última vez. Si la información ha cambiado desde entonces, el servidor ignora la cabecera If-Modified-Since y devuelve la nueva información con un código de estado 200. Pero si los datos no han cambiado desde entonces, el servidor envía un código de estado especial HTTP 304 que significa "estos datos no han cambiado desde la última vez que los pediste". Puedes probar esto en la línea de comando utilizando la sentencia curl[12]:

```
1  you@localhost:~$ curl -I -H "If-Modified-Since:
2  Fri, 22 Aug 2008 04:28:16 GMT" http://wearehugh.com/m.jpg
3  HTTP/1.1 304 Not Modified
4  Date: Sun, 31 May 2009 18:04:39 GMT
5  Server: Apache
6  Connection: close
7  ETag: "3075-ddc8d800"
8  Expires: Mon, 31 May 2010 18:04:39 GMT
9  Cache-Control: max-age=31536000, public
```

¿Porqué se trata de una mejora? Porque cuando el servidor envía un código 304, *no reenvía la información*. Lo único que se obtiene es el código de estado. Incluso después de que tu copia de caché haya expirado, la comprobación de la última fecha de modificación te asegura que no descargas la misma información dos veces si no ha cambiado (Como bono extra, esta respuesta 304 también incluye las cabeceras de caché. Los proxys mantendrán una copia de los datos incluso después

[12] http://curl.haxx.se/

de que hayan expirado "oficialmente", con la esperanza de que los datos no hayan cambiado *realmente* y que la siguiente petición responda con un código de estado 304 y la información de caché actualizada).

Las librerías HTTP de Python no soportan la comprobación de la última fecha de modificación, pero la librería httplib2 sí lo hace.

14.2.3. Caché de ETag

Las ETags son una forma alternativa de conseguir lo mismo que con la validación de la última fecha de modificación. En este caso, el servidor envía un código hash en una cabecera ETag junto con los datos que hayas solicitado (La forma exacta por la que el servidor calcula este hash la determina el propio servidor. El único requisito es que cambie cuando cambie la información). La imagen de fondo referenciada desde diveintomark.org tenía un código ETag.

```
1  HTTP/1.1 200 OK
2  Date: Sun, 31 May 2009 17:14:04 GMT
3  Server: Apache
4  Last-Modified: Fri, 22 Aug 2008 04:28:16 GMT
5  ETag: "3075-ddc8d800"
6  Accept-Ranges: bytes
7  Content-Length: 12405
8  Cache-Control: max-age=31536000, public
9  Expires: Mon, 31 May 2010 17:14:04 GMT
10 Connection: close
11 Content-Type: image/jpeg
```

La segunda vez que solicites la misma información, incluirás el código ETag en una cabecera If-None-Match. Si la información no ha cambiado, el servidor enviará un código de estado 304. Como en el caso de la comprobación de la fecha de última modificación, el servidor únicamente envía el código de estado 304; no envía la misma información una segunda vez. Al incluir un código Etag en tu segunda petición, le estás diciendo al servidor que no existe necesidad de volver a enviar la misma información si aún coincide con este hash, puesto que aún tienes la información desde la última vez.

De nuevo con curl:

> Etag *significa* "no hay nada nuevo bajo el sol"

```
1 you@localhost:~$ curl -I -H "If-None-Match: \"3075-ddc8d800\""
2 http://wearehugh.com/m.jpg
3 HTTP/1.1 304 Not Modified
4 Date: Sun, 31 May 2009 18:04:39 GMT
5 Server: Apache
6 Connection: close
7 ETag: "3075-ddc8d800"
8 Expires: Mon, 31 May 2010 18:04:39 GMT
9 Cache-Control: max-age=31536000, public
```

Las ETag se suelen encerrar entre comillas, pero las comillas forman parte del valor. Esto significa que necesitas enviar al servidor esas comillas en la cabecera If-None-Match.

Las librerías de Python HTTP no soportan ETags, pero httplib2 sí.

14.2.4. Compresión

Cuando hablamos de los servicios web HTTP, siempre se suele hablar de información de texto que va y viene a través de la red. Puede que sea XML, puede que sea JSON o únicamente texto plano. Independientemente del formato, el texto se comprime bastante bien. En el flujo de ejemplo del capítulo sobre XML la longitud del texto sin comprimir es de 3070 bytes, pero serían 941 bytes después de aplicar una compresión gzip. ¡El 30 % del tamaño original!

HTTP soporta varios algoritmos de compresión[13]. Los dos más comunes son gzip y deflate. Cuando solicitas un recurso sobre HTTP puedes pedirle al servidor que lo envíe en formato comprimido. Puedes incluir una cabecera Accept-encoding en tu petición que liste qué algoritmos de compresión soportas. Si el servidor soporta alguno de los algoritmos te enviará la información comprimida (con una cabecera Content-encoding que indica el algoritmo que se ha utilizado). Ya solamente te quedará descomprimir los datos.

> Una pista importante para los desarrolladores del lado del servidor: debe asegurarse que la versión comprimida de un recurso tiene diferente ETag que la versión descomprimida. De otro modo, los proxys de caché se confundirán y pueden servir la versión comprimida a clientes que no pueden manejarla. Lee la discusión de un error de Apache (número 39727[14]) para más detalles sobre este sutil asunto.

Las librerías HTTP de Python no soportan compresión, httplib2 sí.

[13] http://www.iana.org/assignments/http-parameters
[14] https://issues.apache.org/bugzilla/show_bug.cgi?id=39727

14.2.5. Redireccionamiento

Las URIs buenas no cambian, pero muchas no lo son. Los sitios web se reorganizan, las páginas se mueven a nuevas direcciones, incluso los servicios web se pueden reorganizar. Un flujo de información sindicada en http://example.com/index.xml podría moverse a http://example.com/xml/atom.xml. O el dominio completo podría moverse, según una organización pueda expandirse y reorganizarse http://example.com/index.xml se podría convertir en http://server-farm-1.example.com/index.xm.

Cada vez que solicitas alguna clase de recurso de un servidor HTTP, el servidor incluye un código de estado en su respuesta. El código de estado 200 significa "todo es normal, aquí está la página solicitada". El código de estado 404 significa "página no encontrada" (probabblemente te ha pasado alguna vez mientras navegabas en Internet). Los códigos de estado de la gama de los 300 indican algún tipo de redireccionamiento.

HTTP dispone de varias formas de indicar que un recurso se ha movido. Las dos técnicas más habituales son los códigos de estado 302 y 301. El código de estado 302 es una *redirección temporal*; lo que significa "¡uh! se ha movido temporalmente a otra dirección" (y luego se indica la dirección temporal en la cabecera Location). El código de estado 301 es una *redirección permanente*; significa "¡uh! se ha movido permanentemente" (y luego indica la nueva dirección en una cabecera Location). Si el código de estado es 302 y una nueva dirección, la especificación HTTP indica que deberías utilizar la nueva dirección para obtener lo que has solicitdo, pero la siguiente vez que quieras acceder al mismo recurso, deberías reintentarlo con la vieja dirección. Pero si lo que obtienes es un codigo de estado 301 y una nueva dirección, se supone que debes usar la nueva dirección a partir de ese momento.

> Location significa "mira aquí".

El módulo urllib.request sigue automáticamente los redireccionamientos cuando recibe los códigos de estado indicados desde el servidor HTTP, pero no te indica que lo haya hecho. Obtendrás los datos que solicitaste, pero no sabrás nunca que la librería te ayudó siguiendo el redireccionamiento necesario. Siempre seguirás usando la vieja dirección, y cada vez que suceda, serás redirigido a la nueva dirección mediante la ayuda que te presta el módulo urllib.request. En otras palabras, trata las redirecciones permanentes de la misma forma que las temporales. Esto significa que hacen falta dos "vueltas" en lugar de una para cada acceso, lo que es malo para el servidor y para ti.

httplib2 maneja las redirecciones permanentes por ti. No solamente te dirá que ha sucedido una redirección permanente, mantendrá un registro local de estas redirecciones y reescribirá las URL afectadas antes de solicitarlas.

14.3. Cómo no se debe recuperar información a través de HTTP

Digamos que quieres descargar un recurso a través de HTTP, por ejemplo, un flujo de datos Atom. Como se trata de un flujo, no lo vas a descargar una única vez; vas a descargarlo una y otra vez (La mayoría de los lectores de noticias comprueban si ha habido cambios una vez cada hora). Vamos a hacerlo de la forma más manual posible, en primer lugar, y luego veremos cómo hacerlo mejor.

```
1  >>> import urllib.request
2  >>> a_url = 'http://diveintopython3.org/examples/feed.xml'
3  >>> data = urllib.request.urlopen(a_url).read()
4  >>> type(data)
5  <class 'bytes'>
6  >>> print(data)
7  <?xml version='1.0' encoding='utf-8'?>
8  <feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'>
9    <title>dive into mark</title>
10   <subtitle>currently between addictions</subtitle>
11   <id>tag:diveintomark.org,2001-07-29:/</id>
12   <updated>2009-03-27T21:56:07Z</updated>
13   <link rel='alternate' type='text/html' href='http://diveintomark.org/'/>
14   ...
```

1. *Línea 3:* La descarga de cualquier información a través de HTTP es increíblemente sencilla en Python; de hecho se trata de una única línea. El módulo urllib.request dispone de una útil función urlopen() que toma la dirección de la página que quieres y retorna un objeto de flujo (como un fichero) que puedes leer con la función read() para recuperar el contenido completo de la página. No puede ser más sencillo.

2. *Línea 4:* El método urlopen().read() siempre devuelve un objeto bytes, no una cadena de texto. Recuerda que los bytes son bytes y los caracteres no son más que una abstracción. Los servidores HTTP no se preocupan de la abstracción. Si solicitas un recurso, obtienes bytes. Si quieres una cadena de texto, necesitarás determinar la codificación de caracteres utilizada para poder convertir los bytes en una cadena de texto.

¿Qué tiene de mala esta forma de recuperar el recurso? Para una prueba rápida durante el desarrollo no hay nada de malo. Yo mismo lo hago todo el rato. Quería el contenido de un flujo de noticias, y tengo el contenido de dicho flujo. La misma técnica funciona con cualquier página web. Pero una vez comienzas a pensar en términos de un servicio web al que quieres acceder frecuentemente (por ejemplo:

solicitar esta información una vez cada hora), entonces estás siendo ineficiente y basto.

14.4. ¿Qué sucede por debajo?

Para ver porqué esta manera de hacer la descarga es ineficiente y basta, vamos a activar las características de depuración de la librería de HTTP de Python para ver qué se está enviando a través de la red.

```
1 >>> from http.client import HTTPConnection
2 >>> HTTPConnection.debuglevel = 1
3 >>> from urllib.request import urlopen
4 >>> response = urlopen('http://diveintopython3.org/examples/feed.xml')
5 send: b'GET /examples/feed.xml HTTP/1.1
6 Host: diveintopython3.org
7 Accept-Encoding: identity
8 User-Agent: Python-urllib/3.1'
9 Connection: close
10 reply: 'HTTP/1.1 200 OK'
11 ...se omite el resto de la depuración...
```

1. *Línea 2:* Como he comentado al comienzo del capítulo, la librería urllib.request se basa en otra librería estándar de Python, http.client. Normalmente no necesitas tocar directamente http.client (el módulo urllib.request la importa automáticamente). Pero la importamos aquí para modificar el control de depuración de la clase HTTPConnection que es la que utiliza urllib.request para conectarse al servidor HTTP.

2. *Línea 4:* Ahora que se ha activado la depuración, la información de la petición de de la respuesta HTTP se imprime en tiempo real. Como puedes ver, cuando solicitas el flujo Atom, el módulo urllib.request envía cinco líneas al servidor.

3. *Línea 5:* La primera línea especifica el verbo HTTP que estás utilizando y el camino al recurso (menos el nombre de dominio).

4. *Línea 6:* La segunda línea indica el nombre de dominio del que estamos solicitando este flujo.

5. *Línea 7:* La tercera línea especifica los algoritmos de compresión que el cliente admite. Como he comentado antes, urllib.request no permite ningún tipo de compresión por defecto.

6. *Línea 8:* La cuarta línea especifica el nombre de la librería que está realizando la petición. Por defecto, se muestra Pythonurllib más el número de versión. Ambos módulos urllib.request y httplib2 permiten la modificación del agente de usuario, simplemente añadiendo la cabecera User-Agent a la petición (lo que sustituirá el valor por defecto).

Ahora vamos a ver lo que el servidor envía de vuelta como respuesta.

```
1  # sigue del ejemplo anterior
2  >>> print(response.headers.as_string())
3  Date: Sun, 31 May 2009 19:23:06 GMT
4  Server: Apache
5  Last-Modified: Sun, 31 May 2009 06:39:55 GMT
6  ETag: "bfe-93d9c4c0"
7  Accept-Ranges: bytes
8  Content-Length: 3070
9  Cache-Control: max-age=86400
10 Expires: Mon, 01 Jun 2009 19:23:06 GMT
11 Vary: Accept-Encoding
12 Connection: close
13 Content-Type: application/xml
14 >>> data = response.read()
15 >>> len(data)
16 3070
```

1. *Línea 2:* El objeto response devuelto por la función urllib.request.urlopen() contiene las cabeceras HTTP que el servidor ha devuelto. También contiene métodos para descargar la información real devuelta; volveremos a ello en un minuto.

2. *Línea 3:* El servidor te informa del momento en que procesó tu petición.

3. *Línea 5:* La respuesta incluye una cabecera Last-Modified.

4. *Línea 6:* Esta respuesta también incluye una cabecera ETag.

5. *Línea 8:* La información ocupa 3070 bytes. Observa lo que *no aparece*: una cabecera Content-encoding. Tu petición indicó que solamente aceptas información sin comprimir (Accept-encoding: identity), y estamos seguros de que esta respuesta solamente contiene información sin comprimir.

6. *Línea 9:* La respuesta incluye cabecerás de caché que indican que este flujo se puede mantener en caché durante 24 horas (86400 segundos).

7. *Línea 14:* Y finalmente, se descarga la información real mediante una llamada a response.read(). Como puedes ver mediante el uso de la función len(), la descarga es completa: los 3070 bytes de una vez.

Como puedes ver, este código es ineficiente: solicita (y recibe) datos sin comprimir. Sé con seguridad que este servidor soporta compresión gzip, pero la compresión en HTTP es opcional. No lo pedimos en este caso, por lo que no la obtuvimos. Esto significa que estamos descargando 3070 bytes cuando podríamos haber descargado solamente 941. Te has portado mal, no hay premio.

Pero espera, ¡que es peor! Para ver lo ineficiente que es este código vamos a pedir de nuevo el mismo recurso.

```
1 # sigue del ejemplo anterior
2 >>> response2 = urlopen('http://diveintopython3.org/examples/feed.xml')
3 send: b'GET /examples/feed.xml HTTP/1.1
4 Host: diveintopython3.org
5 Accept-Encoding: identity
6 User-Agent: Python-urllib/3.1'
7 Connection: close
8 reply: 'HTTP/1.1 200 OK'
9 ...further debugging information omitted...
```

¿No observas nada raro en esta petición? ¡no ha cambiado! Es exactamente la misma que la primera petición. No existe señal de las cabeceras If-Modified-Since. No hay señal de cabeceras If-None-Match. No se respetan las cabeceras de caché y no hay compresión.

Y ¿qué pasa cuando pides lo mismo dos veces? Pues que obtienes la misma respuesta ¡dos veces!

```
1  # sigue del ejemplo anterior
2  >>> print(response2.headers.as_string())
3  Date: Mon, 01 Jun 2009 03:58:00 GMT
4  Server: Apache
5  Last-Modified: Sun, 31 May 2009 22:51:11 GMT
6  ETag: "bfe-255ef5c0"
7  Accept-Ranges: bytes
8  Content-Length: 3070
9  Cache-Control: max-age=86400
10 Expires: Tue, 02 Jun 2009 03:58:00 GMT
11 Vary: Accept-Encoding
12 Connection: close
13 Content-Type: application/xml
14 >>> data2 = response2.read()
15 >>> len(data2)
16 3070
17 >>> data2 == data
18 True
```

1. *Línea 2:* El servidor envía de nuevo la misma lista de cabeceras "inteligentes": Cache-Control y Expires para permitir el uso de una caché, Last-Modified y ETag para facilitar el seguimiento de una modificación de la página. Incluso la cabecera Vary: Accept-Encoding que informa de que el servidor podría soportar compresión si se lo hubieras pedido. Pero no lo hiciste.

2. *Línea 15:* De nuevo, la recuperación de esta información descarga los 3070 bytes...

3. *Línea 17:* ...exactamente los mismos 3070 bytes que descargaste la última vez.

HTTP está diseñado para funcionar mejor que esto, urllib habla HTTP como yo hablo español —lo suficiente para integrarme en una fiesta, pero no lo suficente como para mantener una conversación. HTTP es una conversación. Es hora de actualizarnos a una librería que hable HTTP de forma fluida.

14.5. Introducción a httplib2

Antes de que puedas utilizar httplib2 necesitarás instalarla. Visita http://code.google.com/p/ y descarga la última versión. httplib2 está disponible para Python 2.x y para Python 3.x; asegúrate de que descargas la versión de Python 3, denominada algo así como httplib2-python3-0.5.0.zip.

Descomprime el archivo, abre el terminal en una ventana y vete al directorio recién creado httplib2. En Windows abre el menú Inicio, selecciona Ejecutar..., teclea cmd.exe y pulsa INTRO.

```
c:\Users\pilgrim\Downloads> dir
 Volume in drive C has no label.
 Volume Serial Number is DED5-B4F8

 Directory of c:\Users\pilgrim\Downloads

07/28/2009  12:36 PM    <DIR>          .
07/28/2009  12:36 PM    <DIR>          ..
07/28/2009  12:36 PM    <DIR>          httplib2-python3-0.5.0
07/28/2009  12:33 PM            18,997 httplib2-python3-0.5.0.zip
               1 File(s)         18,997 bytes
               3 Dir(s)  61,496,684,544 bytes free

c:\Users\pilgrim\Downloads> cd httplib2-python3-0.5.0
c:\Users\pilgrim\Downloads\httplib2-python3-0.5.0> c:\python31\python.exe
setup.py install
running install
running build
running build_py
running install_lib
creating c:\python31\Lib\site-packages\httplib2
copying build\lib\httplib2\iri2uri.py ->
 c:\python31\Lib\site-packages\httplib2
copying build\lib\httplib2\__init__.py ->
 c:\python31\Lib\site-packages\httplib2
byte-compiling c:\python31\Lib\site-packages\httplib2\iri2uri.py to
 iri2uri.pyc
byte-compiling c:\python31\Lib\site-packages\httplib2\__init__.py to
 __init__.pyc
running install_egg_info
Writing c:\python31\Lib\site-packages\
httplib2-python3_0.5.0-py3.1.egg-info
```

En Mac OS X ejecuta la aplicación Terminal.app de la carpeta /Aplicaciones/Utilidades. En Linux, ejecuta la aplicación de Terminal, que normalmente se encuentra en el menu de Aplicaciones bajo Accesorios o Sistema.

```
you@localhost:~/Desktop$ unzip httplib2-python3-0.5.0.zip
Archive:  httplib2-python3-0.5.0.zip
  inflating: httplib2-python3-0.5.0/README
  inflating: httplib2-python3-0.5.0/setup.py
  inflating: httplib2-python3-0.5.0/PKG-INFO
  inflating: httplib2-python3-0.5.0/httplib2/__init__.py
  inflating: httplib2-python3-0.5.0/httplib2/iri2uri.py
you@localhost:~/Desktop$ cd httplib2-python3-0.5.0/
you@localhost:~/Desktop/httplib2-python3-0.5.0$ sudo python3
 setup.py install
running install
running build
running build_py
creating build
creating build/lib.linux-x86_64-3.1
creating build/lib.linux-x86_64-3.1/httplib2
copying httplib2/iri2uri.py -> build/lib.linux-x86_64-3.1/httplib2
copying httplib2/__init__.py -> build/lib.linux-x86_64-3.1/httplib2
running install_lib
creating /usr/local/lib/python3.1/dist-packages/httplib2
copying build/lib.linux-x86_64-3.1/httplib2/iri2uri.py ->
 /usr/local/lib/python3.1/dist-packages/httplib2
copying build/lib.linux-x86_64-3.1/httplib2/__init__.py ->
 /usr/local/lib/python3.1/dist-packages/httplib2
byte-compiling /usr/local/lib/python3.1/dist-packages/httplib2/iri2uri.py
 to iri2uri.pyc
byte-compiling /usr/local/lib/python3.1/dist-packages/httplib2/__init__.py
 to __init__.pyc
running install_egg_info
Writing /usr/local/lib/python3.1/dist-packages/
httplib2-python3_0.5.0.egg-info
```

Para utilizar httplib2 crea una instancia de la clase httplib2.Http.

```
>>> import httplib2
>>> h = httplib2.Http('.cache')
>>> response, content = h.request(
        'http://diveintopython3.org/examples/feed.xml')
>>> response.status
200
>>> content[:52]
b"<?xml version='1.0' encoding='utf-8'?>\r\n<feed xmlns="
>>> len(content)
3070
```

1. *Línea 2:* El interfaz principal de httplib2 es el objeto Http. Por razones que verás en la siguiente sección, siempre deberías pasar el nombre de un directorio

al crear el objeto Http. No es necesario que el directorio exista, httplib2 lo creará si es necesario.

2. *Línea 3:* Una vez has creado el objeto Http, la recuperación de los datos es tan simple como llamar al método request() con la dirección de los datos que necesitas. Esto emitirá una petición HTTP GET para la URL deseada (Más adelante, en este capítulo, te mostraré cómo solicitar otro tipo de peticiones HTTP, como POST).

3. *Línea 4:* El método request() retorna dos valores. El primero es un objeto httplib2.Response, que contiene todas las cabeceras que retorna el servidor. Por ejemplo, un código de status 200 indica que la petición se completó satisfactoriamente.

4. *Línea 6:* La variable content contiene la información real que se retornó desde el servidor HTTP. La información se devuelve como un objeto bytes, no una cadena. Si quieres que sea una cadena, necesitas determinar la codificación de caracteres y convertirla tú mismo.

Probablemente necesites un único objeto httplib2.Http. No obstante, existen razones válidas por las que puedas necesitar más de uno, pero deberías crearlos únicamente si conoces porqué los necesitas. "Necesito datos desde dos URL diferentes" no es una razón válida; en su lugar, reutilizar el objeto Http y llama dos veces al método request().

14.5.1. Una breve disgresión para explicar porqué httplib2 devuelve Bytes en lugar de cadenas de texto

Bytes, cadenas de texto, ¡qué cansancio! ¿Porqué httplib2 no hace simplemente la conversión por ti? Bueno, es complejo, porque las reglas para determinar la codificación de caracteres son específicas del tipo de recurso que estés solicitano. ¿Cómo podría httplib2 conocer la clase de recurso que estás solicitando? Normalmente se encuentra en la cabecera Content-Type HTTP, pero se trata de una característica opcional de HTTP y no todos los servidores HTTP la incluyen. Si esa cabecera no está incluida en la respuesta HTTP, es el cliente el que tiene que adivinarlo (A esto se le suele llamar "inspección del contenido", y no es una solución perfecta).

Si supieras que clase de recursos estás esperando (un documento XML en este caso), tal vez podrías 'simplemente" pasar el objeto bytes a la función xml.etree.ElementTree.parse(). Eso funcionaría siempre que el documento XML incluyera la información de su propia codificación de caracteres (como hace en este caso), pero eso es una característica

opcional y no todos los documentos XML lo indican. Si un documento XML no incluye la información de codificación de caracteres, se supone que el cliente tiene que mirar en el protocolo de transporte —en este caso la cabecera Content-Type HTTP, que puede incluir el parámetro charset.

Pero es aún peor. Ahora la información sobre la codificación de caracteres puede encontrarse en dos lugares: dentro del propio documento XML y dentro de la cabecera Content-Type HTTP. Si la información está en *ambos* lugares... ¿cuál gana? De acuerdo a la especifiación RFC3023http://www.ietf.org/rfc/rfc3023.txt (te lo juro, no me lo estoy inventando), si el tipo de medio indicado en la cabecera Content-Type HTTP es application/xml, application/xml-dtd, application/xml-external-parsed-entity o cualquier otro subtipo de application/xml como application/atom+xml o incluso application/rdf+xml, entonces la codificación de caracteres es:

1. la codificación dada en el parámetro charset de la cabecera Content-Type HTTP o
2. la codificación dada en el atributo encoding de la declaración XML dentro del documento o
3. UTF-8

Por otra parte, si el tipo de medio dado en la cabecera Content-Type HTTP es text/xml, text/xml-external-parsed-entity o un subtipo como text/CualquierCosa+xml, entonces el atributo encoding de la declaración dentro del documento XML se ignora totalmente y la codificación es:

1. la indicada en el parámetro charset de la cabecera Content-Type HTTP o
2. us-ascii

Y eso únicamente para los documentos XML. Para los documentos HTML los navegadores han construido unas reglas tan bizantinas para identificación del contenidohttp://www.adambarth.com/papers/2009/barth-caballero-song.pdf que aún estamos intentando aclarar las que sonhttp://www.google.com/search?q=barth+content-type+processing+model.

14.5.2. Cómo **httplib2** gestiona la caché

¿Recuerdas cuando en la sección anterior te dije que deberías crear siempre el objeto httplib2.Http con un nombre de directorio? La razón es la caché.

```
1 # sigue del ejemplo anterior
2 >>> response2, content2 = h.request(
3         'http://diveintopython3.org/examples/feed.xml')
4 >>> response2.status
5 200
6 >>> content2[:52]
7 b"<?xml version='1.0' encoding='utf-8'?>\r\n<feed xmlns="
8 >>> len(content2)
9 3070
```

1. *Línea 2:* No deberías sorprenderte, es lo mismo que ya hemos hecho antes, excepto que el resultado lo estamos guardando en dos variables nuevas.

2. *Línea 3:* El HTTP status vuelve a ser 200, como antes.

3. *Línea 5:* El contenido descargado también es el mismo que la última vez.

Pero ¿A qué viene esto? Sal de la consola de Python y vuelve a relanzarla en una nueva sesión y te lo mostraré:

```
1 # NO sigue del ejemplo anterior
2 # Por favor, sal a de la consola de Python
3 # y vuelve a entrar en una nueva
4 >>> import httplib2
5 >>> httplib2.debuglevel = 1
6 >>> h = httplib2.Http('.cache')
7 >>> response, content = h.request(
8         'http://diveintopython3.org/examples/feed.xml')
9 >>> len(content)
10 3070
11 >>> response.status
12 200
13 >>> response.fromcache
14 True
```

1. *Línea 5:* Activamos la depuración pa ver lo que está sucediendo en la comunicación. Esta es la forma en la que httlib2 activa la depuración (como hacíamos antes con http.client). En este cao httplib2 imprimirá todos los datos que se envían al servidor e información clave que se retorna desde el mismo.

2. *Línea 6:* Se crea un objeto httplib2.Http con el mismo nombre de directorio que antes.

3. *Línea 7:* Se solicita la misma URL que antes. *Parece que no pasa nada.* De forma más precisa, nada se envía al servidor, y nada se devuelve del servidor. No hay ninguna actividad de red.

4. *Línea 8:* Pero sí que recibimos datos —de hecho, hemos recibido toda la información.

5. *Línea 10:* También hemos "recibido" el código de estado HTTP 200 indicando que la petición fue satisfactoria.

6. *Línea 12:* Este es el secreto: Esta respuesta se ha generado desde la caché local de httplib2. El directorio que le has pasado al rear el objeto httplib2.Http contiene la caché de httplib2 de todas las operaciones que se han ejecutado.

   Si quieres activar la depuración httlib2, necesitas activar una constante al nivel del módulo (httplib2.debuglevel) y luego crear un objeto nuevo httplib2.Http. Si quieres desactivar la depuración, necesitas modificar la misma constante y luego crear otro nuevo objeto httplib2.Http.

Anteriormente solicitaste información de esta URL. Las peticiones fueron satisfactorias (status: 200). Esta respuesta incluye no solamente los datos reales, sino también las cabeceras de caché que indican a quien recuperó los datos que los puede mantener en caché durante 24 horas (Cache-Control: max-age=86400, que son 24 horas medidas en segundos). httlib2 comprende y respeta las cabeceras de caché y almacena la respuesta anterior en el directorio .cache (que hemos pasado como parámetro al crear el objeto Http). Esta caché aún no ha expirado, por lo que la segunda vez que se solicita la información de esta URL httplib2 devuelve el resultado que tiene en la caché sin salir a la red a buscarlo.

Obviamente, esta simplicidad esconde la complejidad que supone esto: httplib2 maneja el cacheo de HTTP de forma *automática y por defecto*. Si por alguna razón necesitases conocer si la respuesta vino de la caché, puedes comprobar el valor de response.fromcache.

Ahora supón que tienes datos en la caché, pero quieres saltarte la caché y solicitar de nuevo los datos del servidor remoto. Los navegadores hacen esto si el usuario lo solicita específicamente. Por ejemplo, al pulsar F5 se refresca la página actual, pero al pulsar Ctrl-F5 se salta la caché y vuelve a consultar la página al servidor remoto. Podrías pensar "bastaría con borrar la caché local o volver a consultar la página actual al servidor remoto". Podrías hacer esto, pero recuerda que hay terceros involucrados en la consulta, no solamente tú y el servidor. ¿Qué pasa con los servidores proxy intermedios? Están completamente fuera de tu control y pueden tener aún datos en sus cachés. Datos que te devolverán porque para ellos la caché aún es válida.

En lugar de manipular la caché local y esperar que haya suerte, deberías utilizar las características que define el protocolo HTTP para asegurarte de que tu consulta realmente alcanza al servidor remoto.

```
1  # sigue del ejemplo anterior
2  >>> response2, content2 = h.request(
3          'http://diveintopython3.org/examples/feed.xml',
4  ...     headers={'cache-control':'no-cache'})
5  connect: (diveintopython3.org, 80)
6  send: b'GET /examples/feed.xml HTTP/1.1
7  Host: diveintopython3.org
8  user-agent: Python-httplib2/Rev:259
9  accept-encoding: deflate, gzip
10 cache-control: no-cache'
11 reply: 'HTTP/1.1 200 OK'
12 ...further debugging information omitted...
13 >>> response2.status
14 200
15 >>> response2.fromcache
16 False
17 >>> print(dict(response2.items()))
18 {'status': '200',
19  'content-length': '3070',
20  'content-location': 'http://diveintopython3.org/examples/feed.xml',
21  'accept-ranges': 'bytes',
22  'expires': 'Wed, 03 Jun 2009 00:40:26 GMT',
23  'vary': 'Accept-Encoding',
24  'server': 'Apache',
25  'last-modified': 'Sun, 31 May 2009 22:51:11 GMT',
26  'connection': 'close',
27  '-content-encoding': 'gzip',
28  'etag': '"bfe-255ef5c0"',
29  'cache-control': 'max-age=86400',
30  'date': 'Tue, 02 Jun 2009 00:40:26 GMT',
31  'content-type': 'application/xml'}
```

1. *Línea 4:* httplib2 te permite añadir cabeceras HTTP a cualquier petición saliente. Para poder saltarnos *todas* las cachés (no únicamente tu caché local, sino todas las cachés de los proxys entre el servidor remoto y tú), añade la cabecera no-cache en el diccionario headers.

2. *Línea 5:* Ahora se observa que httplib2 inicia una conexión a la red. httplib2 comprende y respeta las cabeceras de caché *en ambas direcciones* —como parte de la respuesta *y como parte de la petición de salida*. Detecta que has añadido una cabecera no-cache por lo que se salta la caché local y no tiene otra elección que salir a la red a solicitar la información.

3. *Línea 15:* Esta respuesta no se generó de la caché local. Ya lo sabías, viste la información de depuración de la petición de salida. Pero es bueno disponer de un modo de verificarlo desde el programa.

4. *Línea 17:* La petición terminó satisfactoriamente; descargaste la información completa de nuevo desde el servidor remoto. Desde luego, el servidor volvió a enviar toda la información de cabeceras junto con los datos. Incluye las cabeceras de caché, que httplib2 utilizará para actualizar su caché local, con la esperanza de evitar el acceso a la red la *siguiente vez* que solicites esta información. El cacheo HTTP está diseñado para maximizar el uso de la caché y minimizar el acceso a la red. Incluso aunque te hayas saltado la caché esta vez, el servidor remoto apreciaría que te guardases el resultado en la caché para la próxima vez.

14.5.3. Cómo httplib2 gestiona las cabeceras Last-Modified y ETag

Las cabeceras de caché Cache-Control y Expires se suelen denominar *indicadores de frescura*. Indican al sistema de caché de modo totalmente seguro que se puede evitar totalmente el acceso a la red hasta que expira el plazo. Y ése ha sido el comportamiento que has visto funcionando en la sección anterior: dado un indicador de frescura, httplib2 *no genera ningún byte de actividad en la red* para mostrar información que se encuentre en la caché (a menos que tú lo indiques expresamente, desde luego).

Pero qué sucede cuado los datos podrían haber cambiado. HTTP defne unas cabeceras Last-Modified y Etag para este fin. Estas cabeceras se denominan validadores. Si la caché local ya está desactualizada, el cliente puede enviar los validadores con la siguiente consulta, para ver si los datos han cambiado realmente. Si los datos no han cambiado, el servidor devuelve un código de estado 304 *y ningún dato más*. Por lo que aunque hay una consulta a la red, se descargan menos bytes.

```
1  >>> import httplib2
2  >>> httplib2.debuglevel = 1
3  >>> h = httplib2.Http('.cache')
4  >>> response, content = h.request('http://diveintopython3.org/')
5  connect: (diveintopython3.org, 80)
6  send: b'GET / HTTP/1.1
7  Host: diveintopython3.org
8  accept-encoding: deflate, gzip
9  user-agent: Python-httplib2/$Rev: 259 $'
10 reply: 'HTTP/1.1 200 OK'
11 >>> print(dict(response.items()))
12 {'-content-encoding': 'gzip',
13  'accept-ranges': 'bytes',
14  'connection': 'close',
15  'content-length': '6657',
16  'content-location': 'http://diveintopython3.org/',
17  'content-type': 'text/html',
18  'date': 'Tue, 02 Jun 2009 03:26:54 GMT',
19  'etag': '"7f806d-1a01-9fb97900"',
20  'last-modified': 'Tue, 02 Jun 2009 02:51:48 GMT',
21  'server': 'Apache',
22  'status': '200',
23  'vary': 'Accept-Encoding,User-Agent'}
24 >>> len(content)
25 6657
```

1. *Línea 4:* En lugar del fluo, esta vez vamos a descargar la página de inicio de la sede web, que es HTML.

2. *Línea 11:* La respuesta incluye muchas cabeceras HTTP... pero no incluye información de caché. Sin embargo, sí incluye cabeceras ETag y Last-Modified.

3. *Línea 24:* En el momento en que se incluyó este ejemplo, esta página era de 6657 bytes. Probablemente haya cambiado desde entonces, pero eso no es relevante.

```
1  # sigue del ejemplo anterior
2  >>> response, content = h.request('http://diveintopython3.org/')
3  connect: (diveintopython3.org, 80)
4  send: b'GET / HTTP/1.1
5  Host: diveintopython3.org
6  if-none-match: "7f806d-1a01-9fb97900"
7  if-modified-since: Tue, 02 Jun 2009 02:51:48 GMT
8  accept-encoding: deflate, gzip
9  user-agent: Python-httplib2/Rev:259'
10 reply: 'HTTP/1.1 304 Not Modified'
11 >>> response.fromcache
12 True
13 >>> response.status
14 200
15 >>> response.dict['status']
16 '304'
17 >>> len(content)
18 6657
```

1. *Línea 2:* Solicitas la misma página otra vez, con el mismo objeto Http (y la misma caché local).

2. *Línea 6:* httplib2 envía el valor de la etiqueta Etag al servidor en la cabecera If-None-Match.

3. *Línea 7:* httplib2 también envía el valor de la etiqueta Last-Modified al servidor en la etiqueta If-Modified-Since.

4. *Línea 10:* El servidor recupera estos valores (validadores), observa la página que has solicitado y determina que la página no ha cambiado desde la última vez que la solicitaste, por lo que devuelve un código 304 y *ningún dato real.*

5. *Línea 11:* De vuelta al cliente, httplib2 comprueba el código de estado 304 y carga el contenido de la página desde la caché.

6. *Línea 13:* Esto puede que sea algo extraño. En realidad existen *dos* códigos de estado —304 devuelto del servidor esta vez, que ha provocado que httplib2 recupere de la caché, y 200, devuelto del servidor la *última vez que se recuperaron los datos*, y que está almacenado en la caché de httplib2 junto con los datos. El atributo response.status obtiene el estado residente en la caché.

7. *Línea 15:* Si lo que se desea es ver el código de estado actual del servidor es necesario utilizar el diccionario response.dict, que mantiene las cabeceras recibidas en la consulta actual al servidor.

8. *Línea 17:* Sin embargo, aún tienes el tamaño de los datos en la variable content. Generalmente, no necestias conocer porqué una respuesta ha sido obtenida de la caché. Puede que ni te interese conocer si fue servida o no desde la caché. Cuando el método request() finaliza, httplib2 ya ha actualizado la caché, devolviéndote los datos que están en ella.

14.5.4. Cómo maneja la compresión httplib2

HTTP permite varios tipos de compresión; los dos más comunes son gzip y deflate. httplib2 permite ambos tipos.

```
1  >>> response, content = h.request('http://diveintopython3.org/')
2  connect: (diveintopython3.org, 80)
3  send: b'GET / HTTP/1.1
4  Host: diveintopython3.org
5  accept-encoding: deflate, gzip
6  user-agent: Python-httplib2/Rev:259'
7  reply: 'HTTP/1.1 200 OK'
8  >>> print(dict(response.items()))
9  {'-content-encoding': 'gzip',
10  'accept-ranges': 'bytes',
11  'connection': 'close',
12  'content-length': '6657',
13  'content-location': 'http://diveintopython3.org/',
14  'content-type': 'text/html',
15  'date': 'Tue, 02 Jun 2009 03:26:54 GMT',
16  'etag': '"7f806d-1a01-9fb97900"',
17  'last-modified': 'Tue, 02 Jun 2009 02:51:48 GMT',
18  'server': 'Apache',
19  'status': '304',
20  'vary': 'Accept-Encoding,User-Agent'}
```

1. *Línea 5:* Cada vez que httplib2 envía una petición, incluye una cabecera Accept-Encoding para indicarle al servidor que puede manipular compresiones gzip o deflate.

2. *Línea 9:* En este caso, el servidor ha respondido descargando la información comprimida con formato gzip. En el momento en que request() finaliza, httplib2 ya ha descomprimido el cuerpo de la respuesta y lo ha colocado en la variable content. Si tienes curiosidad de conocer si la respuesta original estaba o no codificada puedes consultar response['-content-encoding'].

14.5.5. Cómo maneja las redirecciones httplib2

HTTP define dos tipos de redirecciones: temporales y permanentes. No hay nada especial a hacer en las redirecciones temporales, excepto seguirlas, lo que httplib2 hace automáticamente.

```
1  >>> import httplib2
2  >>> httplib2.debuglevel = 1
3  >>> h = httplib2.Http('.cache')
4  >>> response, content = h.request(
5              'http://diveintopython3.org/examples/feed-302.xml')
6  connect: (diveintopython3.org, 80)
7  send: b'GET /examples/feed-302.xml HTTP/1.1
8  Host: diveintopython3.org
9  accept-encoding: deflate, gzip
10 user-agent: Python-httplib2/Rev:259'
11 reply: 'HTTP/1.1 302 Found'
12 send: b'GET /examples/feed.xml HTTP/1.1
13 Host: diveintopython3.org
14 accept-encoding: deflate, gzip
15 user-agent: Python-httplib2/Rev:259'
16 reply: 'HTTP/1.1 200 OK'
```

1. *Línea 5:* No existe ningún flujo de datos en esta URL. He configurado mi servidor para enviar una redirección temporal a la dirección correcta.

2. *Línea 7:* Esta es la petición.

3. *Línea 11:* Y esta es la respuesta 302 Found. No se muestra aquí, esta respuesta también incluye una cabecera Location que apunta a la dirección URL real.

4. *Línea 12:* httplib2 se dirige a la nueva dirección mediante una nueva petición a la URL indicada en la cabecera Location: http://diveintopython3.org/examples/feed.xml

La redirección no es más que lo se ha enseñado en este ejemplo. httlib2 envía una petición a la URL que le indicaste. El servidor devuelve una respuesta que dice "No, no, en vez de aquí, mira en este otro sitio". httlib2 envía una nueva petición automáticamente a la nueva URL.

```
1  # sigue del ejemplo anterior
2  >>> response
3  {'status': '200',
4   'content-length': '3070',
5   'content-location': 'http://diveintopython3.org/examples/feed.xml',
6   'accept-ranges': 'bytes',
7   'expires': 'Thu, 04 Jun 2009 02:21:41 GMT',
8   'vary': 'Accept-Encoding',
9   'server': 'Apache',
10  'last-modified': 'Wed, 03 Jun 2009 02:20:15 GMT',
11  'connection': 'close',
12  '-content-encoding': 'gzip',
13  'etag': '"bfe-4cbbf5c0"',
14  'cache-control': 'max-age=86400',
15  'date': 'Wed, 03 Jun 2009 02:21:41 GMT',
16  'content-type': 'application/xml'}
```

1. *Línea 2:* La respuesta que se obtiene en esta llamada, response del método request() es la respuesta de la URL final.

2. *Línea 5:* httplib2 añade la URL final al diccionario response como una cabecera content-location. No es una cabecera que viniera del servidor, es específica de httplib2.

3. *Línea 12:* Por cierto, este flujo está comprimido.

4. *Línea 14:* Y se puede guardar en la caché (Esto es importante, como verás en un minuto).

La respuesta que obtienes (response) te informa sobre la URL *final*. ¿Qué hay que hacer si quieres más información sobre las URLs intermedias, las que te llevaron a la última? httplib2 te lo permite.

```
1  # sigue del ejemplo anterior
2  >>> response.previous
3  {'status': '302',
4   'content-length': '228',
5   'content-location': 'http://diveintopython3.org/examples/feed-302.xml',
6   'expires': 'Thu, 04 Jun 2009 02:21:41 GMT',
7   'server': 'Apache',
8   'connection': 'close',
9   'location': 'http://diveintopython3.org/examples/feed.xml',
10  'cache-control': 'max-age=86400',
11  'date': 'Wed, 03 Jun 2009 02:21:41 GMT',
12  'content-type': 'text/html; charset=iso-8859-1'}
13 >>> type(response)
14 <class 'httplib2.Response'>
15 >>> type(response.previous)
16 <class 'httplib2.Response'>
17 >>> response.previous.previous
18 >>>
```

1. *Línea 2:* El atributo response.previous almacena una referencia al objeto response seguido anteriormente a que httplib2 obtuviera la respuesta actual.

2. *Línea 13:* Ambos objetos, response y response.previous, son del tipo httplib2.Response.

3. *Línea 17:* Esto significa que puedes hacer response.previous.previous para seguir la cadena de redireccionamiento más atrás en el tiempo. El escenario podría ser: una URL redirige a una segunda URL que a su vez redirige a una tercera ¡podría ser! En este ejemplo ya hemos alcanzado el comienzo de la cadena de redireccionamiento por lo que el atributo vale None.

¿Qué sucede si vuelves a pedir la misma URL?

```
1  # sigue del ejemplo anterior
2  >>> response2, content2 = h.request(
3          'http://diveintopython3.org/examples/feed-302.xml')
4  connect: (diveintopython3.org, 80)
5  send: b'GET /examples/feed-302.xml HTTP/1.1
6  Host: diveintopython3.org
7  accept-encoding: deflate, gzip
8  user-agent: Python-httplib2/Rev:259'
9  reply: 'HTTP/1.1 302 Found'
10 >>> content2 == content
11 True
```

1. *Línea 3:* Misma URL, mismo httplib.Http (y, por lo tanto, misma caché).

2. *Línea 5:* La respuesta 302 no se guardó en la caché, por eso httplib2 vuelve a enviar una petición por la misma URL.

3. *Línea 9:* De nuevo, el servidor responde con un código de estado 302. Pero observa lo que *no* ocurrió: No hay una segunda petición a la URL final, http://diveintopython.org/examples/feed.xml. Esta URL está en la caché (recuerda la cabecera Cache-Contro que viste en el ejemplo anterior. Una vez httplib2 recibe el código 302 Found, *comprueba la caché antes de pedir otra vez la página*. La caché contiene una copia vigente de http://diveintopython3.org/examples/feed.xml, por lo que no hay ninguna necesidad de volver a pedirla.

4. *Línea 10:* Al finalizar el método request() ha leído los datos de la caché y los ha devuelto. Por supuesto, son los mismos datos que recibiste la vez anterior.

En otras palabras, no tienes que hacer nada especial para el redireccionamiento temporal. httplib2 los sigue de forma automática, y el hecho de que una URL redirija a otra no afecta ni al soporte de compresión, caché, ETags o cualquier otra característica de HTTP.

El redireccionamiento permanente es igual de simple.

```
1  # sigue del ejemplo anterior
2  >>> response, content = h.request(
3          'http://diveintopython3.org/examples/feed-301.xml')
4  connect: (diveintopython3.org, 80)
5  send: b'GET /examples/feed-301.xml HTTP/1.1
6  Host: diveintopython3.org
7  accept-encoding: deflate, gzip
8  user-agent: Python-httplib2/Rev:259'
9  reply: 'HTTP/1.1 301 Moved Permanently'
10 >>> response.fromcache
11 True
```

1. *Línea 3:* De nuevo, esta URL no existe en realidad. He configurado mi servidor para que retorne una redirección permanente a http://diveintopython3.org/examples/feed.xml.

2. *Línea 9:* Aquí está: el código de estado 301. Pero de nuevo, observa lo que *no* ha sucedido: no hay una nueva petición a la URL nueva. ¿Por qué? porque ya está en la caché.

3. *Línea 10:* httplib2 "siguió" la redirección desde la caché.

Pero ¡espera, que hay más!

```
1 # sigue del ejemplo anterior
2 >>> response2, content2 = h.request(
3         'http://diveintopython3.org/examples/feed-301.xml')
4 >>> response2.fromcache
5 True
6 >>> content2 == content
7 True
```

1. *Línea 3:* Existe una diferencia entre las redirecciones temporales y permanentes: una vez que httplib2 sigue una redirección permanente, todas las peticiones siguientes a la misma URI serán solitadas de forma transparente a la nueva URL *sin pasar por la URL original*. Recuerda, la depuración está aún activada y sin embargo en este caso no se observa actividad en la red de ningún tipo.

2. *Línea 4:* Sí, la respuesta se obtuvo de la caché local.

3. *Línea 6:* Sí, el resultado es el flujo completo (de la caché).

HTTP funciona.

14.6. Un paso más allá de HTTP GET

Los servicios HTTP no se limitan a peticiones GET. ¿Qué sucede si quieres crear algo nuevo? Siempre que envías un comentario a un foro, actualizas tu blog, publicas tu estado en un servicio de microblog (como Twitter o Identi.ca, posiblemente estás usando HTTP POST.

Tanto Twitter como Identi.ca ofrecen una API simple basada en HTTP para publicar y actualizar tu estado en 140 caracteres o menos. Vamos a ver la documentación de la API de Identi.ca para actualizar tu estado[15]:

> **Método Identi.ca REST API: estados/actualizaciones** Actualiza el estado del usuario autenticado. Requiere el parámetro status especificado más abajo. La petición debe ser un POST.
>
> URL - http://identi.ca/api/statuses/update.format
>
> Formatos - xml, json, rss, atom
>
> Métodos HTTP - POST
>
> Necesita autenticación - true

[15] http://laconi.ca/trac/wiki/TwitterCompatibleAPI

Parámetros: status. Requerido. El texto de la actualización de tu estado. Codificación URL.

¿Cómo funciona esto? Para publicar un nuevo mensaje en Identi.ca, necesitas enviar una petición HTTP POST a https://identi.ca/api/statuses/update.*format* (La parte del format no es parte de la URL; lo sustituyes por el formato de datos que quieres que el servidor retorne en respuesta a tu petición. Por lo que si quieres la respuesta en XML deberías enviar la petición a https://identi.ca/api/statuses/update.xml). La petición debe incluir un parámetro denominado status que contiene el texto de la actualización de tu estado que desees. Y la petición necesita autentificación.

¿Autentificación? Para poder actualizar tu estado en Identi.ca, necesitas probar que eres quien dices que eres. Identi.ca no es una wiki; solamente puedes actualizar tu propio estado. Identica.ca utiliza autenticación básica HTTP[16] (RFC 2627[17]) sobre SSL para proporcionar una autentificación segura pero sencilla de usar. httplib2 soporta tanto SSL como autentificación básica HTTP, por lo que esta parte es sencilla.

Una petición POST es diferente a una GET, porque incluye información que hay que enviar al servidor. En el caso de este método de la API de Identi.ca, se *requiere* la información del parámetro status y debería estar en codificación URL. Este es un formato muy simple de serialización que toma un conjunto de parejas clave-valor (como un diccionario) y lo transforma en una cadena.

```
1 >>> from urllib.parse import urlencode
2 >>> data = {'status': 'Test update from Python 3'}
3 >>> urlencode(data)
4 'status=Test+update+from+Python+3'
```

1. *Línea 1:* Python dispone de una función de utiliza para codificar en este formato cualquier diccionario: urllib.parse.urlencode().

2. *Línea 2:* Este es el tipo de diccionario que la API de Identi.ca está esperando. Contiene una clave, status, cuyo valor es el texto de una actualización de estado.

3. *Línea 3:* Así queda el diccionario una vez está en el formato codificado URL. Esta es la información que deberá enviarse por la red al servidor de Identi.ca en la petición HTTP POST.

[16] http://en.wikipedia.org/wiki/Basic_access_authentication
[17] http://www.ietf.org/rfc/rfc2617.txt

```
1  >>> from urllib.parse import urlencode
2  >>> import httplib2
3  >>> httplib2.debuglevel = 1
4  >>> h = httplib2.Http('.cache')
5  >>> data = {'status': 'Test update from Python 3'}
6  >>> h.add_credentials('diveintomark',
7          'MY_SECRET_PASSWORD', 'identi.ca')
8  >>> resp, content = h.request(
9          'https://identi.ca/api/statuses/update.xml',
10 ...      'POST',
11 ...      urlencode(data),
12 ...      headers={'Content-Type': 'application/x-www-form-urlencoded'})
```

1. *Línea 7:* Así es como httplib2 controla la autentificación. Almacena tu código de usuario y clave de acceso con el método add_credentials(). Cuando httplib2 intenta realizar la petición, el servidor responderá con un código de estado 401 Unauthorized, y una lista de métodos de autentificación disponibles (en la cabecera WWW-Authenticate). httlib2 construirá automáticamente una cabecera de Authorization y solicitará la URL.

2. *Línea 10:* El segundo parámetro del método request es el tipo de petición HTTP, en este caso POST.

3. *Línea 11:* El tercer parámetro es la información que se envía al servidor. Estamos enviando el diccionario codificado en URL con el mensaje de estado.

4. *Línea 12:* Finalmente, necesitamos decirle al servidor que la información enviada se encuentra en el formato URL-encoded.

   El tercer parámetro del método add_credentials() es el domínio en el que las credenciales son válidas. ¡Deberías especificar esto siempre! Si dejas el dominio sin especificar y luego reutilizas el objeto httplib2.Http en un sitio diferente que requiera autentificación, httplib2 podría acabar enviando el usuario y clave de una sede web a otra diferente.

   Esto es lo que pasa por debajo:

```
1  # sigue del ejemplo anterior
2  send: b'POST /api/statuses/update.xml HTTP/1.1
3  Host: identi.ca
4  Accept-Encoding: identity
5  Content-Length: 32
6  content-type: application/x-www-form-urlencoded
7  user-agent: Python-httplib2/$Rev: 259 $
8
9  status=Test+update+from+Python+3'
10 reply: 'HTTP/1.1 401 Unauthorized'
11 send: b'POST /api/statuses/update.xml HTTP/1.1
12 Host: identi.ca
13 Accept-Encoding: identity
14 Content-Length: 32
15 content-type: application/x-www-form-urlencoded
16 authorization: Basic SECRET_HASH_CONSTRUCTED_BY_HTTPLIB2
17 user-agent: Python-httplib2/$Rev: 259 $
18
19 status=Test+update+from+Python+3'
20 reply: 'HTTP/1.1 200 OK'
```

1. *Línea 10:* Después de la primera petición, el servidor responde con un código de estado 401 Unauthorized. httplib2 nunca enviará una cabecera de autentificación a no ser que el servidor la solicite expresamente. Esta es la forma en la que el servidor lo hace.

2. *Línea 11:* httplib2 inmediatamente vuelve a pedir la URL por segunda vez.

3. *Línea 16:* Esta vez, incluye el código de usuario y clave de acceso que añadiste en el método add_credentials().

4. *Línea 20:* ¡Funcionó!

¿Qué envía el servidor después de una petición que se completó satisfactoriamente? Depende totalmente de la API del servicio web. En algunos protocolos (como el protocolo de publicación Atom) el servidor devuelve un código de estado 201 Created y la localización del recurso recién creado en la cabecera Location. Identi.ca responde con un 200 OK y un documento XML que contiene información sobre el recurso recién creado.

```
1  # sigue del ejemplo anterior
2  >>> print(content.decode('utf-8'))
3  <?xml version="1.0" encoding="UTF-8"?>
4  <status>
5   <text>Test update from Python 3</text>
6   <truncated>false</truncated>
7   <created_at>Wed Jun 10 03:53:46 +0000 2009</created_at>
8   <in_reply_to_status_id></in_reply_to_status_id>
9   <source>api</source>
10  <id>5131472</id>
11  <in_reply_to_user_id></in_reply_to_user_id>
12  <in_reply_to_screen_name></in_reply_to_screen_name>
13  <favorited>false</favorited>
14  <user>
15   <id>3212</id>
16   <name>Mark Pilgrim</name>
17   <screen_name>diveintomark</screen_name>
18   <location>27502, US</location>
19   <description>tech writer, husband, father</description>
20   <profile_image_url>http://avatar.identi.ca/
21 3212-48-20081216000626.png</profile_image_url>
22   <url>http://diveintomark.org/</url>
23   <protected>false</protected>
24   <followers_count>329</followers_count>
25   <profile_background_color></profile_background_color>
26   <profile_text_color></profile_text_color>
27   <profile_link_color></profile_link_color>
28   <profile_sidebar_fill_color></profile_sidebar_fill_color>
29   <profile_sidebar_border_color></profile_sidebar_border_color>
30   <friends_count>2</friends_count>
31   <created_at>Wed Jul 02 22:03:58 +0000 2008</created_at>
32   <favourites_count>30768</favourites_count>
33   <utc_offset>0</utc_offset>
34   <time_zone>UTC</time_zone>
35   <profile_background_image_url></profile_background_image_url>
36   <profile_background_tile>false</profile_background_tile>
37   <statuses_count>122</statuses_count>
38   <following>false</following>
39   <notifications>false</notifications>
40 </user>
41 </status>
```

1. *Línea 2:* Recuerda, los datos devueltos por httplib2 son siempre bytes, no cadenas de texto. Para convertirlos a una cadena de texto, necesitas decodificarlos utilizando la codificación de caracteres apropiada. La API de Identi.ca siempre devuelve los resultados en UTF-8, así que esto es fácil.

2. *Línea 5:* Este es el texto del mensaje de estado recién publicado.

3. *Línea 10:* Este es el identificador único del nuevo mensaje. Idnti.ca lo utiliza para construir una URL para poder ver el mensaje en la web: http://identi.ca/notice/5131472

14.7. Más allá de HTTP POST

HTTP no está limitado a GET y POST. Son los dos tipos más comunes de peticiones, especialmente en los navegadores web. Pero las APIs de servicios web pueden ir más allá de ellos, y httplib2 está preparado.

```
1 # sigue del ejemplo anterior
2 >>> from xml.etree import ElementTree as etree
3 >>> tree = etree.fromstring(content)
4 >>> status_id = tree.findtext('id')
5 >>> status_id
6 '5131472'
7 >>> url = 'https://identi.ca/api/statuses/destroy/{0}.xml'.format(
8             status_id)
9 >>> resp, deleted_content = h.request(url, 'DELETE')
```

1. *Línea 3:* El servidor retornó XML ¿correcto? Ya sabes cómo analizar un documento XML.

2. *Línea 4:* El método findtext() encuentra la primera ocurrencia de una expresión dada y devuelve el texto del contenido. En este caso, únicamente estamos buscando el elemento id.

3. *Línea 8:* Basándonos en el contenido del elemento id podemos construir una URL para borrar el mensaje de estado que acabamos de publicar.

4. *Línea 9:* Para borrar un mensaje, simplemente envía una petición HTTP DELETE a esa URL.

Esto es lo que sucede por debajo:

```
1  send: b'DELETE /api/statuses/destroy/5131472.xml HTTP/1.1
2  Host: identi.ca
3  Accept-Encoding: identity
4  user-agent: Python-httplib2/Rev: 259
5
6  '
7  reply: 'HTTP/1.1 401 Unauthorized'
8  send: b'DELETE /api/statuses/destroy/5131472.xml HTTP/1.1
9  Host: identi.ca
10 Accept-Encoding: identity
11 authorization: Basic SECRET_HASH_CONSTRUCTED_BY_HTTPLIB2
12 user-agent: Python-httplib2/Rev: 259
13
14 '
15 reply: 'HTTP/1.1 200 OK'
16 >>> resp.status
17 200
```

1. *Línea 1:* "Borra este mensaje de estado".
2. *Línea 7:* "Lo siento, no puedo hacer eso sin saber quién lo pide"
3. *Línea 8:* "¿No tengo permiso? Soy yo, borra el mensaje, por favor..."
4. *Línea 11:* "...y estos son mi código de usuario y clave de acceso."
5. *Línea 15:* "Considéralo hecho".

Así que ahora al intentar el enlace http://identi.ca/notice/5131472 en un navegador, se obtiene "Not Found".

14.8. Lecturas recomendadas

httplib2:

- Página del proyecto httplib2:
  http://code.google.com/p/httplib2/
- Más ejemplos de código httplib2:
  http://code.google.com/p/httplib2/wiki/ExamplesPython3
- Cómo hacer correctamente la caché de HTTP: Introducción a httplib2:
  http://www.xml.com/pub/a/2006/02/01/doing-http-caching-right-introducing-httplib2.html

- httplib2: Persistencia y autentificación HTTP:
  http://www.xml.com/pub/a/2006/03/29/httplib2-http-persistence-and-authentication.html

Caché HTTP:

- Tutorial sobre la caché en HTTP de Mark Nottingham:
  http://www.mnot.net/cache_docs/
- Cómo controlar la caché con cabeceras de HTTP en Google Doctype
  http://code.google.com/p/doctype/wiki/ArticleHttpCaching

RFCs:

- RFC 2616: HTTP
  http://www.ietf.org/rfc/rfc2616.txt
- RFC 2617: HTTP, autenticación básica
  http://www.ietf.org/rfc/rfc2617.txt
- RFC 1951: formato de compresión deflate
  http://www.ietf.org/rfc/rfc1951.txt
- RFC 1952: formato de compresión gzip
  http://www.ietf.org/rfc/rfc1952.txt