Inmersión en Python

Python de novato a experto

Inmersión en Python

25 de enero de 2005

Este libro y su traducción al español se encuentran en http://diveintopython.org/. Si usted lo está leyendo o lo ha obtenido en otro lugar, es posible que no disponga de la última versión.

Tabla de contenidos

Capítulo 1. Instalación de Python

Bienvenido a Python. Zambullámonos. En este capítulo, vamos a instalar la versión de Python que sea más adecuada para usted.

1.1. ¿Qué Python es adecuado para usted?

La primera cosa que debe hacer con Python es instalarlo. ¿O no?

Si está usando una cuenta en un servidor alquilado, puede que el ISP ya haya instalado Python. Las distribuciones de Linux más populares incluyen Python en la instalación predeterminada. Mac OS X 10.2 y posteriores incluyen una versión de Python para la línea de órdenes, aunque probablemente quiera instalar una versión que incluya una interfaz gráfica más acorde con Mac.

Windows no incluye una versión de Python, ¡pero no desespere! Hay varias maneras de llegar a Python en Windows a golpe de ratón.

Como puede ver, Python funciona en una gran cantidad de sistemas operativos. La lista completa incluye Windows, Mac OS, Mac OS X, y todas las variedades de sistemas libres compatibles con UNIX, como Linux. También hay versiones que funcionan en Sun Solaris, AS/400, Amiga, OS/2, BeOS, y una plétora de otras plataformas de las que posiblemente no haya oído hablar siquiera.

Es más, los programas escritos para Python en una plataforma pueden funcionar, con algo de cuidado, en *cualquiera* de las plataformas soportadas. Por ejemplo, habitualmente desarrollo programas para Python en Windows que luego funcionarán en Linux.

De manera que, de vuelta a la pregunta que comenzó esta sección: "¿qué Python es adecuado para usted?". La respuesta es cualquiera que funcione en el computador que posea.

1.2. Python en Windows

En Windows debe hacer un par de elecciones antes de instalar Python.

ActiveState fabrica un instalador de Windows para Python llamado ActivePython, que incluye una versión completa de Python, un IDE con editor de código preparado para Python, así como algunas extensiones para Python propias de Windows que le permiten un acceso completo a servicios específicos, APIs, y al registro de Windows.

ActivePython es de descarga gratuita, aunque no es open source. Es el IDE que utilicé para aprender Python, y le recomiendo que lo pruebe a menos que tenga una razón específica para no hacerlo. Una de estas razones podría ser que ActiveState tarda generalmente varios meses en actualizar su instalador ActivePython con las versiones nuevas de Python que se publican. Si necesita absolutamente la última versión de Python y ActivePython aún se encuentra en una versión anterior cuando lea esto, deberá usar la segunda opción para instalar Python en Windows.

La segunda opción es el instalador "oficial" de Python, distribuido por la propia gente que hace Python. Es de libre descarga y open source, y siempre está al día con la última versión.

Procedimiento 1.1. Opción 1: Instalar ActivePython

Éste es el procedimiento para instalar ActivePython:

1. Descargue ActivePython de http://www.activestate.com/Products/ActivePython/.
2. Si está usando Windows 95, Windows 98, o Windows ME, también necesitará descargar e instalar Windows Installer 2.0 antes de instalar ActivePython.
3. Haga doble clic sobre el instalador, `ActivePython-2.2.2-224-win32-ix86.msi`.
4. Siga los pasos que indique el instalador.
5. Si le falta espacio en el disco, puede realizar una instalación a medida y eliminar la documentación, pero no se lo recomiendo a menos que le sean preciosos esos 14MB.
6. Tras completar la instalación, cierre el instalador y escoja Inicio->Programas->ActiveState ActivePython 2.2->PythonWin IDE. Verá algo como lo siguiente:

```
PythonWin 2.2.2 (#37, Nov 26 2002, 10:24:37) [MSC 32 bit (Intel)] on
win32.
Portions Copyright 1994-2001 Mark Hammond (mhammond@skippinet.com.au)
-
see 'Help/About PythonWin' for further copyright information.
>>>
```

Procedimiento 1.2. Opción 2: Instalar Python de Python.org

1. Descargue el último instalador de Python para Windows yendo a http://www.python.org/ftp/python/ y escogiendo el número de versión más alto que esté en la lista, para descargar el instalador `.exe`.
2. Haga doble clic en el instalador, `Python-2.xxx.yyy.exe`. El nombre dependerá de la versión de Python disponible cuando lea esto.
3. Siga los pasos que indique el instalador.
4. Si le falta espacio en el disco, puede eliminar el fichero HTMLHelp, los *scripts* de utilidades (`Tools/`), y la batería de pruebas (`Lib/test/`).

5. Si no dispone de derechos administrativos en su máquina, puede escoger Advanced Options, y elegir entonces Non-Admin Install. Esto sólo afecta al lugar donde se crean las entradas en el Registro y los atajos en el menú Inicio.
6. Tras completar la instalación, cierre el instalador y escoja Inicio->Programas->Python 2.3->IDLE (Python GUI). Verá algo como lo siguiente:

```
Python 2.3.2 (#49, Oct  2 2003, 20:02:00) [MSC v.1200 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.

    ****************************************************************
    Personal firewall software may warn about the connection IDLE
    makes to its subprocess using this computer's internal loopback
    interface.  This connection is not visible on any external
    interface and no data is sent to or received from the Internet.
    ****************************************************************

IDLE 1.0
>>>
```

1.3. Python en Mac OS X

En Mac OS X cuenta con dos opciones para instalar Python: instalarlo o no instalarlo. Probablemente quiera instalarlo.

Mac OS X 10.2 y posteriores incluyen de serie una versión de Python para la línea de órdenes (el emulador de terminal). Si se encuentra cómodo en ese entorno, puede usar esta versión para el primer tercio del libro. Sin embargo, la versión preinstalada no incluye un analizador de XML, de manera que cuando lleguemos al capítulo de XML necesitará instalar la versión completa.

En lugar de usar la versión preinstalada, probablemente desee instalar la última versión, que también incluye un intérprete interactivo (*shell*) gráfico.

Procedimiento 1.3. Ejecución de la Versión Preinstalada de Python en Mac OS X

Para usar la versión preinstalada de Python, siga estos pasos:

1. Abra la carpeta `/Aplicaciones`.
2. Abra la carpeta `Utilidades`.
3. Haga doble clic sobre `Terminal` para abrir una ventana de terminal y acceder a la línea de órdenes.
4. Escriba **`python`** en la línea de órdenes.

Pruebe:

```
Welcome to Darwin!
[localhost:~] usted% python
Python 2.2 (#1, 07/14/02, 23:25:09)
[GCC Apple cpp-precomp 6.14] on darwin
Type "help", "copyright", "credits", or "license" for more
information.
>>> [pulse Ctrl+D para volver a la línea de órdenes]
[localhost:~] usted%
```

Procedimiento 1.4. Instalación de la última versión de Python en Mac OS X

Siga estos pasos para descargar e instalar la última versión de Python:

1. Descargue la imagen de disco `MacPython-OSX` desde http://homepages.cwi.nl/~jack/macpython/download.html.
2. Si su navegador no lo ha hecho ya, haga doble clic sobre `MacPython-OSX-2.3-1.dmg` para montar la imagen de disco en el escritorio.
3. Haga doble clic en el instalador, `MacPython-OSX.pkg`.
4. El instalador le pedirá su nombre de usuario y clave administrativos.
5. Siga los pasos del instalador.
6. Tras completar la instalación, cierre el instalador y abra la carpeta `/Aplicaciones`.

7. Abra la carpeta `MacPython-2.3`
8. Haga doble clic en `PythonIDE` para lanzar Python.

El IDE MacPython debería mostrar una ventana de inicio, y luego mostarle el intérprete interactivo. Si no aparece el intérprete, escoja Ventana->Python Interactive (**Cmd-0**). La ventana que se abra tendrá este aspecto:

```
Python 2.3 (#2, Jul 30 2003, 11:45:28)
[GCC 3.1 20020420 (prerelease)]
Type "copyright", "credits" or "license" for more information.
MacPython IDE 1.0.1
>>>
```

Tenga en cuenta que una vez instale la última versión, la preinstalada seguirá presente. Si ejecuta *scripts* desde la línea de órdenes, debe saber qué versión de Python está usando.

Ejemplo 1.1. Dos versiones de Python

```
[localhost:~] usted% python
Python 2.2 (#1, 07/14/02, 23:25:09)
[GCC Apple cpp-precomp 6.14] on darwin
Type "help", "copyright", "credits", or "license" for more
information.
>>> [pulse Ctrl+D para volver a la línea de órdenes]
[localhost:~] usted% /usr/local/bin/python
Python 2.3 (#2, Jul 30 2003, 11:45:28)
[GCC 3.1 20020420 (prerelease)] on darwin
Type "help", "copyright", "credits", or "license" for more
information.
>>> [pulse Ctrl+D para volver a la línea de órdenes]
[localhost:~] usted%
```

1.4. Python en Mac OS 9

Mac OS 9 no incluye una versión de Python pero instalarla es muy sencillo, y sólo hay una opción.

Siga estos pasos para instalar Python en Mac OS 9:

1. Descargue el fichero `MacPython23full.bin` desde http://homepages.cwi.nl/~jack/macpython/download.html.
2. Si su navegador no descomprime el fichero automáticamente, haga doble clic `MacPython23full.bin` para descomprimir el fichero con Stuffit Expander.
3. Haga doble clic sobre el instalador, `MacPython23full`.
4. Siga los pasos del programa instalador.
5. Tras completar la instalación, cierre el instalador y abra la carpeta `/Aplicaciones`.
6. Abra la carpeta `MacPython-OS9 2.3`.
7. Haga doble clic en `Python IDE` para lanzar Python.

El IDE MacPython debería mostrar una pantalla de inicio, y entonces llevarle al intérprete interactivo. Si no aparece el intérprete, escoja Ventana->Python Interactive (**Cmd-0**). Verá una pantalla parecida a ésta:

```
Python 2.3 (#2, Jul 30 2003, 11:45:28)
[GCC 3.1 20020420 (prerelease)]
Type "copyright", "credits" or "license" for more information.
MacPython IDE 1.0.1
>>>
```

1.5. Python en RedHat Linux

La instalación en sistemas operativos compatibles con Unix como Linux es sencilla si desea instalar un paquete binario. Existen paquetes binarios precompilados disponibles para las distribuciones de Linux más populares. Aunque siempre puede compilar el código fuente.

Descarge el último RPM de Python yendo a http://www.python.org/ftp/python/ y escogiendo el número de versión más alto en la lista, y dentro de ahí, el directorio `rpms/`. Entonces descargue el RPM con el número de versión más alto. Puede instalarlo con la orden **rpm**, como se muestra aquí:

Ejemplo 1.2. Instalación en RedHat Linux 9

```
localhost:~$ su -
Password: [introduzca la clave de root]
[root@localhost root]# wget
http://python.org/ftp/python/2.3/rpms/redhat-9/python2.3-2.3-
5pydotorg.i386.rpm
Resolving python.org... done.
Connecting to python.org[194.109.137.226]:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 7,495,111 [application/octet-stream]
...
[root@localhost root]# rpm -Uvh python2.3-2.3-5pydotorg.i386.rpm
Preparing...
########################################### [100%]
   1:python2.3
########################################### [100%]
[root@localhost root]# python          ❶
Python 2.2.2 (#1, Feb 24 2003, 19:13:11)
[GCC 3.2.2 20030222 (Red Hat Linux 3.2.2-4)] on linux2
Type "help", "copyright", "credits", or "license" for more
information.
>>> [pulse Ctrl+D para salir]
[root@localhost root]# python2.3       ❷
Python 2.3 (#1, Sep 12 2003, 10:53:56)
[GCC 3.2.2 20030222 (Red Hat Linux 3.2.2-5)] on linux2
Type "help", "copyright", "credits", or "license" for more
information.
>>> [pulse Ctrl+D para salir]
[root@localhost root]# which python2.3 ❸
/usr/bin/python2.3
```

❶ ¡Vaya! Escribir **`python`** sólamente le lleva a la versión anterior de Python (la que incluía la instalación). Esa no es la que usted quiere.

❷ En el momento de escribir esto, la versión más moderna se llama **`python2.3`**. Probablemente quiera cambiar la ruta en la primera línea de los *script* de ejemplo para que apunten a la nueva versión.

❸ Ésta es la ruta completa de la versión más moderna de Python que acaba de instalar. Utilize esto en la línea `#!` (la primera de cada *script*) para asegurarse

de que los *scripts* se ejecutan usando la última versión de Python, y asegúrese de escribir **`python2.3`** para entrar en el intérprete.

1.6. Python en Debian GNU/Linux

Si tiene la suerte de usar Debian GNU/Linux, instale Python usando la orden **apt**.

Ejemplo 1.3. Instalación en Debian GNU/Linux

```
localhost:~$ su -
Password: [introduzca la clave de root]
localhost:~# apt-get install python
Reading Package Lists... Done
Building Dependency Tree... Done
The following extra packages will be installed:
  python2.3
Suggested packages:
  python-tk python2.3-doc
The following NEW packages will be installed:
  python python2.3
0 upgraded, 2 newly installed, 0 to remove and 3 not upgraded.
Need to get 0B/2880kB of archives.
After unpacking 9351kB of additional disk space will be used.
Do you want to continue? [Y/n] Y
Selecting previously deselected package python2.3.
(Reading database ... 22848 files and directories currently
installed.)
Unpacking python2.3 (from .../python2.3_2.3.1-1_i386.deb) ...
Selecting previously deselected package python.
Unpacking python (from .../python_2.3.1-1_all.deb) ...
Setting up python (2.3.1-1) ...
Setting up python2.3 (2.3.1-1) ...
Compiling python modules in /usr/lib/python2.3 ...
Compiling optimized python modules in /usr/lib/python2.3 ...
localhost:~# exit
logout
localhost:~$ python
Python 2.3.1 (#2, Sep 24 2003, 11:39:14)
[GCC 3.3.2 20030908 (Debian prerelease)] on linux2
```

```
Type "help", "copyright", "credits" or "license" for more information.
>>> [pulse Ctrl+D para salir]
```

1.7. Instalación de Python desde el Código Fuente

Si prefiere compilar el código fuente, puede descargar el de Python desde http://www.python.org/ftp/python/. Escoja el número de versión más alto, descargue el fichero `.tgz`, y ejecute entonces el ritual habitual de **`configure`**, **`make`**, **`make install`**.

Ejemplo 1.4. Instalación desde el código fuente

```
localhost:~$ su -
Password: [introduzca la clave de root]
localhost:~# wget http://www.python.org/ftp/python/2.3/Python-2.3.tgz
Resolving www.python.org... done.
Connecting to www.python.org[194.109.137.226]:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 8,436,880 [application/x-tar]
...
localhost:~# tar xfz Python-2.3.tgz
localhost:~# cd Python-2.3
localhost:~/Python-2.3# ./configure
checking MACHDEP... linux2
checking EXTRAPLATDIR...
checking for --without-gcc... no
...
localhost:~/Python-2.3# make
gcc -pthread -c -fno-strict-aliasing -DNDEBUG -g -O3 -Wall -Wstrict-
prototypes
-I. -I./Include  -DPy_BUILD_CORE -o Modules/python.o Modules/python.c
gcc -pthread -c -fno-strict-aliasing -DNDEBUG -g -O3 -Wall -Wstrict-
prototypes
-I. -I./Include  -DPy_BUILD_CORE -o Parser/acceler.o Parser/acceler.c
gcc -pthread -c -fno-strict-aliasing -DNDEBUG -g -O3 -Wall -Wstrict-
prototypes
-I. -I./Include  -DPy_BUILD_CORE -o Parser/grammar1.o
Parser/grammar1.c
```

```
...
localhost:~/Python-2.3# make install
/usr/bin/install -c python /usr/local/bin/python2.3
...
localhost:~/Python-2.3# exit
logout
localhost:~$ which python
/usr/local/bin/python
localhost:~$ python
Python 2.3.1 (#2, Sep 24 2003, 11:39:14)
[GCC 3.3.2 20030908 (Debian prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> [pulse Ctrl+D para volver a la línea de órdenes]
localhost:~$
```

1.8. El intérprete interactivo

Ahora que ya ha instalado Python, ¿qué es este intérprete interactivo que está ejecutando?

Es algo así: Python lleva una doble vida. Es un intérprete para *scripts* que puede ejecutar desde la línea de órdenes o como aplicaciones si hace clic dos veces sobre sus iconos. Pero también es un intérprete interactivo que puede evaluar sentencias y expresiones arbitrarias. Esto es muy útil para la depuración, programación rápida y pruebas. ¡Incluso conozco gente que usa el intérprete interactivo de Python a modo de calculadora!

Lance el intérprete interactivo en la manera que se haga en su plataforma, y zambullámonos en él con los pasos que se muestran aquí:

Ejemplo 1.5. Primeros pasos en el Intérprete Interactivo

```
>>> 1 + 1                ❶
2
>>> print 'hola mundo'   ❷
hola mundo
>>> x = 1                ❸
>>> y = 2
>>> x + y
```

1. El intérprete interactivo de Python puede evaluar expresiones de Python arbitrarias, incluyendo expresiones aritméticas básicas.
2. El intérprete interactivo puede ejecutar sentencias de Python arbitrarias, incluyendo la sentencia **print**.
3. También puede asignar valores a las variables, y estos valores serán recordados mientras el intérprete siga abierto (pero no más allá de eso).

1.9. Resumen

Ahora debería tener una versión de Python instalada que funcione.

Dependiendo de la plataforma, puede que tenga más de una versión de Python instalada. Si es el caso, debe tener cuidado con las rutas. Si escribir simplemente **python** en la línea de órdenes no ejecuta la versión de Python que quiere usar, puede que necesite introducir la ruta completa hasta su versión preferida.

Felicidades, y bienvenido a Python.

Capítulo 2. Su primer programa en Python

¿Sabe cómo empiezan otros libros hablando de fundamentos de programación y acaban construyendo un programa completo y que funciona? Saltémonos todo eso.

2.1. Inmersión

Aquí tiene un programa en Python, completo y funcional.

Probablemente no tenga mucho sentido para usted. No se preocupe por eso, porque voy a hacer una disección línea por línea. Pero léalo todo antes y vea si puede comprender algo.

Ejemplo 2.1. `odbchelper.py`

Si aún no lo ha hecho, puede descargar éste ejemplo y otros usados en este libro.

```
def buildConnectionString(params):
    """Crea una cadena de conexión partiendo de un diccionario de
parámetros.
```

```
    Devuelve una cadena."""
    return ";".join(["%s=%s" % (k, v) for k, v in params.items()])

if __name__ == "__main__":
    myParams = {"server":"mpilgrim", \
                "database":"master", \
                "uid":"sa", \
                "pwd":"secret" \
                }
    print buildConnectionString(myParams)
```

Ahora ejecute este programa y vea lo que sucede.

En el IDE ActivePython para Windows puede ejecutar el programa de Python que esté editando escogiendo File->Run... (**Ctrl-R**). La salida se muestra en la pantalla interactiva.

En el IDE de Python de Mac OS puede ejecutar un programa de Python con Python->Run window... (**Cmd-R**), pero hay una opción importante que debe activar antes. Abra el fichero `.py` en el IDE, y muestre el menú de opciones pulsando en el triángulo negro en la esquina superior derecha de la ventana, asegurándose de que está marcada la opción Run as __main__. Esta preferencia está asociada a cada fichero por separado, pero sólo tendrá que marcarla una vez por cada uno.

En sistemas compatibles con UNIX (incluido Mac OS X), puede ejecutar un programa de Python desde la línea de órdenes: **`python odbchelper.py`**

La salida de `odbchelper.py` será algo así:

```
server=mpilgrim;uid=sa;database=master;pwd=secret
```

2.2. Declaración de funciones

- 2.2.1. Los tipos de Python frente a los de otros lenguajes de programación

Python tiene funciones como la mayoría de otros lenguajes, pero no dispone de ficheros de cabeceras como C++ o secciones `interface/implementation` como tiene Pascal. Cuando necesite una función, limítese a declararla, como aquí:

```
def buildConnectionString(params):
```

Fíjese en que la palabra clave `def` empieza la declaración de la función, seguida de su nombre y de los argumentos entre paréntesis. Si hay varios argumentos (no se muestra aquí) irán separados por comas.

Observe también que la función no define un tipo de retorno. Las funciones de Python no especifican el tipo de dato que retornan; ni siquiera especifican si devuelven o no un valor. En realidad, cada función de Python devuelve un valor; si la función ejecuta alguna vez una sentencia `return` devolverá ese valor, y en caso contrario devolverá `None`, el valor nulo de Python.

En Visual Basic las funciones (devuelven un valor) comienzan con `function`, y las subrutinas (no devuelven un valor) lo hacen con `sub`. En Python no tenemos subrutinas. Todo son funciones, todas las funciones devuelven un valor (incluso si es `None`) y todas las funciones comienzan por `def`.

El argumento `params` no especifica un tipo de dato. En Python nunca se indica explícitamente el tipo de las variables. Python averigua el tipo de la variable y lo almacena de forma interna.

En Java, C++ y otros lenguajes de tipo estático debe especificar el tipo de dato del valor de retorno de la función y de cada uno de sus argumentos. En Python nunca especificará de forma explícita el tipo de dato de nada.

Python lleva un registro interno del tipo de dato basándose en el valor asignado.

2.2.1. Los tipos de Python frente a los de otros lenguajes de programación

Un erudito lector me envió esta explicación de cómo se comparan los tipos de Python con otros lenguajes de programación:

Lenguajes de tipado estático
: Un lenguaje cuyos tipos se fijan en el momento de compilar. La mayoría de los lenguajes de tipado estático fuerzan esto exigiéndole que declare todas las varibles con sus tipos antes de usarlas. Java y C son lenguajes de tipado estático.

Lenguajes de tipado dinámico
: Un lenguaje cuyos tipos se descubren en tiempo de ejecución; es lo opuesto del tipado estático. VBScript y Python son de tipado dinámico, porque fijan el tipo que va a tener una variable cada vez que se le asigna un valor.

Lenguajes fuertemente tipados
: Un lenguaje cuyos tipos son estrictos. Java y Python son fuertemente tipados. Si tiene un entero, no puede tratarlo como una cadena de texto sin convertirlo explícitamente.

Lenguajes débilmente tipados
: Un lenguaje cuyos tipos pueden ser ignorados; lo opuesto a fuertemente tipados. VBScript es débilmente tipado. En VBScript puede concatenar la cadena `'12'` y el entero `3` para obtener la cadena `'123'` y entonces tratarlo como el entero `123`, todo ello sin conversiones explícitas.

De manera que Python es tanto *dinámicamente tipado* (porque no usa declaraciones explícitas de tipos de dato) como *fuertemente tipado* (porque una vez la variable adquiere un tipo, sí que importa).

2.3. Documentación de funciones

Puede documentar una función en Python proporcionando una `cadena de documentación`.

Ejemplo 2.2. Definición de la `cadena de documentación` de la función `buildConnectionString`

```
def buildConnectionString(params):
    """Crea una cadena de conexión partiendo de un diccionario de
parámetros.

    Devuelve una cadena."""
```

Las comillas triples implican una cadena multilínea. Todo lo que haya entre el principio y el final de las comillas es parte de una sola cadena, incluyendo los retornos de carro y otros comillas. Puede usarlas para definir cualquier cadena, pero donde las verá más a menudo es haciendo de `cadena de documentación`.

Las comillas triples también son una manera sencilla de definir una cadena que contenga comillas tanto simples como dobles, como `qq/.../` en Perl.

Todo lo que hay entre las comillas triples es la `cadena de documentación` de la función, y se usa para explicar lo que hace la función. En caso de que exista una `cadena de documentación`, debe ser la primera cosa definida en una función (esto es, lo primero tras los dos puntos). Técnicamente, no necesita dotar a su función de una `cadena de documentación`, pero debería hacerlo siempre. Sé que habrá escuchado esto en toda clase de programación a la que haya asistido alguna vez, pero Python le da un incentivo añadido: la `cadena de documentación` está disponible en tiempo de ejecución como atributo de la función.

Muchos IDE de Python utilizan la `cadena de documentación` para proporcionar una ayuda sensible al contexto, de manera que cuando escriba el nombre de una función aparezca su `cadena de documentación` como ayuda. Esto puede ser increíblemente útil, pero lo será tanto como buenas las `cadenas de documentación` que usted escriba.

Lecturas complementarias sobre las funciones de documentación

- PEP 257 define las conveciones al respecto de las `cadenas de documentación`.
- La *Guía de estilo de Python* indica la manera de escribir una buena `cadena de documentación`.
- El *Tutorial de Python* expone convenciones para el espaciado dentro de las `cadenas de documentación`.

2.4. Todo es un objeto

En caso de que no se haya dado cuenta, acabo de decir que las funciones de Python tienen atributos y que dispone de esos atributos en tiempo de ejecución.

Una función es un objeto, igual que todo lo demás en Python.

Abra su IDE favorito para Python y escriba:

Ejemplo 2.3. Acceso a la `cadena de documentación` de la función `buildConnectionString`

```
>>> import odbchelper                                    ❶
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa",
"pwd":"secret"}
>>> print odbchelper.buildConnectionString(params)       ❷
```

```
server=mpilgrim;uid=sa;database=master;pwd=secret
>>> print odbchelper.buildConnectionString.__doc__ ❸
Crea una cadena de conexión partiendo de un diccionario de parámetros.

Devuelve una cadena.
```

❶ La primera línea importa el programa `odbchelper` como módulo (un trozo de código que puede usar interactivamente, o desde un programa de Python mayor. -- Verá ejemplos de programas de Python multimódulo en el Capítulo 4). Una vez importe un módulo podrá referirse a cualquiera de sus funciones, clases o atributos públicos. Esto puede hacerlo un módulo para acceder a funcionalidad existente en otros módulos, y puede hacerlo también usted en el IDE. Esto es un concepto importante y hablaremos de ello más adelante.

❷ Cuando quiera usar funciones definidas en módulos importados, deberá incluir el nombre del módulo. De manera que no puede simplemente decir `buildConnectionString`; debe ser `odbchelper.buildConnectionString`. Si ha usado clases en Java, esto debería serle vagamente familiar.

❸ En lugar de llamar a la función tal como cabría esperar, ha solicitado uno de los atributos de la función, `__doc__`.

`import` en Python es como `require` en Perl. Una vez que hace `import` sobre un módulo de Python, puede acceder a sus funciones con *`módulo.función`*; una vez que hace `require` sobre un módulo de Perl, puede acceder a sus funciones con *`módulo::función`*.

2.4.1. La ruta de búsqueda de import

Antes de continuar, quiero mencionar brevemente la ruta de búsqueda de bibliotecas. Python busca en varios sitios cuando intenta importar un módulo. Específicamente, busca en todos los directorios definidos en `sys.path`. Esto es simplemente una lista, y puede verla o modificarla fácilmente con los métodos estándar de una lista (conocerá las listas más adelante en este capítulo).

Ejemplo 2.4. Ruta de búsqueda de import

```
>>> import sys                    ❶
>>> sys.path                      ❷
['', '/usr/local/lib/python2.2', '/usr/local/lib/python2.2/plat-
linux2',
'/usr/local/lib/python2.2/lib-dynload',
'/usr/local/lib/python2.2/site-packages',
'/usr/local/lib/python2.2/site-packages/PIL',
'/usr/local/lib/python2.2/site-packages/piddle']
>>> sys                           ❸
<module 'sys' (built-in)>
>>> sys.path.append('/mi/nueva/ruta') ❹
```

❶ Importar el módulo `sys` deja disponibles todas sus funciones y atributos.

❷ `sys.path` es una lista de nombres de directorios que constituye la ruta de búsqueda actual (la suya puede ser diferente, dependiendo del sistema operativo, qué versión de Python esté ejecutando, y dónde la instaló originalmente). Python buscará en estos directorios (y en ese orden) un fichero `.py` que corresponda al nombre del módulo que intenta importar.

❸ En realidad mentí; la verdad es más complicada que eso, porque no todos los módulos se instalan como ficheros `.py`. Algunos, como el módulo `sys`, son «módulos incorporados» ("*built-in*"); y están dentro del propio Python. Los módulos *built-in* se comportan exactamente como los normales pero no dispone de su código fuente en Python, ¡porque no se escriben usando Python! (el módulo `sys` está escrito en C.)

❹ Puede añadir un nuevo directorio a la ruta de búsqueda de Python en tiempo de ejecución agregando el nombre del directorio a `sys.path`, y entonces Python buscará en ese directorio también cada vez que intente importar un módulo. El efecto dura tanto como esté en ejecución Python. (Hablaremos más sobre `append` y otros métodos de listas en el Capítulo 3)

2.4.2. ¿Qué es un objeto?

Todo en Python es un objeto, y casi todo tiene atributos y métodos. Todas las funciones tienen un atributo `__doc__` que devuelve la `cadena de documentación`

definida en su código fuente. El módulo `sys` es un objeto que contiene (entre otras cosas) un atributo llamado `path`. Y así con todo.

Aún así, la pregunta sigue sin contestar. ¿Qué es un objeto? Los diferentes lenguajes de programación definen "objeto" de maneras diferentes. En algunos significa que *todos* los objetos *deben* tener atributos y métodos; en otros esto significa que todos los objetos pueden tener subclases. En Python la definición es más amplia: algunos objetos no tienen ni atributos ni métodos (más sobre esto en el Capítulo 3), y no todos los objetos pueden tener subclases (más al respecto en el Capítulo 5). Pero todo es un objeto en el sentido de que se puede asignar a una variable o ser pasado como argumento a una función (más sobre en el Capítulo 4).

Esto es tan importante que voy a repetirlo en caso de que se lo haya perdido las últimas veces: *todo en Python es un objeto*. Las cadenas son objetos. Las listas son objetos. Las funciones son objetos. Incluso los módulos son objetos.

Lecturas complementarias sobre objetos

- La *Referencia del lenguaje Python* explica exactamente lo que quiere decir que todo en Python es un objeto, porque algunas personas son pedantes y les gusta discutir este tipo de cosas hasta la muerte.
- eff-bot hace un resumen sobre los objetos en Python.

2.5. Sangrado (indentado) de código

Las funciones de Python no tienen `begin` o `end` explícitos, ni llaves que marquen dónde empieza o termina su código. El único delimitador son dos puntos (`:`) y el sangrado del propio código.

Ejemplo 2.5. Sangrar la función `buildConnectionString`

```
def buildConnectionString(params):
```

```
    """Crea una cadena de conexión partiendo de un diccionario de
parámetros.

    Devuelve una cadena."""
    return ";".join(["%s=%s" % (k, v) for k, v in params.items()])
```

Los bloques de código van definidos por su sangrado. Con «bloque de código» quiero decir funciones, sentencias `if`, bucles `for`, `while`, etc. El sangrado comienza un bloque y su ausencia lo termina. No hay llaves, corchetes ni palabras clave explícitas. Esto quiere decir que el espacio en blanco es significativo y debe ser consistente. En este ejemplo el código de la función (incluida la `cadena de documentación`) está sangrado a cuatro espacios. No tienen por qué ser cuatro, el único requisito es que sea consistente. La primera línea que no esté sangrada queda ya fuera de la función.

Ejemplo 2.6, "Sentencias if" muestra un ejemplo de sangrado de código con sentencias `if`

Ejemplo 2.6. Sentencias `if`

```
def fib(n):                     ❶
    print 'n =', n              ❷
    if n > 1:                   ❸
        return n * fib(n - 1)
    else:                       ❹
        print 'fin de la línea'
        return 1
```

❶ Esta función llamada `fib` toma un argumento, `n`. Todo el código dentro de la función está sangrado.

❷ Imprimir en la pantalla es muy fácil en Python, basta usar `print`. Las sentencias `print` pueden tomar cualquier tipo de dato, incluyendo cadenas de texto, enteros, y otros tipos nativos como diccionarios y listas de los que oirá hablar en el siguiente capítulo. Puede incluso mezclarlos e imprimir varias cosas en una sola línea usando una lista de valores separados por comas. Cada valor se imprime en la misma línea, separado por espacios (las

comas no se imprimen). Así que cuando se llama a `fib` con `5`, imprime "n = 5".

❸ Las sentencias `if` son un tipo de bloque de código. Si la expresión de `if` se evalúa a un valor "verdadero" se ejecuta el código sangrado, y n caso contrario se ejecuta el bloque `else`.

❹ Por supuesto, los bloques `if` y `else` pueden contener varias líneas siempre que mantengan un sangrado consistente. Este bloque `else` tiene dos líneas de código dentro. No hay una sintaxis especial para bloques de código de varias líneas. Simplemente indéntelas y siga con su vida.

Tras protestar bastante al principio y hacer unas cuántas analogías despectivas con Fortran, llegará a reconciliarse con esto y empezará a ver los beneficios. Uno de los más significativos es que todos los programas en Python tienen un aspecto similar, ya que el sangrado es no es una cuestión de estilo sino un requisito del lenguaje. Esto hace más sencilla la lectura y comprensión de código en Python escrito por otras personas.

Python utiliza retornos de carro para separar sentencias y los dos puntos y el sangrado para reconocer bloques de código. C++ y Java usan puntos y coma para separar sentencias, y llaves para indicar bloques de código.

Lecturas complementarias sobre el sangrado de código

- La *Referencia del lenguaje Python* comenta problemas de sangrado entre plataformas y muestra varios errores de identación.
- La *Guía de estilo de Python* comenta buenos estilos de sangrado.

2.6. Prueba de módulos

Los módulos de Python son objetos y tienen varios atributos útiles. Puede usar este hecho para probar sus módulos de forma sencilla a medida que los escribe. Aquí tiene un ejemplo que usa el truco de `if __name__`.

```
if __name__ == "__main__":
```

Algunas observaciones antes de que empiece lo bueno. Primero, no se necesitan paréntesis que encierren la expresión de `if`. Segundo, la sentencia `if` termina con dos puntos, y va seguida por código sangrado.

Al igual que C, Python utiliza == para la comparación y = para la asignación. Al contrario que C, Python no permite la asignación embebida, de manera que no existe la posibilidad de asignar un valor accidentalmente donde deseaba hacer una comparación.

De manera que... ¿por qué es un truco esta sentencia `if` en particular? Los módulos son objetos y todos los módulos tienen un atributo llamado `__name__`. El valor del `__name__` de un módulo depende de cómo esté usándolo. Si `importa` el módulo, entonces `__name__` es el nombre del fichero del módulo, sin el directorio de la ruta ni la extensión del fichero. Pero también puede ejecutar el módulo directamente como si fuera un programa, en cuyo caso `__name__` tendrá un valor especial predefinido, `__main__`.

```
>>> import odbchelper
>>> odbchelper.__name__
'odbchelper'
```

Sabiendo esto, puede diseñar una batería de pruebas para su módulo dentro del propio módulo situándola dentro de esta sentencia `if`. Cuando ejecuta el módulo directamente, `__name__` es `__main__`, de manera que se ejecutan las pruebas. Cuando importa el módulo, `__name__` es otra cosa, de manera que se ignoran las pruebas. Esto hace más sencillo desarrollar y depurar nuevos módulos antes de integrarlos en un programa mayor.

En MacPython, hay que dar un paso adicional para hacer que funcione el truco `if __name__`. Muestre el menú de opciones pulsando el triángulo

negro en la esquina superior derecha de la ventana, y asegúrese de que está marcado Run as __main__.

Lecturas complementarias sobre importar módulos

- La *Referencia del lenguaje Python* comenta los detalles de bajo nivel de la importación de módulos.

Capítulo 3. Tipos de dato nativos

Volveremos a su primer programa en Python en un minuto. Pero antes, se impone una pequeña digresión, porque necesita saber cosas sobre los diccionarios, las tuplas y las listas (¡oh, dios!). Si es un hacker de Perl, probablemente puede saltarse los comentarios sobre diccionarios y listas, pero debería prestar atención a las tuplas.

3.1. Presentación de los diccionarios

Uno de los tipos incorporados de Python es el diccionario, que define relaciones uno a uno entre claves y valores.

Un diccionario en Python es como un hash en Perl. En Perl, las variables que almacenan hashes siempre empiezan con un carácter `%`. En Python las variables se pueden llamar de cualquier manera, y Python sabe su tipo internamente.

Un diccionario en Python es como una instancia de la clase `Hashtable` de Java.

Un diccionario en Python es como una instancia del objeto `Scripting.Dictionary` de Visual Basic.

3.1.1. Definir diccionarios

Ejemplo 3.1. Definición de un diccionario

```
>>> d = {"server":"mpilgrim", "database":"master"} ❶
>>> d
{'server': 'mpilgrim', 'database': 'master'}
>>> d["server"]                                    ❷
'mpilgrim'
>>> d["database"]                                  ❸
'master'
>>> d["mpilgrim"]                                  ❹
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
KeyError: mpilgrim
```

❶ Primero creamos un nuevo diccionario con dos elementos y lo asignamos a la variable `d`. Cada elemento es un par clave-valor, y el conjunto de los elementos se encierra entre llaves.

❷ `'server'` es una clave, y su valor asociado, referenciado por `d["server"]`, es `'mpilgrim'`.

❸ `'database'` es una clave, y su valor asociado, referenciado por `d["database"]`, es `'master'`.

❹ Puede obtener los valores por su clave pero no las claves por su valor. De manera que `d["server"]` es `'mpilgrim'`, pero `d["mpilgrim"]` genera una excepción, porque `'mpilgrim'` no es una clave.

3.1.2. Modificar diccionarios

Ejemplo 3.2. Modificación de un diccionario

```
>>> d
{'server': 'mpilgrim', 'database': 'master'}
>>> d["database"] = "pubs" ❶
>>> d
{'server': 'mpilgrim', 'database': 'pubs'}
>>> d["uid"] = "sa"        ❷
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'pubs'}
```

❶ No puede tener claves duplicadas en un diccionario. Asignar un valor nuevo a una clave existente elimina el valor antiguo.

❷ Puede añadir pares clave-valor en cualquier momento. Esta sintaxis es idéntica a la usada para modificar valores existentes (sí, esto le irritará algún día cuando piense que está añadiendo valores nuevos pero en realidad sólo está modificando el mismo valor una y otra vez porque la clave no está cambiando como usted piensa.)

Advierta que el nuevo elemento (clave `'uid'`, valor `'sa'`) aparece en el medio. En realidad es sólo coincidencia que los elementos apareciesen en orden en el primer ejemplo; también es coincidencia que ahora aparezcan en desorden.

Los diccionarios no tienen concepto de orden entre sus elementos. Es incorrecto decir que los elementos están “desordenados”, ya que simplemente no tienen orden. Esto es una distinción importante que le

irritará cuando intente acceder a los elementos de un diccionario en un orden específico y repetible (por ejemplo, en orden alfabético por clave). Hay maneras de hacer esto, pero no vienen de serie en el diccionario.

Cuando trabaje con diccionarios, ha de tener en cuenta que las claves diferencian entre mayúsculas y minúsculas.[1]

Ejemplo 3.3. Las claves de los diccionarios distinguen las mayúsculas

```
>>> d = {}
>>> d["clave"] = "valor"
>>> d["clave"] = "otro valor" ❶
>>> d
{'clave': 'otro valor'}
>>> d["Clave"] = "tercer valor" ❷
>>> d
{'Clave': 'tercer valor', 'clave': 'otro valor'}
```

❶ Asignar un valor a una clave existente en un diccionario simplemente reemplaza el valor antiguo por el nuevo.

❷ Esto no asigna un valor a una clave existente, porque las cadenas en Python distinguen las mayúsculas, de manera que `'clave'` no es lo mismo que `'Clave'`. Esto crea un nuevo par clave/valor en el diccionario; puede parecerle similar, pero en lo que concierne a Python es completamente diferente.

Ejemplo 3.4. Mezcla de tipos de dato en un diccionario

```
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'pubs'}
>>> d["retrycount"] = 3 ❶
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master',
'retrycount': 3}
>>> d[42] = "douglas"   ❷
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master',
```

```
42: 'douglas', 'retrycount': 3}
```

❶ Los diccionarios no son sólo para las cadenas. Los valores de los diccionarios pueden ser cualquier tipo de dato, incluyendo cadenas, enteros, objetos o incluso otros diccionarios. Y dentro de un mismo diccionario los valores no tienen por qué ser del mismo tipo: puede mezclarlos según necesite.

❷ Las claves de los diccionarios están más restringidas, pero pueden ser cadenas, enteros y unos pocos tipos más. También puede mezclar los tipos de claves dentro de un diccionario.

3.1.3. Borrar elementos de diccionarios

Ejemplo 3.5. Borrar elementos de un diccionario

```
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master',
42: 'douglas', 'retrycount': 3}
>>> del d[42] ❶
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master',
'retrycount': 3}
>>> d.clear() ❷
>>> d
{}
```

❶ `del` le permite borrar elementos individuales de un diccionario por su clave.

❷ `clear` elimina todos los elementos de un diccionario. Observe que unas llaves vacías indica un diccionario sin elementos.

Lecturas complementarias sobre diccionarios

- *How to Think Like a Computer Scientist* le instruye sobre los diccionarios y le muestra cómo usarlos para modelar matrices dispersas.
- La Python Knowledge Base tiene muchos ejemplos de código que usan diccionarios.
- El Python Cookbook comenta cómo ordenar los valores de un diccionario por clave.

- La *Referencia de bibliotecas de Python* lista todos los métodos de los diccionarios.

Footnotes

[1] N. del T.: En inglés se dice que es *case sensible*, que en castellano se traduce como *sensible a la caja*. "Caja", en el ámbito de la tipografía, se usa tradicionalmente para definir si una letra es mayúscula (caja alta) o minúscula (caja baja), en referencia al lugar en que se almacenaban los tipos móviles en las imprentas antiguas. En el resto del libro diré simplemente que se "distinguen las mayúsculas" (o no)

3.2. Presentación de las listas

Las listas son el caballo de tiro de Python. Si su única experiencia con listas son los array de Visual Basic o (dios no lo quiera) los datastore de Powerbuilder, prepárese para las listas de Python.

Una lista de Python es como un array en Perl. En Perl, las variables que almacenan arrays siempre empiezan con el carácter `@`; en Python, las variables se pueden llamar de cualquier manera, y Python se ocupa de saber el tipo que tienen.

Una lista en Python es mucho más que un array en Java (aunque puede usarse como uno si es realmente eso todo lo que quiere en esta vida). Una mejor analogía podría ser la clase `ArrayList`, que puede contener objetos

arbitrarios y expandirse de forma dinámica según se añaden otros nuevos.

3.2.1. Definir listas

Ejemplo 3.6. Definición de una lista

```
>>> li = ["a", "b", "mpilgrim", "z", "example"] ❶
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[0]                                        ❷
'a'
>>> li[4]                                        ❸
'example'
```

❶ Primero definimos una lista de cinco elementos. Observe que mantienen su orden original. Esto no es un accidente. Una lista es un conjunto ordenado de elementos encerrados entre corchetes.

❷ Una lista se puede usar igual que un array basado en cero. El primer elemento de cualquier lista que no esté vacía es siempre `li[0]`.

❸ El último elemento de esta lista de cinco elementos es `li[4]`, porque las listas siempre empiezan en cero.

Ejemplo 3.7. Indices negativos en las listas

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[-1] ❶
'example'
>>> li[-3] ❷
'mpilgrim'
```

❶ Un índice negativo accede a los elementos desde el final de la lista contando hacia atrás. El último elemento de cualquier lista que no esté vacía es siempre `li[-1]`.

❷ Si el índice negativo le confunde, piense de esta manera: `li[-n] == li[len(li) - n]`. De manera que en esta lista, `li[-3] == li[5 - 3] == li[2]`.

Ejemplo 3.8. *Slicing* de una lista

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[1:3]  ❶
['b', 'mpilgrim']
>>> li[1:-1] ❷
['b', 'mpilgrim', 'z']
>>> li[0:3]  ❸
['a', 'b', 'mpilgrim']
```

❶ Puede obtener un subconjunto de una lista, llamado "*slice*"[2], especificando dos índices. El valor de retorno es una nueva lista que contiene todos los elementos de la primera lista, en orden, comenzando por el primer índice del *slice* (en este caso `li[1]`), hasta el segundo índice sin incluirlo (en este caso `li[3]`).

❷ El particionado (*slicing*) funciona si uno de los dos índices o ambos son negativos. Si le ayuda, puede pensarlo de esta manera: leyendo la lista de izquierda a derecha, el primer índice especifica el primer elemento que quiere, y el segundo especifica el primer elemento que no quiere. El valor de retorno es todo lo que hay en medio.

❸ Las listas empiezan en cero, así que `li[0:3]` devuelve los tres primeros elementos de la lista, empezando en `li[0]`, y hasta `li[3]`, pero sin incluirlo.

Ejemplo 3.9. Atajos para particionar

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[:3] ❶
['a', 'b', 'mpilgrim']
>>> li[3:] ❷ ❸
['z', 'example']
>>> li[:]  ❹
['a', 'b', 'mpilgrim', 'z', 'example']
```

❶ Si el índice izquierdo es 0, puede no ponerlo, y el 0 queda implícito. De manera que `li[:3]` es lo mismo que el `li[0:3]` del Ejemplo 3.8, "Slicing de

una lista".

❷ De forma similar, si el índice de la derecha es la longitud de la lista, puede eliminarlo. Así que `li[3:]` es lo mismo que `li[3:5]`, porque esta lista tiene cinco elementos.

❸ Advierta la simetría. En esta lista de cinco elementos, `li[:3]` devuelve los 3 primeros elementos, y `li[3:]` devuelve los dos últimos. En realidad, `li[:n]` siempre devolverá los primeros `n` elementos, y `li[n:]` devolverá el resto, independientemente del tamaño de la lista.

❹ Si se omiten ambos índices se incluyen todos los elementos de la lista. Pero no es la misma que la lista original `li`; es una nueva lista que tiene todos los mismos elementos. `li[:]` es un atajo para hacer una copia completa de una lista.

3.2.2. Añadir de elementos a listas

Ejemplo 3.10. Adición de elementos a una lista

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li.append("new")                    ❶
>>> li
['a', 'b', 'mpilgrim', 'z', 'example', 'new']
>>> li.insert(2, "new")                 ❷
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new']
>>> li.extend(["two", "elements"])      ❸
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new', 'two',
'elements']
```

❶ `append` añade un único elemento al final de la lista.

❷ `insert` inserta un único elemento en una lista. El argumento numérico es el índice del primer elemento que cambia de posición. Observe que los elementos de la lista no tienen por qué ser únicos; ahora hay dos elementos con el valor `'new'`, `li[2]` y `li[6]`.

❸ `extend` concatena listas. Verá que no se llama a `extend` con varios argumentos; se le llama con uno, una lista. En este caso, esa lista tiene dos elementos.

Ejemplo 3.11. La diferencia entre `extend` y `append`

```
>>> li = ['a', 'b', 'c']
>>> li.extend(['d', 'e', 'f']) ❶
>>> li
['a', 'b', 'c', 'd', 'e', 'f']
>>> len(li)                    ❷
6
>>> li[-1]
'f'
>>> li = ['a', 'b', 'c']
>>> li.append(['d', 'e', 'f']) ❸
>>> li
['a', 'b', 'c', ['d', 'e', 'f']]
>>> len(li)                    ❹
4
>>> li[-1]
['d', 'e', 'f']
```

❶ Las listas tienen dos métodos, `extend` y `append`, que parecen hacer lo mismo, pero en realidad son completamente diferentes. `extend` toma un único argumento, que es siempre una lista, y añade cada uno de los elementos de esa lista a la original.

❷ Aquí empezamoscon una lista de tres elementos (`'a'`, `'b'`, y `'c'`), y la extendemos con una lista de otros tres elementos (`'d'`, `'e'`, y `'f'`), de manera que ahora tenemos una de seis.

❸ Por otro lado, `append` toma un argumento, que puede ser cualquier tipo de dato, y simplemente lo añade al final de la lista. Aquí, estamos llamado al método `append` con un único argumento, que es una lista de tres elementos.

❹ Ahora la lista original, que empezó siendo una lista de tres elementos, contiene cuatro. ¿Por qué cuatro? Porque el último elemento que acabamos de añadir *es una lista*. Las listas contienen cualquier tipo de dato, incluyendo

otras listas. Puede que esto es lo que usted quiere, puede que no. No use `append` si lo que quiere hacer es `extend`.

3.2.3. Buscar en listas

Ejemplo 3.12. Búsqueda en una lista

```
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new', 'two',
'elements']
>>> li.index("example") ❶
5
>>> li.index("new")     ❷
2
>>> li.index("c")       ❸
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: list.index(x): x not in list
>>> "c" in li           ❹
False
```

❶ `index` encuentra la primera aparición de un valor en la lista y devuelve su índice.

❷ `index` encuentra la *primera* aparición de un valor en la lista. En este caso, `'new'` aparece dos veces en la lista, en `li[2]` y `li[6]`, pero `index` devolverá sólo el primer índice, `2`.

❸ Si el valor no se encuentra en la lista, Python lanza una excepción. Esto es notablemente diferente a la mayoría de los lenguajes, que devolverán algún índice inválido. Aunque pueda parecer irritante, es bueno, porque significa que el programa terminará con error al hallar la fuente del problema, en lugar de más adelante cuando intente usar el índice no válido.

❹ Para probar si un valor está en la lista, utilice `in`, que devuelve `True` si el valor existe o `False` si no.

Antes de la versión 2.2.1, Python no tenía un tipo booleano. Para

compensarlo, Python aceptaba casi cualquier cosa en un contexto booleano (como una sentencia `if`), de acuerdo a las siguientes reglas:

- `0` es falso; el resto de los números son verdaderos.
- Una cadena vacía (`""`) es falso, cualquier otra cadena es verdadera.
- Una lista vacía (`[]`) es falso; el resto de las listas son verdaderas.
- Una tupla vacía (`()`) es falso; el resto de las tuplas son verdaderas.
- Un diccionario vacío (`{}`) es falso; todos los otros diccionarios son verdaderos.

Estas reglas siguen aplicándose en Python 2.2.1 y siguientes, pero ahora además puedes usar un verdadero booleano, que tiene el valor de `True` o `False`. Tenga en cuenta las mayúsculas; estos valores, como todo lo demás en Python, las distinguen.

3.2.4. Borrar elementos de listas

Ejemplo 3.13. Borrado de elementos de una lista

```
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new', 'two',
'elements']
>>> li.remove("z")   ❶
>>> li
['a', 'b', 'new', 'mpilgrim', 'example', 'new', 'two', 'elements']
>>> li.remove("new") ❷
>>> li
['a', 'b', 'mpilgrim', 'example', 'new', 'two', 'elements']
>>> li.remove("c")   ❸
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: list.remove(x): x not in list
>>> li.pop()         ❹
'elements'
>>> li
['a', 'b', 'mpilgrim', 'example', 'new', 'two']
```

❶ `remove` elimina la primera aparición de un valor en una lista.

❷ `remove` elimina *sólo* la primera aparición de un valor. En este caso, `'new'` aparecía dos veces en la lista, pero `li.remove("new")` sólo eliminó la primera aparición.

❸ Si el valor no se encuentra en la lista, Python lanza una excepción. Esto semeja el comportamiento del método `index`.

❹ `pop` es una bestia interesante. Hace dos cosas: elimina el último elemento de la lista, y devuelve el valor que borró. Observará que esto es diferente de `li[-1]`, que devuelve un valor pero no cambia la lista, y de `li.remove(`*`valor`*`)`, que cambia la lista pero no devuelve un valor.

3.2.5. Uso de operadores de lista

Ejemplo 3.14. Operadores de lista

```
>>> li = ['a', 'b', 'mpilgrim']
>>> li = li + ['example', 'new'] ❶
>>> li
['a', 'b', 'mpilgrim', 'example', 'new']
>>> li += ['two']                ❷
>>> li
['a', 'b', 'mpilgrim', 'example', 'new', 'two']
>>> li = [1, 2] * 3              ❸
>>> li
[1, 2, 1, 2, 1, 2]
```

❶ Las listas también se pueden concatenar con el operador `+`. *`lista`* `= ` *`lista`* `+ ` *`otralista`* da el mismo resultado que *`lista`*`.extend(`*`otralista`*`)`. Pero el operador `+` devuelve una nueva lista (concatenada) como valor, mientras que `extend` sólo altera una existente. Esto significa que `extend` es más rápido, especialmente para listas grandes.

❷ Python admite el operador `+=`. `li += ['two']` es equivalente a `li.extend(['two'])`. El operador `+=` funciona con listas, cadenas y enteros, y también puede sobrecargarse para trabajar con clases definidas por el usuario (más sobre clases en Capítulo 5.)

❸ El operador `*` funciona en las listas como repetidor. `li = [1, 2] * 3` es el

equivalente a `li = [1, 2] + [1, 2] + [1, 2]`, que concatena las tres listas en una.

Lecturas complementarias sobre listas

- *How to Think Like a Computer Scientist* le instruye sobre listas y señala algo importante al respecto de pasar listas como argumentos a funciones.
- El *Tutorial de Python* muestra cómo usar listas como pilas y colas.
- La Python Knowledge Base contesta preguntas frecuentes sobre listas y tiene un montón de código de ejemplo que usa listas.
- La *Referencia de bibliotecas de Python* enumera todos los métodos de las listas.

Footnotes

[2] N. del T.: porción, partición

3.3. Presentación de las tuplas

Una tupla es una lista inmutable. Una tupla no puede cambiar de ninguna manera una vez creada.

Ejemplo 3.15. Definir una tupla

```
>>> t = ("a", "b", "mpilgrim", "z", "example") ❶
>>> t
('a', 'b', 'mpilgrim', 'z', 'example')
>>> t[0]                                       ❷
'a'
>>> t[-1]                                      ❸
'example'
>>> t[1:3]                                     ❹
('b', 'mpilgrim')
```

❶ Una tupla se define de la misma manera que una lista, excepto que el conjunto de elementos se encierra entre paréntesis en lugar de corchetes.

❷ Los elementos de una tupla tienen un orden definido, como una lista. Los índices de las tuplas comienzan en cero, igual que una lista, de manera que el primer elemento de una tupla que no esté vacía siempre es `t[0]`.

❸ Los índices negativos cuentan desde el final de la tupla, just como en una lista.

❹ También funciona el *slicing*. Observe que cuando trocea una lista, obtiene una nueva lista; cuando trocea una tupla, obtiene una nueva tupla.

Ejemplo 3.16. La tuplas no tienen métodos

```
>>> t
('a', 'b', 'mpilgrim', 'z', 'example')
>>> t.append("new")    ❶
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'append'
>>> t.remove("z")      ❷
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'remove'
>>> t.index("example") ❸
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'index'
>>> "z" in t           ❹
True
```

❶ No puede añadir métodos a una tupla. Las tuplas no tienen métodos `append` o `extend`.

❷ No puede eliminar elementos de una tupla. Las tuplas no tienen métodos `remove` o `pop`.

❸ No puede buscar elementos en una tupla. Las tuplas no tienen método `index`.

❹ Sin embargo, puede usar `in` para ver comprobar si existe un elemento en la tupla.

¿Para qué sirven entonces las tuplas?

- Las tuplas son más rápidas que las listas. Si define un conjunto constante de valores y todo lo que va a hacer es iterar sobre ellos, use una tupla en lugar de una lista.
- "Proteger contra escritura" los datos que no necesita cambiar hace el código más seguro. Usar una tupla en lugar de una ristra es como tener una sentencia `assert` implícita mostrando que estos datos son constantes, y que se necesita una acción consciente (y una función específica) para cambiar eso.
- ¿Recuerda que dije que las claves de diccionario pueden ser enteros, cadenas y "unos pocos otros tipos"? Las tuplas son uno de estos tipos. Se pueden usar como claves en un diccionario, pero las listas no.En realidad, es más complicado que esto. Las claves de diccionario deben ser inmutables. Las tuplas son inmutables, pero si tiene una tupla de listas, eso cuenta como mutable, y no es seguro usarla como clave de un diccionario. Sólo las tuplas de cadenas, números y otras tuplas seguras para los diccionarios pueden usarse como claves.
- Se pueden usar tuplas para dar formato a cadenas, como veremos en breve.

Las tuplas se pueden convertir en listas, y viceversa. La función incorporada `tuple` toma una lista y devuelve una tupla con los mismos elementos, y la función `list` toma una tupla y devuelve una lista. En efecto, `tuple` "congela" una lista, y `list` "descongela" una `tuple`.

Más sobre tuplas

- *How to Think Like a Computer Scientist* instruye sobre las tuplas y muestra cómo concatenarlas.
- La Python Knowledge Base muestra cómo ordenar una tupla.
- El *Tutorial de Python* muestra cómo definir una tupla con un elemento.

3.4. Declaración de variables

- 3.4.1. Referencia a variables
- 3.4.2. Asignar varios valores a la vez

Ahora que ya sabe algo sobre diccionarios, tuplas y listas (¡oh dios mío!), volvamos al programa de ejemplo de Capítulo 2, `odbchelper.py`.

Python tiene variables locales y globales como casi todo el resto de lenguajes, pero no tiene declaración explícitade variables. Las variables cobran existencia al asignársele un valor, y se destruyen automáticamente al salir de su ámbito.

Ejemplo 3.17. Definición de la variable `myParams`

```
if __name__ == "__main__":
    myParams = {"server":"mpilgrim", \
                "database":"master", \
                "uid":"sa", \
                "pwd":"secret" \
                }
```

Advierta el sangrado. Una sentencia `if` es un bloque de código y necesita ser sangrado igual que una función.

Observe también que la asignación de la variable es una orden en varias líneas, donde la barra inversa (“\”) sirve como marcador de continuación de línea.

Cuando una orden se divide en varias líneas con la marca de continuación de línea (“\”), las siguientes líneas se pueden sangrar de cualquier manera; la habitual sangrado astringente de Python no se aplica aquí. Si su IDE de Python autosangra la línea a continuación, probablemente debería aceptar este comportamiento a menos que tenga una imperiosa razón para no hacerlo.

Estrictamente hablando, las expresiones entre paréntesis, corchetes y llaves (como la definición de un diccionario) también pueden ocupar varias líneas con

o sin el carácter de continuación (“\”). Me gustaría incluir la barra inversa aunque no haga falta debido a que pienso que hace el código más sencillo de leer, pero esto es cuestión de estilo.

Tercero, nunca llegamos a declarar la variable `myParams`, simplemente le asignamos un valor. Esto es como en VBScript sin la opción `option explicit`. Por suerte, al contrario que VBScript, Python no le permite hacer referencia a una variable a la que nunca se asignó un valor; intentar esto lanzará una excepción.

3.4.1. Referencia a variables

Ejemplo 3.18. Referencia a una variable sin asignar

```
>>> x
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
NameError: There is no variable named 'x'
>>> x = 1
>>> x
1
```

Le agradecerá esto a Python algún día.

3.4.2. Asignar varios valores a la vez

Uno de los atajos más molones de Python es el uso de secuencias para asignar múltiples valores a la vez.

Ejemplo 3.19. Asignación de múltiples valores simultáneamente

```
>>> v = ('a', 'b', 'e')
>>> (x, y, z) = v     ❶
>>> x
'a'
>>> y
'b'
```

```
>>> z
'e'
```

❶ `v` es una tupla de tres elementos, y `(x, y, z)` es una tupla de tres variables. Asignar una a la otra provoca que cada uno de los valores de `v` se asigne a las variables correspondientes, en orden.

Esto tiene todo tipo de usos. A menudo quiero asignar nombres a un rango de valores. En C usaríamos `enum` y listaríamos de forma manual cada constante y sus valores asociados, lo que parece especialmente tedioso cuando los valores son consecutivos. En Python, podemos usar la función incorporada `range` con la asignación a múltiples variables para asignar valores consecutivos rápidamente.

Ejemplo 3.20. Asignación de valores consecutivos

```
>>> range(7)
❶
[0, 1, 2, 3, 4, 5, 6]
>>> (LUNES, MARTES, MIERCOLES, JUEVES, VIENRES, SABADO, DOMINGO) =
range(7) ❷
>>> LUNES
❸
0
>>> MARTES
1
>>> DOMINGO
6
```

❶ La función incorporada `range` devuelve una lista de enteros. En su forma más sencilla, toma un límite superior y devuelve una lista que empieza en cero hasta el límite superior, sin incluirlo (si lo desea, puede psar otros para especificar una base diferente de `0` y un paso diferente a `1`. Puede hacer `print range.__doc__` si quiere más detalles)

❷ `LUNES`, `MARTES`, `MIERCOLES`, `JUEVES`, `VIERNES`, `SABADO`, y `DOMINGO` son las variables que estamos definiendo. (Este ejemplo viene del módulo `calendar` un módulo pequeño y simpático que imprime calendarios, como el programa `cal` de UNIX. El módulo de calendario define constntes enteras para los días

de la semana).

❸ Ahora cada variable tiene su valor: `LUNES` es `0`, `MARTES` es `1`, etc.

También puede usar la asignación multivariable para construir funciones que devuelvan varios valores, simplemente retornando una tupla con todos los valores. Quien llama puede tratarlo el valor de vuelto como una tupla, o asignar los valores a variables individuales. Muchas bibliotecas estándar de Python hacen esto, incluido el módulo `os`, que comentaremos en Capítulo 6.

Lecturas complementarias sobre variables

- La *Referencia del lenguaje Python* muestra ejemplos de cuándo puede pasar sin el carácter de continuación de línea y cuándo necesita usarlo.
- *How to Think Like a Computer Scientist* le muestra cómo usar la asignación multivariable para intercambiar los valores de dos variables.

3.5. Formato de cadenas

Python admite dar formato a valores dentro de cadenas. Aunque esto puede incluir expresiones muy complicadas, el uso más básico es insertar valores dentro de una cadena con el sustituto `%s`.

El formato de cadenas en Python usa la misma sintaxis que la función `sprintf` en C.

Ejemplo 3.21. Presentación del formato de cadenas

```
>>> k = "uid"
>>> v = "sa"
>>> "%s=%s" % (k, v) ❶
'uid=sa'
```

❶ La expresión completa se evalúa a una cadena. El primer `%s` lo reemplaza el valor de `k`; el segundo `%s` se ve sustituido por el valor de `v`. Todos los otros

caracteres de la cadena (en este caso, el signo «igual») quedan tal como están.

Observe que `(k, v)` es una tupla. Ya le dije que servían para algo.

Puede que esté pensando que esto es mucho trabajo sólo para hacer una simple concatenación, y estaría en lo correcto, excepto porque el formato de cadenas no es sólo concatenación. Ni siquiera es sólo formato. También trata sobre conversión de tipos.

Ejemplo 3.22. Formato de cadenas frente a Concatenación

```
>>> uid = "sa"
>>> pwd = "secret"
>>> print pwd + " no es una buena clave para " + uid      ❶
secret no es una buena clave para sa
>>> print "%s no es una buena clave para %s" % (pwd, uid) ❷
secret no es una buena clave para sa
>>> userCount = 6
>>> print "Usuarios conectados: %d" % (userCount, )          ❸ ❹
Usuarios conectados: 6
>>> print "Usuarios conectados: " + userCount                ❺
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

❶ + es el operador de concatenación de cadenas.

❷ En este caso trivial, la cadena de formato lleva al mismo resultado que la concatenación.

❸ `(userCount, )` es una tupla con un elemento. Sí, la sintaxis es un poco extraña, pero hay una buena razón para ella: es una tupla sin ambigüedades. De hecho, siempre puede incluir una coma tras el último elemento cuando define una lista, tupla o diccionario, pero se precisa la coma cuando se define una tupla con un solo elemento. Si no se precisase la coma, Python no podría saber si `(userCount)` es una tupla de un elemento, o sólo el valor de `userCount`.

❹ El formato de cadenas funciona con enteros especificando `%d` en lugar de `%s`.

❺ Tratar de concatenar una cadena con algo que no lo eso lanza una excepción. Al contrario que el formato de cadenas, la concatenación sólo funciona cuando todo son cadenas.

Como con la `printf` en C, el formato de cadenas de Python es como una navaja suiza. Hay gran cantidad de opciones, y modificadores de cadena que dan formato especialmente a varios tipos de valores diferentes.

Ejemplo 3.23. Dar formato a números

```
>>> print "Precio de hoy del stock: %f" % 50.4625    ❶
50.462500
>>> print "Precio de hoy del stock: %.2f" % 50.4625 ❷
50.46
>>> print "Cambio desde ayer: %+.2f" % 1.5 ❸
+1.50
```

❶ La opción de formato `%f` trata el valor como decimal, e imprime seis dígitos decimales.

❷ El modificador ".2" de la opción `%f` trunca el valor a dos dígitos decimales.

❸ Incluso puede combinar modificadores. Añadir el modificador + muestra un signo más o menos delante del valor. Verá que el modificador ".2" sigue ahí, y está haciendo que el valor aparezca con exactamente dos dígitos decimales.

Lecturas complementarias sobre formato de cadenas

- La *Referencia de bibliotecas de Python* enumera todos los caracteres de formato de cadenas.
- El *Effective AWK Programming* comenta todos los caracteres de formato y técnicas avanzadas de formato de cadenas como especificar ancho, precisión y rellenado con ceros.

3.6. Inyección de listas (*mapping*)

Una de las carácterísticas más potentes de Python es la lista por comprensión (*list comprehension*), que proporciona una forma compacta de inyectar una lista en otra aplicando una función a cada uno de sus elementos.

Ejemplo 3.24. Presentación de las listas por comprensión (*list comprehensions*)

```
>>> li = [1, 9, 8, 4]
>>> [elem*2 for elem in li]      ❶
[2, 18, 16, 8]
>>> li                           ❷
[1, 9, 8, 4]
>>> li = [elem*2 for elem in li] ❸
>>> li
[2, 18, 16, 8]
```

❶ Para que esto tenga sentido, léalo de derecha a izquierda. `li` es la lista que está inyectando. Python itera sobre `li` elemento a elemento, asignando temporalmente el valor de cada elemento a la variable `elem`. Python aplica entonces la función `elem*2` y añade el resultado a la lista que ha de devolver.

❷ Observe que las listas por comprensión no modifican la original.

❸ Es seguro asignar el resultado de una lista por comprensión a la variable que está inyectando. Python construye la nueva lista en memoria, y cuando la completa, asigna el resultado a la variable.

Aquí está la lista por comprensión de la función `buildConnectionString` que declaramos en Capítulo 2:

```
["%s=%s" % (k, v) for k, v in params.items()]
```

Primero, comprobará que está llamando a la función `items` del diccionario `params`. Esta función devuelve una lista de tuplas con todos los datos del diccionario.

Ejemplo 3.25. Las funciones `keys`, `values`, e `items`

```
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa",
"pwd":"secret"}
>>> params.keys()    ❶
['server', 'uid', 'database', 'pwd']
>>> params.values() ❷
['mpilgrim', 'sa', 'master', 'secret']
>>> params.items()  ❸
[('server', 'mpilgrim'), ('uid', 'sa'), ('database', 'master'),
('pwd', 'secret')]
```

❶ El método `keys` de un diccionario devuelve una lista de todas las claves. La lista no está en el orden en que se definió el diccionario (recuerde que los elementos de un diccionario no tienen orden), pero es una lista.

❷ El método `values` devuelve una lista de todos los valores. La lista está en el mismo orden que la devuelta por `keys`, de manera que `params.values()[n] == params[params.keys()[n]]` para todos los valores de `n`.

❸ El método `items` devuelve una lista de tuplas de la forma `(`*`clave`*`, `*`valor`*`)`. La lista contiene todos los datos del diccionario.

Ahora veamos qué hace `buildConnectionString`. Toma una lista, `params.items()`, y la inyecta en una nueva lista aplicando formato de cadenas a cada elemento. La nueva lista contendrá el mismo número de elementos que `params.items()`, pero cada uno de estos nuevos elemetnos será una cadena que contenga una clave y su valor asociado en el diccionario `params`.

Ejemplo 3.26. Listas por comprensión en `buildConnectionString`, paso a paso

```
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa",
"pwd":"secret"}
>>> params.items()
[('server', 'mpilgrim'), ('uid', 'sa'), ('database', 'master'),
('pwd', 'secret')]
>>> [k for k, v in params.items()]                ❶
['server', 'uid', 'database', 'pwd']
>>> [v for k, v in params.items()]                ❷
['mpilgrim', 'sa', 'master', 'secret']
>>> ["%s=%s" % (k, v) for k, v in params.items()] ❸
```

```
['server=mpilgrim', 'uid=sa', 'database=master', 'pwd=secret']
```

❶ Observe que estamos usando dos variables para iterar sobre la lista `params.items()`. Éste es otro uso de la asignación multivariable. El primer elemento de `params.items()` es `('server', 'mpilgrim')`, de manera que en la primera iteración de la lista por comprensión, `k` será `'server'` y `v` será `'mpilgrim'`. En este caso, estamos ignorando el valor de `v` y sólo incluimos el valor de `k` en la lista devuelta, de manera que esta lista acaba siendo equivalente a `params.keys()`.

❷ Aquí hacemos lo mismo, pero ignoramos el valor de `k`, así que la lista acaba siendo equivalente a `params.values()`.

❸ Combinar los dos ejemplos anteriores con formato de cadenas sencillo, nos da una lista de cadenas que incluye tanto la clave como el valor de cada elemento del diccionario. Esto se parece sospechosamente a la salida del programa. Todo lo que nos queda es juntar los elementos de esta lista en una sola cadena.

Lecturas complementarias sobre listas por comprensión

- El *Tutorial de Python* comenta otra manera de inyectar listas usando la función incorporada `map` function.
- El *Tutorial de Python* muestra cómo hacer listas por comprensión anidadas.

3.7. Unir listas y dividir cadenas

Tenemos una lista de pares clave-valor de forma *`clave=valor`*, y queremos juntarlos en una sola cadena. Para juntar una lita de cadenas en una sola, usaremos el método `join` de un objeto de cadena.

Aquí tiene un ejemplo de unión de una lista sacado de la función `buildConnectionString`:

```
    return ";".join(["%s=%s" % (k, v) for k, v in params.items()])
```

Una nota interesante antes de continuar. He repetido que las funciones son objetos, las cadenas son objetos... todo es un objeto. Podría estar pensando que quiero decir que las *variables* de cadena son objetos. Pero no, mire atentamente a este ejemplo y verá que la cadena `";"` en sí es un objeto, y está llamando a su método `join`.

El método `join` une los elementos de la lista en una única cadena, separado cada uno por un punto y coma. El delimitador no tiene por qué ser un punto y coma; no tiene siquiera por qué ser un único carácter. Puede ser cualquier cadena.

`join` funciona sólo sobre listas de cadenas; no hace ningún tipo de conversión de tipos. Juntar una lista que tenga uno o más elementos que no sean cadenas provocará una excepción.

Ejemplo 3.27. La salida de `odbchelper.py`

```
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa",
"pwd":"secret"}
>>> ["%s=%s" % (k, v) for k, v in params.items()]
['server=mpilgrim', 'uid=sa', 'database=master', 'pwd=secret']
>>> ";".join(["%s=%s" % (k, v) for k, v in params.items()])
'server=mpilgrim;uid=sa;database=master;pwd=secret'
```

Esta cadena la devuelve la función `odbchelper` y el bloque que hace la llamada la imprime, lo que nos da la salida de la que se maravilló cuando empezó a leer este capítulo.

Probablemente se pregunta si hay un método análogo para dividir una cadena en una lista de trozo. Y por supuesto la hay, y se llama `split`.

Ejemplo 3.28. Dividir una cadena

```
>>> li = ['server=mpilgrim', 'uid=sa', 'database=master',
'pwd=secret']
>>> s = ";".join(li)
>>> s
'server=mpilgrim;uid=sa;database=master;pwd=secret'
>>> s.split(";")    ❶
['server=mpilgrim', 'uid=sa', 'database=master', 'pwd=secret']
>>> s.split(";", 1) ❷
['server=mpilgrim', 'uid=sa;database=master;pwd=secret']
```

❶ `split` deshace el trabajo de `join` dividiendo una cadena en una lista de varios elementos. Advierta que el delimitador (“;”) queda completamente eliminado; no aparece en ninguno de los elementos de la cadena devuelta.

❷ `split` toma un segundo argumento opcional, que es el número de veces a dividir. (“"Oooooh, argumentos opcionales...” Aprenderá cómo hacer esto en sus propias funciones en el siguiente capítulo.)

`unacadena``.split(`*`delimitador`*`, 1)` es una forma útil de buscar una subcadena dentro de una cadena, y trabajar después con todo lo que hay antes de esa subcadena (que es el primer elemento de la lista devuelta) y todo lo que hay detrás (el segundo elemento).

Lecturas complementarias sobre métodos de cadenas

- La Python Knowledge Base responde a las preguntas comunes sobre cadenas y tiene mucho código de ejemplo que usa cadenas.
- La *Referencia de bibliotecas de Python* enumera todos los métodos de las cadenas.
- La *Referencia de bibliotecas de Python* documenta el módulo `string`.
- La *The Whole Python FAQ* explica por qué `join` es un método de cadena en lugar de ser un método de lista.

3.7.1. Nota histórica sobre los métodos de cadena

Cuando aprendí Python, esperaba que `join` fuese un método de una lista, que tomaría delimitadores como argumento. Mucha gente piensa lo mismo, pero

hay toda una historia tras el método `join`. Antes de Python 1.6, las cadenas no tenían todos estos métodos tan útiles. Había un módulo `string` aparte que contenía todas las funciones de cadenas; cada función tomaba una cadena como primer argumento. Se consideró que estas funciones eran suficientemente importantes como para ponerlas en las propias cadenas, lo cual tenía sentido en funciones como `lower`, `upper`, y `split`. Pero muchos programadores del ala dura de Python objetaron ante el nuevo método `join`, argumentando que debería ser un método de las listas, o que no debería moverse de lugar siquiera, sino seguir siendo parte exclusiva del módulo `string` (que aún sigue conteniendo muchas cosas útiles). Yo uso el nuevo método `join` de forma exclusiva, pero verá mucho código escrito de la otra manera, y si realmente le molesta, puede limitarse a usar la antigua función `string.join` en su lugar.

3.8. Resumen

El programa `odbchelper.py` y su salida debería tener total sentido ahora.

```
def buildConnectionString(params):
    """Crea una cadena de conexión partiendo de un diccionario de
parámetros.

    Devuelve una cadena."""
    return ";".join(["%s=%s" % (k, v) for k, v in params.items()])

if __name__ == "__main__":
    myParams = {"server":"mpilgrim", \
                "database":"master", \
                "uid":"sa", \
                "pwd":"secret" \
                }
    print buildConnectionString(myParams)
```

Aquí está la salida de `odbchelper.py`:

```
server=mpilgrim;uid=sa;database=master;pwd=secret
```

Antes de sumergirnos en el siguiente capítulo, asegúrese de que hace las siguientes cosas con comodidad:

- Usar el IDE de Python para probar expresiones de forma interactiva
- Escribir programas en Python y ejecutarlos desde dentro del IDE, o desde la línea de órdenes
- Importar módulos y llamar a sus funciones
- Declarar funciones y usar `cadenas de documentación`, variables locales, e sangrado adecuado
- Definir diccionarios, tuplas, y listas
- Acceder a atributos y métodos de cualquier objeto, incluidas cadenas, listas, diccionarios, funciones y módulos
- Concatenar valores mediante formato de cadenas
- Inyectar listas en otras usando listas por comprensión
- Dividir cadenas en listas y juntar listas en cadenas

Capítulo 4. El poder de la introspección

Este capítulo trata uno de los puntos fuertes de Python: la introspección. Como usted sabe, todo en Python es un objeto, y la introspección es código que ve otros módulos y funciones en memoria como objetos, obtiene información sobre ellos y los manipula. De paso, definiremos funciones sin nombre, llamaremos a funciones con argumentos sin el orden establecido, y haremos referencia a funciones cuyos nombres incluso desconocemos.

4.1. Inmersión

Aquí hay un programa en Python completo y funcional. Debería poder comprenderlo en gran parte sólo observándolo. Las líneas numeradas ilustran conceptos cubiertos en Capítulo 2, *Su primer programa en Python*. No se preocupe si el resto del código le parece inquietante; aprenderá todo sobre él en este capítulo.

Ejemplo 4.1. `apihelper.py`

Si aún no lo ha hecho, puede descargar éste ejemplo y otros usados en este libro.

```
def info(object, spacing=10, collapse=1): ❶ ❷ ❸
    """Print methods and doc strings.

    Takes module, class, list, dictionary, or string."""
    methodList = [method for method in dir(object) if
callable(getattr(object, method))]
    processFunc = collapse and (lambda s: " ".join(s.split())) or
(lambda s: s)
    print "\n".join(["%s %s" %
                     (method.ljust(spacing),
                      processFunc(str(getattr(object,
method).__doc__)))
                     for method in methodList])

if __name__ == "__main__":                    ❹ ❺
    print info.__doc__
```

❶ Este módulo tiene una funcion, `info`. Según su declaración, admite tres parámetros: `object`, `spacing` y `collapse`. Los dos últimos son en realidad parámetros opcionales, como veremos en seguida.

❷ La función `info` tiene una `cadena de documentación` de varias líneas que describe sucintamente el propósito de la función. Advierta que no se indica valor de retorno; esta función se utilizará solamente por sus efectos, no por su valor.

❸ El código de la función está sangrado.

❹ El truco `if __name__` permite a este programa hacer algo útil cuando se ejecuta aislado, sin que esto interfiera conque otros programas lo usen como módulo. En este caso, el programa simplemente imprime la `cadena de documentación` de la función `info`.

❺ Las sentencias `if` usan == para la comparación, y no son necesarios los

paréntesis.

La función `info` está diseñada para que la utilice usted, el programador, mientras trabaja en el IDE de Python. Toma cualquier objeto que tenga funciones o métodos (como un módulo, que tiene funciones, o una lista, que tiene métodos) y muestra las funciones y sus `cadenas de documentación`.

Ejemplo 4.2. Ejemplo de uso de `apihelper.py`

```
>>> from apihelper import info
>>> li = []
>>> info(li)
append     L.append(object) -- append object to end
count      L.count(value) -> integer -- return number of occurrences
of value
extend     L.extend(list) -- extend list by appending list elements
index      L.index(value) -> integer -- return index of first
occurrence of value
insert     L.insert(index, object) -- insert object before index
pop        L.pop([index]) -> item -- remove and return item at index
(default last)
remove     L.remove(value) -- remove first occurrence of value
reverse    L.reverse() -- reverse *IN PLACE*
sort       L.sort([cmpfunc]) -- sort *IN PLACE*; if given, cmpfunc(x,
y) -> -1, 0, 1
```

Por omisión, la salida se formatea para que sea de fácil lectura. Las `cadenas de documentación` de varias líneas se unen en una sola línea larga, pero esta opción puede cambiarse especificando `0` como valor del argumento *`collapse`*. Si los nombres de función tienen más de 10 caracteres, se puede especificar un valor mayor en el argumento *`spacing`* para hacer más legible la salida.

Ejemplo 4.3. Uso avanzado de `apihelper.py`

```
>>> import odbchelper
>>> info(odbchelper)
buildConnectionString Build a connection string from a dictionary
Returns string.
>>> info(odbchelper, 30)
```

```
buildConnectionString          Build a connection string from a
dictionary Returns string.
>>> info(odbchelper, 30, 0)
buildConnectionString          Build a connection string from a
dictionary

    Returns string.
```

4.2. Argumentos opcionales y con nombre

Python permite que los argumentos de las funciones tengan valores por omisión; si se llama a la función sin el argumento, éste toma su valor por omisión. Además, los argumentos pueden especificarse en cualquier orden indicando su nombre. Los procedimientos almacenados en SQL Server Transact/SQL pueden hacer esto; si es usted un gurú de los *scripts* en SQL Server, puede saltarse esta parte.

Aquí tiene un ejemplo de `info`, una función con dos argumentos opcionales

```
def info(object, spacing=10, collapse=1):
```

`spacing` y `collapse` son opcionales, porque tienen asignados valores por omisión. `object` es obligatorio, porque no tiene valor por omisión. Si se llama a `info` sólo con un argumento, `spacing` valdrá `10` y `collapse` valdrá `1`. Si se llama a `info` con dos argumentos, `collapse` seguirá valiendo `1`.

Supongamos que desea usted especificar un valor para `collapse`, pero acepta el valor por omisión de `spacing`. En la mayoría de los lenguajes estaría abandonado a su suerte, pues tendría que invocar a la función con los tres argumentos. Pero en Python, los argumentos pueden indicarse por su nombre en cualquier orden.

Ejemplo 4.4. Llamadas válidas a `info`

```
info(odbchelper)                    ❶
info(odbchelper, 12)                ❷
```

```
info(odbchelper, collapse=0)          ❸
info(spacing=15, object=odbchelper) ❹
```

❶ Con un único argumento, `spacing` toma su valor por omisión de `10` y `collapse` toma su valor por omisión de `1`.

❷ Con dos argumentos, `collapse` toma su valor por omisión de `1`.

❸ Aquí se nombra explícitamente el argumento `collapse` y se le da un valor. `spacing` sigue tomando su valor por omisión de `10`.

❹ Incluso los argumentos obligatorios (como `object`, que no tiene valor por omisión) pueden indicarse por su nombre, y los argumentos así indicados pueden aparecer en cualquier orden.

Esto sorprende hasta que se advierte que los argumentos simplemente forman un diccionario. El método "normal" de invocar a funciones sin nombres de argumentos es realmente un atajo por el que Python empareja los valores con sus nombres en el orden en que fueron especificados en la declaración de la función. La mayor parte de las veces llamará usted a las funciones de la forma "normal", pero siempre dispone de esta flexibilidad adicional si la necesita.

Lo único que necesita para invocar a una función es especificar un valor (del modo que sea) para cada argumento obligatorio; el modo y el orden en que se haga esto depende de usted.

Lecturas complementarias sobre argumentos opcionales

- El *Tutorial de Python* explica exactamente cuándo y cómo se evalúan los argumentos por omisión, lo cual es interesante cuando el valor por omisión es una lista o una expresión con efectos colaterales.

4.3. Uso de `type`, `str`, `dir`, y otras funciones incorporadas

Python tiene un pequeño conjunto de funciones incorporadas enormemente útiles. Todas las demás funciones están repartidas en módulos. Esto es una decisión consciente de diseño, para que el núcleo del lenguaje no se hinche como en otros lenguajes de *script* (cof cof, Visual Basic).

4.3.1. La función `type`

La función `type` devuelve el tipo de dato de cualquier objeto. Los tipos posibles se enumeran en el módulo `types`. Esto es útil para funciones auxiliares que pueden manejar distintos tipos de datos.

Ejemplo 4.5. Presentación de `type`

```
>>> type(1)           ❶
<type 'int'>
>>> li = []
>>> type(li)          ❷
<type 'list'>
>>> import odbchelper
>>> type(odbchelper)  ❸
<type 'module'>
>>> import types      ❹
>>> type(odbchelper) == types.ModuleType
True
```

❶ `type` toma cualquier cosa y devuelve su tipo. Y quiero decir cualquier cosa: enteros, cadenas, listas, diccionarios, tuplas, funciones, clases, módulos, incluso se aceptan los tipos.

❷ `type` puede tomar una variable y devolver su tipo.

❸ `type` funciona también con módulos.

❹ Pueden utilizarse las constantes del módulo `types` para comparar tipos de objetos. Esto es lo que hace la función `info`, como veremos en seguida.

4.3.2. La función `str`

La función `str` transforma un dato en una cadena. Todos los tipos de datos pueden transformarse en cadenas.

Ejemplo 4.6. Presentación de `str`

```
>>> str(1)          ❶
'1'
>>> horsemen = ['war', 'pestilence', 'famine']
>>> horsemen
['war', 'pestilence', 'famine']
>>> horsemen.append('Powerbuilder')
>>> str(horsemen)   ❷
"['war', 'pestilence', 'famine', 'Powerbuilder']"
>>> str(odbchelper) ❸
"<module 'odbchelper' from 'c:\\docbook\\dip\\py\\odbchelper.py'>"
>>> str(None)       ❹
'None'
```

❶ Para tipos simples de datos como los enteros, el comportamiento de `str` es el esperado, ya que casi todos los lenguajes tienen una función que convierte enteros en cadenas.

❷ Sin embargo, `str` funciona con cualquier objeto de cualquier tipo. Aquí funciona sobre una lista que hemos construido por partes.

❸ `str` funciona también con módulos. Fíjese que la representación como cadena del módulo incluye su ruta en el disco, por lo que lo que usted obtenga será diferente.

❹ Un comportamiento sutil pero importante de `str` es que funciona con `None`, el valor nulo de Python. Devuelve la cadena `'None'`. Aprovecharemos esto en la función `info`, como veremos en breve.

En el corazón de nuestra función `info` está la potente función `dir`. `dir` devuelve una lista de los atributos y métodos de cualquier objeto: módulos, funciones, cadenas, listas, diccionarios... prácticamente todo.

Ejemplo 4.7. Presentación de `dir`

```
>>> li = []
```

```
>>> dir(li)          ❶
['append', 'count', 'extend', 'index', 'insert',
'pop', 'remove', 'reverse', 'sort']
>>> d = {}
>>> dir(d)           ❷
['clear', 'copy', 'get', 'has_key', 'items', 'keys', 'setdefault',
'update', 'values']
>>> import odbchelper
>>> dir(odbchelper)  ❸
['__builtins__', '__doc__', '__file__', '__name__',
'buildConnectionString']
```

❶ `li` es una lista, luego `dir(li)` devuelve una lista de todos los métodos de una lista. Advierta que la lista devuelta contiene los nombres de los métodos en forma de cadenas, no los propios métodos.

❷ `d` es un diccionario, luego `dir(d)` devuelve una lista con los nombres de los métodos de un diccionario. Al menos uno de estos, `keys`, debería serle familiar.

❸ Aquí es donde empieza lo interesante. `odbchelper` es un módulo, por lo que `dir(odbchelper)` devuelve una lista con cosas definidas en el módulo, incluidos atributos incorporados, como `__name__`, `__doc__`, y cualesquiera otros atributos y métodos que se hayan definido. En este caso, `odbchelper` sólo tiene definido un método, la función `buildConnectionString` que estudiamos en Capítulo 2.

Finalmente, la función `callable` toma cualquier objeto y devuelve `True` si se puede invocar al objeto, o `False` en caso contrario. Los objetos que pueden ser invocados son funciones, métodos de clase o incluso las propias clases. (Más sobre clases en el siguiente capítulo).

Ejemplo 4.8. Presentación de `callable`

```
>>> import string
>>> string.punctuation          ❶
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
>>> string.join                 ❷
<function join at 00C55A7C>
```

```
>>> callable(string.punctuation) ❸
False
>>> callable(string.join)        ❹
True
>>> print string.join.__doc__    ❺
join(list [,sep]) -> string

    Return a string composed of the words in list, with
    intervening occurrences of sep.  The default separator is a
    single space.

    (joinfields and join are synonymous)
```

❶ Las funciones del módulo `string` están desaconsejadas (aunque mucha gente utiliza la función `join`), pero el módulo contiene muchas constantes útiles como `string.puctuation`, que contiene todos los signos habituales de puntuación.

❷ `string.join` es una función que une una lista de cadenas.

❸ `string.punctuation` no puede invocarse; es una cadena. (Una cadena tiene métodos a los que invocar, pero no podemos hacerlo con la propia cadena).

❹ `string.join` puede ser invocada; es una función que toma dos argumentos.

❺ Cualquier objeto al que se pueda invocar puede tener una `cadena de documentación`. Utilizando la función `callable` sobre cada uno de los atributos de un objeto, podemos averiguar cuáles nos interesan (métodos, funciones, clases) y cuáles queremos pasar por alto (constantes, *etc.*) sin saber nada sobre el objeto por anticipado.

4.3.3. Funciones incorporadas

`type`, `str`, `dir` y el resto de funciones incorporadas de Python se agrupan en un módulo especial llamado `__builtin__` (con dos subrayados antes y después). Por si sirve de ayuda, puede interpretarse que Python ejecuta automáticamente `from __builtins__ import *` al inicio, con lo que se importan todas las funciones "incorporadas" en el espacio de nombres de manera que puedan utilizarse directamente.

La ventaja de interpretarlo así es que se puede acceder a todas las funciones y atributos incorporados en grupo, obteniendo información sobre el módulo `__builtins__`. E imagíneselo, tenemos una función para ello: se llama `info`. Inténtelo usted y prescinda ahora de la lista; nos sumergiremos más tarde en algunas de las principales funciones. (Algunas de las clases de error incorporadas, como `AttributeError`, deberían resultarle familiares).

Ejemplo 4.9. Atributos y funciones incorporados

```
>>> from apihelper import info
>>> import __builtin__
>>> info(__builtin__, 20)
ArithmeticError      Base class for arithmetic errors.
AssertionError       Assertion failed.
AttributeError       Attribute not found.
EOFError             Read beyond end of file.
EnvironmentError     Base class for I/O related errors.
Exception            Common base class for all exceptions.
FloatingPointError   Floating point operation failed.
IOError              I/O operation failed.

[...snip...]
```

Python se acompaña de excelentes manuales de referencia, que debería usted leer detenidamente para aprender todos los módulos que Python ofrece. Pero mientras en la mayoría de lenguajes debe usted volver continuamente sobre los manuales o las páginas de manual para recordar cómo se usan estos módulos, Python está autodocumentado en su mayoría.

Lecturas complementarias sobre las funciones incorporadas

- La *Referencia de bibliotecas de Python* documenta todas las funciones incorporadas y todas las excepciones incorporadas.

4.4. Obtención de referencias a objetos con `getattr`

- 4.4.1. getattr con módulos
- 4.4.2. getattr como dispatcher

Ya sabe usted que las funciones de Python son objetos. Lo que no sabe es que se puede obtener una referencia a una función sin necesidad de saber su nombre hasta el momento de la ejecución, utilizando la función `getattr`.

Ejemplo 4.10. Presentación de `getattr`

```
>>> li = ["Larry", "Curly"]
>>> li.pop                      ❶
<built-in method pop of list object at 010DF884>
>>> getattr(li, "pop")          ❷
<built-in method pop of list object at 010DF884>
>>> getattr(li, "append")("Moe") ❸
>>> li
["Larry", "Curly", "Moe"]
>>> getattr({}, "clear")        ❹
<built-in method clear of dictionary object at 00F113D4>
>>> getattr((), "pop")          ❺
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'pop'
```

❶ Esto obtiene una referencia al método `pop` de la lista. Observe que no se invoca al método `pop`; la llamada sería `li.pop()`. Esto es el método en sí.

❷ Aquí también se devuelve una referencia al método `pop`, pero esta vez el nombre del método se especifica como una cadena, argumento de la función `getattr`. `getattr` es una función incorporada increíblemente útil que devuelve cualquier atributo de cualquier objeto. En este caso, el objeto es una lista, y el atributo es el método `pop`.

❸ Si no le ha impresionado aún la increíble utilidad de esta función, intente esto: el valor de retorno de `getattr` *es* el método, que puede ser invocado igual que si se hubiera puesto directamente `li.append("Moe")`. Pero no se ha llamado directamente a la función; en lugar de esto, se ha especificado su nombre como una cadena.

❹ `getattr` también funciona con diccionarios.

❺ En teoría, `getattr` debería funcionar con tuplas, pero como las tuplas no tienen métodos, así que `getattr` lanzará una excepción sea cual sea el nombre de atributo que se le pida.

4.4.1. `getattr` con módulos

`getattr` no sirve sólo para tipos de datos incorporados. También funciona con módulos.

Ejemplo 4.11. La función `getattr` en `apihelper.py`

```
>>> import odbchelper
>>> odbchelper.buildConnectionString                ❶
<function buildConnectionString at 00D18DD4>
>>> getattr(odbchelper, "buildConnectionString")    ❷
<function buildConnectionString at 00D18DD4>
>>> object = odbchelper
>>> method = "buildConnectionString"
>>> getattr(object, method)                         ❸
<function buildConnectionString at 00D18DD4>
>>> type(getattr(object, method))                   ❹
<type 'function'>
>>> import types
>>> type(getattr(object, method)) == types.FunctionType
True
>>> callable(getattr(object, method))               ❺
True
```

❶ Esto devuelve una referencia a la función `buildConnectionString` del módulo `odbchelper`, que estudiamos en Capítulo 2, *Su primer programa en Python*. (La dirección hexadecimal que se ve es específica de mi máquina; la suya será diferente).

❷ Utilizando `getattr`, podemos obtener la misma referencia para la misma función. En general, `getattr(`*`objeto`*`, "`*`atributo`*`")` es equivalente a *`objeto.atributo`*. Si *`objeto`* es un módulo, entonces *`atributo`* puede ser cualquier cosa definida en el módulo: una función, una clase o una variable

global.

❸ Y esto es lo que realmente utilizamos en la función `info`. `object` se pasa como argumento de la función; `method` es una cadena que contiene el nombre de un método o una función.

❹ En este caso, `method` es el nombre de una función, lo que podemos comprobar obteniendo su tipo con `type`.

❺ Como `method` es una función, podemos invocarla.

4.4.2. `getattr` como *dispatcher*

Un patrón común de uso de `getattr` es como *dispatcher*. Por ejemplo, si tuviese un programa que pudiera mostar datos en diferentes formatos, podría definir funciones separadas para cada formato de salida y una única función de despacho para llamar a la adecuada.

Por ejemplo, imagine un programa que imprima estadísticas de un sitio en formatos HTML, XML, y texto simple. La elección del formato de salida podría especificarse en la línea de órdenes, o almacenarse en un fichero de configuración. Un módulo `statsout` define tres funciones, `output_html`, `output_xml`, y `output_text`. Entonces el programa principal define una única función de salida, así:

Ejemplo 4.12. Creación de un *dispatcher* con `getattr`

```
import statsout

def output(data, format="text"):                                ❶
    output_function = getattr(statsout, "output_%s" % format)   ❷
    return output_function(data)                                ❸
```

❶ La función `output` toma un argumento obligatorio, `data`, y uno opcional, `format`. Si no se especifica `format`, por omisión asume `text`, y acabará llamado a la función de salida de texto simple.

❷ Concatenamos el argumento `format` con "output_" para producir un nombre de función, y entonces obtenemos esa función del módulo `statsout`. Esto nos permite extender fácilmente el programa más adelante para que admita otros formatos de salida, sin cambiar la función de despacho. Simplemente añada otra función a `statsout` que se llame, por ejemplo, `output_pdf`, y pase "pdf" como `format` a la función `output`.

❸ Ahora simplemente llamamos a la función de salida de la misma manera que con cualquier otra función. La variable `output_function` es una referencia a la función apropiada del módulo `statsout`.

¿Encontró el fallo en el ejemplo anterior? Este acomplamiento entre cadenas y funciones es muy débil, y no hay comprobación de errores. ¿Qué sucede si el usuario pasa un formato que no tiene definida la función correspondiente en `statsout`? Bien, `getattr` devolverá `None`, que se asignará a `output_function` en lugar de una función válida, y la siguiente línea que intente llamar a esa función provocará una excepción. Esto es malo.

Por suerte, `getattr` toma un tercer argumento opcional, un valor por omisión.

Ejemplo 4.13. Valores por omisión de `getattr`

```
import statsout

def output(data, format="text"):
    output_function = getattr(statsout, "output_%s" % format,
statsout.output_text)
    return output_function(data) ❶
```

❶ Está garantizado que esta llamada a función tendrá éxito, porque hemos añadido un tercer argumento a la llamada a `getattr`. El tercer argumento es un valor por omisión que será devuelto si no se encontrase el atributo o método especificado por el segundo argumento.

Como puede ver, `getattr` es bastante potente. Es el corazón de la introspección, y verá ejemplos aún más poderosos en los siguientes capítulos.

4.5. Filtrado de listas

Como ya sabe, Python tiene potentes capacidades para convertir una lista en otra por medio de las listas por comprensión (Sección 3.6, "Inyección de listas (mapping)"). Esto puede combinarse con un mecanismo de filtrado en el que se van a tomar algunos elementos de la lista mientras otros se pasarán por alto.

Ésta es la sintaxis del filtrado de listas:

```
[expresión de relación for
elemento in lista
origen if expresión de filtrado]
```

Esto es una extensión de las listas por comprensión que usted conoce y que tanto le gustan. Las dos primeras partes son como antes; la última parte, la que comienza con `if`, es la expresión de filtrado. Una expresión de filtrado puede ser cualquier expresión que se evalúe como verdadera o falsa (lo cual, en Python, puede ser casi cualquier resultado). Cualquier elemento para el cual la expresión resulte verdadera, será incluido en el proceso de relación. Todos los demás elementos se pasan por alto, de modo que no entran en el proceso de relación y no se incluyen en la lista de salida.

Ejemplo 4.14. Presentación del filtrado de listas

```
>>> li = ["a", "mpilgrim", "foo", "b", "c", "b", "d", "d"]
>>> [elem for elem in li if len(elem) > 1]       ❶
['mpilgrim', 'foo']
>>> [elem for elem in li if elem != "b"]         ❷
['a', 'mpilgrim', 'foo', 'c', 'd', 'd']
>>> [elem for elem in li if li.count(elem) == 1] ❸
['a', 'mpilgrim', 'foo', 'c']
```

❶ Esta expresión de relación es sencilla (simplemente devuelve el valor de cada elemento), así que concentrémonos en la expresión de filtrado. Mientras Python recorre la lista, aplica la expresión de filtrado a cada elemento; si la expresión es verdadera, se aplica la relación al elemento y el resultado se incluye en la lista final. Aquí estamos filtrando todas las cadenas de un solo

carácter, por lo que se obtendrá una lista con todas las cadenas más largas que eso.

❷ Aquí filtramos un valor concreto, `b`. Observe que esto filtra todas las apariciones de `b`, ya que cada vez que aparezca este valor, la expresión de filtrado será falsa.

❸ `count` es un método de lista que devuelve el número de veces que aparece un valor en una lista. Se podría pensar que este filtro elimina los valores duplicados en una lista, devolviendo otra que contiene una única copia de cada valor de la lista original. Pero no es así, porque los valores que aparecen dos veces en la lista original (en este caso, `b` y `d`) son completamente excluidos. Hay modos de eliminar valores duplicados en una lista, pero el filtrado no es la solución.

Volvamos a esta línea de `apihelper.py`:

```
    methodList = [method for method in dir(object) if
callable(getattr(object, method))]
```

Esto parece complicado, y lo es, pero la estructura básica es la misma. La expresión de filtrado devuelve una lista, que se asigna a la variable `methodList`. La primera mitad de la expresión es la parte de relación. Es una relación de identidad, que devuelve el valor de cada elemento. `dir(object)` devuelve una lista de los atributos y métodos de `object`; esa es la lista a la que se aplica la relación. Luego la única parte nueva es la expresión de filtrado que sigue a `if`.

La expresión de filtrado parece que asusta, pero no. Usted ya conoce `callable`, `getattr`, e `in`. Como se vio en la sección anterior, la expresión `getattr(object, method)` devuelve un objeto función si `object` es un módulo y `method` el nombre de una función de ese módulo.

Por tanto esta expresión toma un objeto (llamado `object`). Obtiene la lista de sus atributos, métodos, funciones y algunas cosas más. A continuación filtra esta lista para eliminar todo lo que no nos interesa. Esto lo hacemos tomando el nombre de cada atributo/método/función y obteniendo una referencia al

objeto real por medio de la función `getattr`. Después comprobamos si ese objeto puede ser invocado, que serán los métodos y funciones, tanto incorporados (como el método `pop` de una lista) como definidos por el usuario (como la función `buildConnectionString` del módulo `odbchelper`). No nos interesan otros atributos, como el atributo `__name__` que está incorporado en todos los módulos.

Lecturas complementarias sobre filtrado de listas

- El *Tutorial de Python* expone otro modo de filtrar listas utilizando la función incorporada `filter`.

4.6. La peculiar naturaleza de `and` y `or`

- 4.6.1. Uso del truco the and-or

En Python, `and` y `or` realizan las operaciones de lógica booleana como cabe esperar, pero no devuelven valores booleanos; devuelven uno de los valores que están comparando.

Ejemplo 4.15. Presentación de `and`

```
>>> 'a' and 'b'          ❶
'b'
>>> '' and 'b'           ❷
''
>>> 'a' and 'b' and 'c'  ❸
'c'
```

❶ Cuando se utiliza `and`, los valores se evalúan en un contexto booleano de izquierda a derecha. `0`, `''`, `[]`, `()`, `{}` y `None` son falsos en un contexto booleano; todo lo demás es verdadero. Bueno, casi todo. Por omisión, las instancias de clases son verdaderas en un contexto booleano, pero se pueden definir métodos especiales en las clases para hacer que una instancia se evalúe como falsa. Aprenderá usted todo sobre las clases y los métodos especiales en Capítulo 5. Si todos los valores son verdaderos en un contexto

booleano, `and` devuelve el último de ellos. En este caso, `and` evalúa `'a'`, que es verdadera, después `'b'`, que es verdadera, y devuelve `'b'`.

❷ Si alguno de los valores es falso en contexto booleano, `and` devuelve el primer valor falso. En este caso, `''` es el primer valor falso.

❸ Todos los valores son verdaderos, luego `and` devuelve el último valor, `'c'`.

Ejemplo 4.16. Presentación de `or`

```
>>> 'a' or 'b'          ❶
'a'
>>> '' or 'b'           ❷
'b'
>>> '' or [] or {}      ❸
{}
>>> def sidefx():
...     print "in sidefx()"
...     return 1
>>> 'a' or sidefx()     ❹
'a'
```

❶ Cuando se utiliza `or`, los valores se evalúan en un contexto booleano, de izquierda a derecha, como con `and`. Si algún valor es verdadero, `or` devuelve ese valor inmediatamente. En este caso, `'a'` es el primer valor verdadero.

❷ `or` evalúa `''`, que es falsa, después `'b'`, que es verdadera, y devuelve `'b'`.

❸ Si todos los valores son falsos, `or` devuelve el último. `or` evalúa `''`, que es falsa, después `[]`, que es falsa, después `{}`, que es falsa, y devuelve `{}`.

❹ Advierta que `or` sólo evalúa valores hasta que encuentra uno verdadero en contexto booleano, y entonces omite el resto. Esta distinción es importante si algunos valores tienen efectos laterales. Aquí no se invoca nunca a la función `sidefx`, porque `or` evalúa u`'a'`, que es verdadera, y devuelve `'a'` inmediatamente.

Si es usted un hacker de C, le será familiar la expresión *`booleano`* `? a : b`, que se evalúa como `a` si *`bool`* es verdadero, y `b` en caso contrario. Por el modo en que `and` y `or` funcionan en Python, se puede obtener el mismo efecto.

4.6.1. Uso del truco the `and-or`

Ejemplo 4.17. Presentación del truco `and-or`

```
>>> a = "first"
>>> b = "second"
>>> 1 and a or b ❶
'first'
>>> 0 and a or b ❷
'second'
```

❶ Esta sintaxis resulta similar a la de la expresión *`booleano ? a : b`* en C. La expresión entera se evalúa de izquierda a derecha, luego `and` se evalúa primero. `1 and 'first'` da como resultado `'first'`, después `'first' or 'second'` da como resultado `'first'`.

❷ `0 and 'first'` da como resultado `0`, después `0 or 'second'` da como resultado `'second'`.

Sin embargo, como esta expresión de Python es simplemente lógica booleana, y no una construcción especial del lenguaje, hay una diferencia muy, muy, muy importante entre este truco `and-or` en Python y la sintaxis *`booleano ? a : b`* en C. Si el valor de `a` es falso, la expresión no funcionará como sería de esperar. (¿Puede imaginar que llegué a tener problemas con esto? ¿Más de una vez?)

Ejemplo 4.18. Cuando falla el truco `and-or`

```
>>> a = ""
>>> b = "second"
>>> 1 and a or b         ❶
'second'
```

❶ Como `a` es una cadena vacía, que Python considera falsa en contexto booleano, `1 and ''` se evalúa como `''`, después `'' or 'second'` se evalúa como `'second'`. ¡Uy! Eso no es lo que queríamos.

El truco `and-or`, *`bool`* `and a or b`, no funcionará como la expresión *`booleano`* `? a : b` en C cuando `a` sea falsa en contexto booleano.

El truco real que hay tras el truco `and-or` es pues asegurarse de que el valor de `a` nunca es falso. Una forma habitual de hacer esto es convertir `a` en `[a]` y `b` en `[b]`, y después tomar el primer elemento de la lista devuelta, que será `a` o `b`.

Ejemplo 4.19. Utilización segura del truco `and-or`

```
>>> a = ""
>>> b = "second"
>>> (1 and [a] or [b])[0] ❶
''
```

❶ Dado que `[a]` es una lista no vacía, nunca es falsa. Incluso si `a` es `0` o `''` o cualquier otro valor falso, la lista `[a]` es verdadera porque tiene un elemento.

Hasta aquí, puede parecer que este truco tiene más inconvenientes que ventajas. Después de todo, se podría conseguir el mismo efecto con una sentencia `if`, así que ¿por qué meterse en este follón? Bien, en muchos casos, se elige entre dos valores constantes, luego se puede utilizar la sintaxis más simple sin preocuparse, porque se sabe que el valor de `a` será siempre verdadero. E incluso si hay que usar la forma más compleja, hay buenas razones para ello; en algunos casos no se permite la sentencia `if`, por ejemplo en las funciones `lambda`.

Lecturas complementarias sobre el truco `and-or`

- Python Cookbook expone alternativas al truco `and-or`.

4.7. Utilización de las funciones `lambda`

Python admite una interesante sintaxis que permite definir funciones mínimas, de una línea, sobre la marcha. Tomada de Lisp, se trata de las denominadas funciones `lambda`, que pueden utilizarse en cualquier lugar donde se necesite una función.

Ejemplo 4.20. Presentación de las funciones `lambda`

```
>>> def f(x):
...     return x*2
...
>>> f(3)
6
>>> g = lambda x: x*2  ❶
>>> g(3)
6
>>> (lambda x: x*2)(3) ❷
6
```

❶ Ésta es una función `lambda` que consigue el mismo efecto que la función anterior. Advierta aquí la sintaxis abreviada: la lista de argumentos no está entre paréntesis, y falta la palabra reservada `return` (está implícita, ya que la función entera debe ser una única expresión). Igualmente, la función no tiene nombre, pero puede ser llamada mediante la variable a que se ha asignado.

❷ Se puede utilizar una función `lambda` incluso sin asignarla a una variable. No es lo más útil del mundo, pero sirve para mostrar que una `lambda` es sólo una función en línea.

En general, una función `lambda` es una función que toma cualquier número de argumentos (incluso argumentos opcionales) y devuelve el valor de una expresión simple. Las funciones `lambda` no pueden contener órdenes, y no pueden contener tampoco más de una expresión. No intente exprimir demasiado una función `lambda`; si necesita algo más complejo, defina en su lugar una función normal y hágala tan grande como quiera.

Las funciones `lambda` son una cuestión de estilo. Su uso nunca es necesario. En cualquier sitio en que puedan utilizarse, se puede definir una función normal separada y utilizarla en su lugar. Yo las utilizo en lugares donde deseo encapsulación, código no reutilizable que no ensucie mi propio código con un montón de pequeñas funciones de una sola línea.

4.7.1. Funciones `lambda` en el mundo real

Aquí están las funciones `lambda` de `apihelper.py`:

```
    processFunc = collapse and (lambda s: " ".join(s.split())) or
(lambda s: s)
```

Observe que aquí usamos la forma simple del truco `and-or`, lo cual está bien, porque una función `lambda` siempre es verdadera en un contexto booleano (esto no significa que una función `lambda` no pueda devolver un valor falso; la función es siempre verdadera; su valor de retorno puede ser cualquier cosa).

Adiverta también que estamos usando la función `split` sin argumentos. Ya ha visto usarla con uno o más argumentos, pero sin argumentos hace la división en los espacios en blanco.

Ejemplo 4.21. `split` sin argumentos

```
>>> s = "this   is\na\ttest"  ❶
>>> print s
this   is
a       test
>>> print s.split()           ❷
['this', 'is', 'a', 'test']
>>> print " ".join(s.split()) ❸
'this is a test'
```

❶ Ésta es una cadena de varias líneas, definida por caracteres de escape en lugar detriples comillas. `\n` es el retorno de carro; `\t` es el carácter de tabulación.

❷ `split` sin argumentos divide en los espacios en blanco. De modo que tres espacios, un retorno de carro y un carácter de tabulación son lo mismo.

❸ Se puede normalizar el espacio en blanco dividiendo una cadena con `split` y volviéndola a unir con `join` con un espacio simple como delimitador. Esto es lo que hace la función `info` para unificar las `cadenas de documentación` de varias líneas en una sola.

Entonces, ¿qué hace realmente la función `info` con estas funciones `lambda`, con `split` y con los trucos `and-or`?

```
    processFunc = collapse and (lambda s: " ".join(s.split())) or
(lambda s: s)
```

`processFunc` es ahora una función, pero qué función sea depende del valor de la variable `collapse`. Si `collapse` es verdadera, `processFunc(`*`cadena`*`)` unificará el espacio en blanco; en caso contrario, `processFunc(`*`cadena`*`)` devolverá su argumento sin cambios.

Para hacer esto con un lenguaje menos robusto, como Visual Basic, probablemente crearía usted una función que tomara una cadena y un argumento *`collapse`* utilizando una sentencia `if` para decidir si unificar o no el espacio en blanco, para devolver después el valor apropiado. Esto sería ineficaz, porque la función tendría que considerar todos los casos posibles; cada vez que se le llamara, tendría que decidir si unificar o no el espacio en blanco antes de devolver el valor deseado. En Python, puede tomarse esta decisión fuera de la función y definir una función `lambda` adaptada para devolver exactamente (y únicamente) lo que se busca. Esto es más eficaz, más elegante, y menos propenso a esos errores del tipo "Uy, creía que esos argumentos estaban al revés".

Lecturas complementarias sobre funciones `lambda`

- La Python Knowledge Base explica el uso de funciones `lambda` para llamar funciones indirectamente..
- El *Tutorial de Python* muestra cómo \ acceder a variables externas desde dentro de una función `lambda`. (PEP 227 expone cómo puede cambiar esto en futuras versiones de Python.)
- *The Whole Python FAQ* tiene ejemplos de códigos confusos de una línea que utilizan funciones `lambda`.

4.8. Todo junto

La última línea de código, la única que no hemos desmenuzado todavía, es la que hace todo el trabajo. Pero el trabajo ya es fácil, porque todo lo que necesitamos está dispuesto de la manera en que lo necesitamos. Las fichas de dominó están en su sitio; lo que queda es golpear la primera.

El meollo de `apihelper.py`:

```
    print "\n".join(["%s %s" %
                     (method.ljust(spacing),
                      processFunc(str(getattr(object,
method).__doc__)))
                    for method in methodList])
```

Advierta que esto es una sola orden, dividida en varias líneas, pero sin utilizar el carácter de continuación de línea ("\"). ¿Recuerda cuando dije que algunas expresiones pueden dividirse en varias líneas sin usar la barra inversa? Una lista por comprensión es una de estas expresiones, ya que la expresión completa está entre corchetes.

Vamos a tomarlo por el final y recorrerlo hacia atrás. El fragmento

```
for method in methodList
```

nos muestra que esto es una lista por comprensión. Como ya sabe, `methodList` es una lista de todos los métodos que nos interesan de `object`. De modo que estamos recorriendo esta lista con la variable `method`.

Ejemplo 4.22. Obtención de una `cadena de documentación` de forma dinámica

```
>>> import odbchelper
>>> object = odbchelper                    ❶
>>> method = 'buildConnectionString'       ❷
>>> getattr(object, method)                ❸
<function buildConnectionString at 010D6D74>
>>> print getattr(object, method).__doc__  ❹
Crea una cadena de conexión partiendo de un diccionario de parámetros.
```

```
    Devuelve una cadena.
```

❶ En la función `info`, `object` es el objeto sobre el que pedimos ayuda, pasado como argumento.

❷ Cuando recorremos `methodList`, `method` es el nombre del método actual.

❸ Usando la función `getattr`, obtenemos una referencia a la función *`método`* del módulo *`objeto`*.

❹ Ahora, mostrar la `cadena de documentación` del método es fácil.

La siguiente pieza del puzzle es el uso de `str` con la `cadena de documentación`. Como recordará usted, `str` es una función incorporada que convierte datos en cadenas. Pero una `cadena de documentación` es siempre una cadena, luego ¿por qué molestarse en usar la función `str`? La respuesta es que no todas las funciones tienen una `cadena de documentación`, y en este caso su atributo `__doc__` es `None`.

Ejemplo 4.23. ¿Por qué usar `str` con una `cadena de documentación`?

```
>>> >>> def foo(): print 2
>>> >>> foo()
2
>>> >>> foo.__doc__      ❶
>>> foo.__doc__ == None ❷
True
>>> str(foo.__doc__)     ❸
'None'
```

❶ Se puede definir fácilmente una función que no tenga `cadena de documentación`, de manera que su atributo `__doc__` es `None`. Para más confusión, si se evalúa directamente el atributo `__doc__`, el IDE de Python no muestra nada en absoluto, lo cual tiene sentido si se piensa bien, pero es poco práctico.

❷ Se puede verificar que el valor del atributo `__doc__` es realmente `None` comparándolo directamente.

❸ Con el uso de la función `str` se toma el valor nulo y se devuelve su

representación como cadena, `'None'`.

En SQL, se utiliza `IS NULL` en vez de `= NULL` para comparar un valor nulo. En Python puede usar tanto `== None` como `is None`, pero `is None` es más rápido.

Ahora que estamos seguros de tener una cadena, podemos pasarla a `processFunc`, que hemos definido como una función que unifica o no el espacio en blanco. Ahora se ve por qué es importante utilizar `str` para convertir el valor `None` en su representación como cadena. `processFunc` asume que su argumento es una cadena y llama a su método `split`, el cual fallaría si le pasamos `None`, ya que `None` no tiene un método `split`.

Yendo más atrás, verá que estamos utilizando las cadenas de formato de nuevo para concatenar el valor de retorno de `processFunc` con el valor de retorno del método `ljust` de `method`. Éste es un nuevo método de cadena que no hemos visto antes.

Ejemplo 4.24. Presentación de `ljust`

```
>>> s = 'buildConnectionString'
>>> s.ljust(30) ❶
'buildConnectionString        '
>>> s.ljust(20) ❷
'buildConnectionString'
```

❶ `ljust` rellena la cadena con espacios hasta la longitud indicada. Esto lo usa la función `info` para hacer dos columnas y alinear todas las `cadenas de documentación` en la segunda columna.

❷ Si la longitud indicada es menor que la longitud de la cadena, `ljust` devuelve simplemente la cadena sin cambios. Nunca corta la cadena.

Casi hemos terminado. Dados el nombre del método relleno con espacios con el método `ljust` y la `cadena de documentación` (posiblemente con el espacio blanco unificado), que resultó de la llamada a `processFunc`, concatenamos las

dos y obtenemos una única cadena. Como estamos recorriendo `methodList`, terminamos con una lista de cadenas. Utilizando el método `join` de la cadena `"\n"`, unimos esta lista en una única cadena, con cada elemento de la lista en una línea diferente, y mostramos el resultado.

Ejemplo 4.25. Mostrar una List

```
>>> li = ['a', 'b', 'c']
>>> print "\n".join(li) ❶
a
b
c
```

❶ Éste es también un útil truco de depuración cuando se trabaja con listas. Y en Python, siempre se trabaja con listas.

Ésta es la última pieza del puzzle. Este código debería ahora entenderse perfectamente.

```
    print "\n".join(["%s %s" %
                      (method.ljust(spacing),
                       processFunc(str(getattr(object,
method).__doc__)))
                     for method in methodList])
```

4.9. Resumen

El programa `apihelper.py` y su salida deberían entenderse ya perfectamente.

```
def info(object, spacing=10, collapse=1):
    """Print methods and doc strings.

    Takes module, class, list, dictionary, or string."""
    methodList = [method for method in dir(object) if
callable(getattr(object, method))]
    processFunc = collapse and (lambda s: " ".join(s.split())) or
(lambda s: s)
    print "\n".join(["%s %s" %
                      (method.ljust(spacing),
```

```
                        processFunc(str(getattr(object,
method).__doc__)))
                    for method in methodList])

if __name__ == "__main__":
    print info.__doc__
```

Aquí está la salida de `apihelper.py`:

```
>>> from apihelper import info
>>> li = []
>>> info(li)
append     L.append(object) -- append object to end
count      L.count(value) -> integer -- return number of occurrences
of value
extend     L.extend(list) -- extend list by appending list elements
index      L.index(value) -> integer -- return index of first
occurrence of value
insert     L.insert(index, object) -- insert object before index
pop        L.pop([index]) -> item -- remove and return item at index
(default last)
remove     L.remove(value) -- remove first occurrence of value
reverse    L.reverse() -- reverse *IN PLACE*
sort       L.sort([cmpfunc]) -- sort *IN PLACE*; if given, cmpfunc(x,
y) -> -1, 0, 1
```

Antes de sumergirnos en el siguiente capítulo, asegúrese de que se siente cómodo haciendo todo esto:

- Definir y llamar funciones con argumentos opcionales y por nombre.
- Utilizar `str` para convertir un valor arbitrario en una representación de cadena.
- Utilizar `getattr` para obtener referencias a funciones y otros atributos dinámicamente.
- Extender la sintaxis de listas por comprensión para hacer filtrado de listas.
- Reconocer el truco `and-or` y usarlo sin riesgos.
- Definir funciones `lambda`.

- Asignar funciones a variables y llamar a la función haciendo referencia a la variable. Nunca se remarcará bastante: este modo de pensar es vital para avanzar en la comprensión de Python. Se verán aplicaciones más complejas de este concepto a través de este libro.

Capítulo 5. Objetos y orientación a objetos

Este capítulo, y básicamente todos los que le siguen trabajan con programación en Python orientada a objetos.

5.1. Inmersión

Aquí tiene un programa en Python completo y funcional. Lea las `cadenas de documentación` del módulo, las clases, y las funciones para obtener una idea general de lo que hace el programa y cómo funciona. Como de costumbre, no se preocupe por lo que no entienda; para eso está el resto del capítulo.

Ejemplo 5.1. `fileinfo.py`

Si aún no lo ha hecho, puede descargar éste ejemplo y otros usados en este libro.

```
"""Framework for getting filetype-specific metadata.
```

```
Instantiate appropriate class with filename.  Returned object acts
like a
dictionary, with key-value pairs for each piece of metadata.
    import fileinfo
    info = fileinfo.MP3FileInfo("/music/ap/mahadeva.mp3")
    print "\\n".join(["%s=%s" % (k, v) for k, v in info.items()])

Or use listDirectory function to get info on all files in a directory.
    for info in fileinfo.listDirectory("/music/ap/", [".mp3"]):
        ...

Framework can be extended by adding classes for particular file types,
e.g.
HTMLFileInfo, MPGFileInfo, DOCFileInfo.  Each class is completely
responsible for
parsing its files appropriately; see MP3FileInfo for example.
"""
import os
import sys
from UserDict import UserDict

def stripnulls(data):
    "strip whitespace and nulls"
    return data.replace("\00", "").strip()

class FileInfo(UserDict):
    "store file metadata"
    def __init__(self, filename=None):
        UserDict.__init__(self)
        self["name"] = filename

class MP3FileInfo(FileInfo):
    "store ID3v1.0 MP3 tags"
    tagDataMap = {"title"   : (  3,  33, stripnulls),
                  "artist"  : ( 33,  63, stripnulls),
                  "album"   : ( 63,  93, stripnulls),
                  "year"    : ( 93,  97, stripnulls),
                  "comment" : ( 97, 126, stripnulls),
                  "genre"   : (127, 128, ord)}

    def __parse(self, filename):
        "parse ID3v1.0 tags from MP3 file"
```

```
        self.clear()
        try:
            fsock = open(filename, "rb", 0)
            try:
                fsock.seek(-128, 2)
                tagdata = fsock.read(128)
            finally:
                fsock.close()
            if tagdata[:3] == "TAG":
                for tag, (start, end, parseFunc) in
self.tagDataMap.items():
                    self[tag] = parseFunc(tagdata[start:end])
        except IOError:
            pass

    def __setitem__(self, key, item):
        if key == "name" and item:
            self.__parse(item)
        FileInfo.__setitem__(self, key, item)

def listDirectory(directory, fileExtList):
    "get list of file info objects for files of particular extensions"
    fileList = [os.path.normcase(f)
                for f in os.listdir(directory)]
    fileList = [os.path.join(directory, f)
                for f in fileList
                if os.path.splitext(f)[1] in fileExtList]
    def getFileInfoClass(filename,
module=sys.modules[FileInfo.__module__]):
        "get file info class from filename extension"
        subclass = "%sFileInfo" %
os.path.splitext(filename)[1].upper()[1:]
        return hasattr(module, subclass) and getattr(module, subclass)
or FileInfo
    return [getFileInfoClass(f)(f) for f in fileList]

if __name__ == "__main__":
    for info in listDirectory("/music/_singles/", [".mp3"]): ❶
        print "\n".join(["%s=%s" % (k, v) for k, v in info.items()])
        print
```

❶ La salida de este programa depende de los ficheros de su disco duro. Para

obtener una salida con sentido, necesitará cambiar la ruta de directorio para que apunte a uno que contenga ficheros MP3 en su propia máquina.

Ésta es la salida que obtengo en mi máquina. La de usted puede diferir, a menos que, por alguna sorprendente coincidencia, tenga exactamente mis mismos gustos musicales.

```
album=
artist=Ghost in the Machine
title=A Time Long Forgotten (Concept
genre=31
name=/music/_singles/a_time_long_forgotten_con.mp3
year=1999
comment=http://mp3.com/ghostmachine

album=Rave Mix
artist=***DJ MARY-JANE***
title=HELLRAISER****Trance from Hell
genre=31
name=/music/_singles/hellraiser.mp3
year=2000
comment=http://mp3.com/DJMARYJANE

album=Rave Mix
artist=***DJ MARY-JANE***
title=KAIRO****THE BEST GOA
genre=31
name=/music/_singles/kairo.mp3
year=2000
comment=http://mp3.com/DJMARYJANE

album=Journeys
artist=Masters of Balance
title=Long Way Home
genre=31
name=/music/_singles/long_way_home1.mp3
year=2000
comment=http://mp3.com/MastersofBalan

album=
artist=The Cynic Project
```

```
title=Sidewinder
genre=18
name=/music/_singles/sidewinder.mp3
year=2000
comment=http://mp3.com/cynicproject

album=Digitosis@128k
artist=VXpanded
title=Spinning
genre=255
name=/music/_singles/spinning.mp3
year=2000
comment=http://mp3.com/artists/95/vxp
```

5.2. Importar módulos usando `from módulo import`

Python tiene dos maneras de importar módulos. Ambas son útiles, y debe saber cuándo usar cada cual. Una de las maneras, `import módulo`, ya la hemos visto en Sección 2.4, "Todo es un objeto". La otra hace lo mismo, pero tiene diferencias sutiles e importantes.

Ésta es la sintaxis básica de `from módulo import`:

```
from UserDict import UserDict
```

Esto es similar a la sintaxis de `import módulo` que conoce y adora, pero con una diferencia importante: los atributos y métodos de lo módulo importado `types` se sitúan directamente en el espacio de nombres local, de manera que están disponibles directamente, sin necesidad de acceder a ellos mediante el nombre del módulo. Puede importar elementos individuales o usar `from módulo import *` para importarlo todo.

`from módulo import *` en Python es como `use módulo` en Perl; `import módulo` en Python es como `require módulo` en Perl.

`from módulo import *` en Python es como `import módulo.*` en Java; `import módulo` en Python es como `import módulo` en Java.

Ejemplo 5.2. `import módulo` frente a `from módulo import`

```
>>> import types
>>> types.FunctionType             ❶
<type 'function'>
>>> FunctionType                   ❷
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
NameError: There is no variable named 'FunctionType'
>>> from types import FunctionType ❸
>>> FunctionType                   ❹
<type 'function'>
```

❶ El módulo `types` no contiene métodos; sino sólo atributos para cada tipo de objeto en Python. Observe que el atributo, `FunctionType`, debe cualificarse usando el nombre del módulo, `types`.

❷ `FunctionType` no ha sido definida en este espacio de nombres; existe sólo en el contexto de `types`.

❸ Esta sintaxis importa el atributo `FunctionType` del módulo `types` directamente en el espacio de nombres local.

❹ Ahora se puede acceder directamente a `FunctionType`, sin hacer referencia a `types`.

¿Cuándo debería usar `from módulo import`?

- Si quiere acceder a atributos y métodos a menudo y no quiere escribir el nombre del módulo una y otra vez, utilice `from módulo import`.
- Si desea importar de forma selectiva algunos atributos y métodos pero no otros, use `from módulo import`.
- Si el módulo contiene atributos o funciones con el mismo nombre que su propio módulo, deberá usar `import módulo` para evitar conflicto de nombres.

Aparte de esto, es sólo cuestión de estilo, y verá código en Python escrito de ambas maneras.

Utilice `from module import *` lo menos posible, porque hace difícil determinar de dónde vino una función o atributo en particular, y complica más la depuración y el refactorizado.

Lecturas complementarias sobre técnicas de importar módulos

- eff-bot tiene más cosas que decir sobre `import módulo` frente a `from módulo import`.
- El *Tutorial de Python* comenta técnicas avanzadas de importación, incluyendo `from módulo import *`.

5.3. Definición de clases

Python está completamente orientado a objetos: puede definir sus propias clases, heredar de las que usted defina o de las incorporadas en el lenguaje, e instanciar las clases que haya definido.

Definir una clase en Python es simple. Como con las funciones, no hay una definición interfaz separada. Simplemente defina la clase y empiece a escribir código. Una clase de Python empieza con la palabra reservada `class`, seguida de su nombre. Técnicamente, esto es todo lo que necesita, ya que la clase no tiene por qué heredar de ninguna otra.

Ejemplo 5.3. La clase más simple en Python

```
class Loaf: ❶
```

```
    pass      ❷ ❸
```

❶ El nombre de esta clase es `Loag`, y no hereda de ninguna otra. Los nombres de clases suelen comenzar en mayúscula, `CadaPalabraAsí`, pero esto es sólo una convención, no un requisito.

❷ Esta clase no define métodos ni atributos, pero sintácticamente, hace falta que exista alguna cosa en la definición, de manera que usamos `pass`. Ésta es una palabra reservada de Python que simplemente significa "múevanse, nada que ver aquí". Es una sentencia que no hace nada, y es un buen sustituto de funciones o clases modelo, que no hacen nada.

❸ Probablemente haya adivinado esto, pero todo dentro de una clase va sangrado, igual que el código dentro de una función, sentencia `if`, bucle `for`, *etc.*. La primera cosa que no esté sangrada, no pertenece a la clase.

La sentencia `pass` de Python es como unas llaves vacías (`{}`) en Java o C.

Por supuesto, siendo realistas, la mayoría de las clases heredarán de alguna otra, y definirán sus propios métodos y atributos. Pero como acabamos de ver, no hay nada que una función deba tener, aparte de su nombre. En particular, los programadores de C++ encontrarán extraño que las clases de Python no tengan constructores o destructores explícitos. Las clases de Python tienen algo similar a un constructor: el método `__init__`.

Ejemplo 5.4. Definición de la clase `FileInfo`

```
from UserDict import UserDict

class FileInfo(UserDict): ❶
```

❶ En Python, el ancestro de una clase se lista simplemente entre paréntesis inmediatamente del nombre de la clase. Así que la clase `FileInfo` hereda de la clase `UserDict` (que importamos del módulo `UserDict`). `UserDict` es una clase que actúa como un diccionario, permitiéndole derivar el tipo de datos diccionario y añadir comportamientos a su gusto (existen las clases similares

`UserList` y `UserString` que le permiten derivar listas y cadenas). Hay un poco de magia negra tras todo esto, que demistificaré más adelante en este capítulo cuando exploremos la clase `UserDict` en más profundidad.

En Python, el ancestro de una clase se lista entre paréntesis inmediatamente tras el nombre de la clase. No hay palabras reservadas especiales como `extends` en Java.

Python admite herencia múltiple. En los paréntesis que siguen al nombre de la clase, puede enumerar tantas clases ancestro como desee, separadas por comas.

5.3.1. Inicialización y programación de clases

Este ejemplo muestra la inicialización de la clase `FileInfo` usando el método `__init__`.

Ejemplo 5.5. Inicialización de la clase `FileInfo`

```
class FileInfo(UserDict):
    "store file metadata"                  ❶
    def __init__(self, filename=None): ❷ ❸ ❹
```

❶ Las clases pueden también (y deberían) tener `cadenas de documentación`, al igual que los módulos y funciones.

❷ `__init__` se llama inmediatamente tras crear una instancia de la clase. Sería tentador pero incorrecto denominar a esto el constructor de la clase. Es tentador porque parece igual a un constructor (por convención, `__init__` es el primer método definido para la clase), actúa como uno (es el primer pedazo de código que se ejecuta en una instancia de la clase recién creada), e incluso suena como una ("init" ciertamente sugiere una naturaleza constructórica). Incorrecto, porque el objeto ya ha sido construido para cuando se llama a `__init__`, y ya tiene una referencia válida a la nueva instancia de la clase. Pero `__init__` es lo más parecido a un constructor que

va a encotnrar en Python, y cumple el mismo papel.

❸ El primer método de cada método de clase, incluido `__init__`, es siempre una referencia a la instancia actual de la clase. Por convención, este argumento siempre se denomina `self`. En el método `__init__`, `self` se refiere al objeto recién creado; en otros métodos de la clase, se refiere a la instancia cuyo método ha sido llamado. Aunque necesita especificar `self` de forma explícita cuando define el método, *no* se especifica al invocar el método; Python lo añadirá de forma automática.

❹ Los métodos `__init__` pueden tomar cualquier cantidad de argumentos, e igual que las funciones, éstos pueden definirse con valores por defecto, haciéndoles opcionales para quien invoca. En este caso, `filename` tiene el valor por omisión de `None`, que es el valor nulo de Python.

Por convención, el primer argumento de cualquier clase de Python (la referencia a la instancia) se denomina `self`. Este argumento cumple el papel de la palabra reservada `this` en C++ o Java, pero `self` no es una palabra reservada en Python, sino una mera convención. De todas maneras, por favor no use otro nombre sino `self`; es una convención muy extendida.

Ejemplo 5.6. Programar la clase `FileInfo`

```
class FileInfo(UserDict):
    "store file metadata"
    def __init__(self, filename=None):
        UserDict.__init__(self)        ❶
        self["name"] = filename        ❷
                                       ❸
```

❶ Algunos objetos pseudo-orientados a objeto como Powerbuilder tienen un concepto de "extender" constructores y otros eventos, para los que el método ancestro se llama de forma automática antes de que se ejecute el método del descendiente. Python no hace esto; siempre ha de llamar de forma explícita a los métodos de los ancestros.

❷ Le dije antes que esta clase actúa como un diccionario, y aquí está la primera señal de ello. Está asignando el argumento `filename` como valor de la clave `name` de este objeto.

❸ Advierta que el método `__init__` nunca devuelve un valor.

5.3.2. Saber cuándo usar `self` e `__init__`

Cuando defina los métodos de su clase, *debe* enumerar `self` de forma explícita como el primer argumento de cada método, incluido `__init__`. Cuando llame a un método de una clase ancestra desde dentro de la clase, *debe* incluir el argumento `self`. Pero cuando invoque al método de su clase desde fuera, no debe especificar nada para el argumento `self`; evítelo completamente, y Python añadirá automáticamente la referencia a la instancia. Soy consciente de que esto es confuso al principio; no es realmente inconsistente, pero puede parecer inconsistente debido a que se basa en una distinción (entre métodos *bound* y *unbound*) que aún no conoce.

¡Vaya! Me doy cuenta de que es mucho por absorber, pero le acabará pillando el truco. Todas las clases de Python funcionan de la misma manera, de manera que cuando aprenda una, las habrá aprendido todas. Si se olvida de algo, recuerde esto, porque prometo que se lo tropezará:

Los métodos `__init__` son opcionales, pero cuando define uno, debe recordar llamar explícitamente al método `__init__` del ancestro (si define uno). Suele ocurrir que siempre que un descendiente quiera extender el comportamiento de un ancestro, el método descendiente deba llamar al del ancestro en el momento adecuado, con los argumentos adecuados.

Lecturas complementarias sobre clases de Python

- *Learning to Program* contiene una introducción a las clases más suave.
- *How to Think Like a Computer Scientist* muestra cómo usar clases pra modelar tipos de datos compuestos.

- El *Tutorial de Python* da un vistazo en profundidad a clases, espacios de nombres y herencia.
- La Python Knowledge Base responde preguntas comunes sobre clases.

5.4. Instanciación de clases

La instanciación de clases en Python es trivial. Para instanciar una clase, simplemente invoque a la clase como si fuera una función, pasando los argumentos que defina el método `__init__`. El valor de retorno será el objeto recién creado.

Ejemplo 5.7. Creación de una instancia de `FileInfo`

```
>>> import fileinfo
>>> f = fileinfo.FileInfo("/music/_singles/kairo.mp3") ❶
>>> f.__class__                                        ❷
<class fileinfo.FileInfo at 010EC204>
>>> f.__doc__                                          ❸
'store file metadata'
>>> f                                                  ❹
{'name': '/music/_singles/kairo.mp3'}
```

❶ Está creando una instancia de la clase `FileInfo` (definida en el módulo `fileinfo`) y asignando la instancia recién creada a la variable `f`. Está pasando un parámetro, `/music/_singles/kairo.mp3`, que será el argumento `filename` en el método `__init__` de `FileInfo`

❷ Cada instancia de una clase contiene un atributo incorporado, `__class__`, que es el objeto de su clase (observer que la representación de esto incluye la dirección física de la instancia en mi máquina; la de usted diferirá). Los programadores de Java estarán familiarizados con la clase `Class`, que contiene métodoscomo `getName` y `getSuperclass` para obtener información de metadatos de un objeto. En Python este tipo de metadatos se obtienen directamente del propio objeto mediante atributos como `__class__`,

`__name__`, y `__bases__`.

❸ Puede acceder a la `cadena de documentación` de la instancia igual que con una función o módulo. Todas las instancias de una clase comparten la misma `cadena de documentación`.

❹ ¿Recuerda cuando el método `__init__` asignó su argumento `filename` a `self["name"]`? Bien, aquí está el resultado. Los argumentos que pasa cuando crea la instancia de la clase se envían junco al método `__init__` (junto con la referencia al objeto, `self`, que Python añade por su cuenta).

En Python, simplemente invocamos a una clase como si fuese una función para crear una nueva instancia de la clase. No hay operador `new` explícito como en C++ o Java.

5.4.1. Recolección de basura

Si crear instancias nuevas es sencillo, destruirlas lo es más. En general, no hay necesidad de liberar de forma explícita las instancias, porque se eliminan automáticamente cuando las variables a las que se asignan salen de ámbito. Son raras las pérdidas de memoria en Python.

Ejemplo 5.8. Intento de implementar una pérdida de memoria

```
>>> def leakmem():
...     f = fileinfo.FileInfo('/music/_singles/kairo.mp3') ❶
...
>>> for i in range(100):
...     leakmem()                                          ❷
```

❶ Cada vez que se invoca la función `leakmem`, se crea una instancia de `FileInfo`, asignándola a la variable `f`, que es local a la función. Entonces la función termina sin liberar `f`, de manera que esperaríamos tener aquí una fuga de memoria (*memory leak*), pero estaríamos equivocados. Cuando termina la función, se dice que la variable `f` sale de ámbito. En este momento, ya no hay referencias a la instancia recién creada de `FileInfo` (ya que no la hemos

asignado a ninguna otra cosa aparte de `f`), así que Python la destruye por nosotros.

❷ No importa cuántas veces llamemos a la función `leakmem`, que nunca perderá memoria, porque cada vez, Python destruirá la instancia de `FileInfo` recién creada antes de salir de `leakmem`.

El término técnico para esta forma de recolección de basura es "conteo de referencias" (*reference counting*). Python mantiene una lista de referencias a cada instancia creada. En el ejemplo anterior, sólo hay una referencia a la instancia de `FileInfo`: la variable local `f`. Cuando termina la función, la variable `f` se encuentra fuera de ámbito, así que la cuenta de referencias cae a `0`, y Python destruye la instancia de forma automática.

En versiones anterios de Python, existían situaciones donde la cuenta de referencias fallaba, y Python no podía hacer limpieza tras de usted. Si creaba dos instancias que se referenciaban una a otra (por ejemplo, una lista doblemente enlazada, donde cada nodo tiene un puntero hacia el anterior y el siguiente en la lista), ninguna instancia se hubiera destruido automáticamente, porque Python (correctamente) creería que siempre hay una referencia a cada instancia. Python 2.0 tiene una forma adicional de recolección de basuras llamada "*mark-and-sweep*" que es lo suficientemente inteligente como para darse cuenta de este cerrojo virtual y elimina correctamente referencias circulares.

Habiendo estudiado filosofía, me resulta chocante pensar que las cosas desaparecen cuando uno no las está mirando, pero eso es exactamente lo que sucede en Python. En general, puede simplemente olvidarse de la gestión de memoria, y dejarle a Python que haga la limpieza.

Lecturas complementarias sobre recolección de basuras

- La *Referencia de bibliotecas de Python* habla sobre atributos incorporados como `__class__`.

- La *Referencia de bibliotecas de Python* documenta el módulo >gc, que le da control a bajo nivel sobre la recolección de basura de Python.

5.5. Exploración de `UserDict`: Una clase cápsula

Como ha podido ver, `FileInfo` es una clase que actúa como un diccionario. Para explorar esto un poco más, veamos la clase `UserDict` del módulo `UserDict`, que es el ancestro de la clase `FileInfo` . No es nada especial; la clase está escrita en Python y almacenada en un fichero `.py`, igual que cualquier otro código Python. En particular, está almacenada en el directorio `lib` de su instalación de Python.

En el IDE ActivePython en Windows, puede abrir rápidamente cualquier módulo de su ruta de bibliotecas mediante File->Locate... (**Ctrl-L**).

Ejemplo 5.9. Definición de la clase `UserDict`

```
class UserDict:                                 ❶
    def __init__(self, dict=None):              ❷
        self.data = {}                          ❸
        if dict is not None: self.update(dict) ❹ ❺
```

❶ Observe que `UserDict` es una clase base, que no hereda de ninguna otra.

❷ Éste es el método `__init__` que sustituimos en la clase `FileInfo`. Advierta que la lista de argumentos en la clase ancestro es diferente que en la descendiente. No pasa nada; cada subclase puede tener su propio juego de argumentos, siempre que llame a su ancestro de la manera correcta. Aquí la clase ancestro tiene una manera de definir valores iniciales (pasando un diccionario en el argumento `dict`) que no usa `FileInfo`.

❸ Python admite atributos de datos (llamados "variables de instancia" en Java y Powerbuilder, y "variables miembro" en C++). En este caso, cada instancia de `UserDict` tendrá un atributo de datos `data`. Para hacer referencia a este

atributo desde código que esté fuera de la clase, debe calificarlo con el nombre de la instancia, `instancia.data`, de la misma manera que calificaría una función con el nombre de su módulo. Para hacer referencia a atributos de datos desde dentro de la clase, use `self` como calificador. Por convención, todos los atributos de datos se inicializan en el método `__init__` con valores razonables. Sin embargo, esto no es un requisito, ya que los atributos, al igual que las variables locales, comienzan a existir cuando se les asigna su primer valor.

4. El método `update` es un duplicador de diccionarios: copia todas las claves y valores de un diccionario dentro de otro. Esto *no* borra antes los valores previos del diccionario modificado; si el diccionario objetivo ya contenía algunas claves, las que coincidan con el diccionario fuente serán modificadas, pero no se tocará las otras. Piense en `update` como una función de mezcla, no de copia.

5. Ésta es una sintaxis que puede no haber visto antes (no la he usado en los ejemplos de este libro). Hay una sentencia `if`, pero en lugar de un bloque sangrado en la siguiente líena, la sentencia está en una sola línea, tras los dos puntos. Esta sintaxis es perfectamente válida, y es sólo un atajo que puede usar cuando el bloque conste de sólo una sentencia (es como especificar una sola sentencia sin llaves en C++). Puede usar esta sintaxis, o sangrar el código en las siguientes líneas, pero no puede hacer ambas cosas en el mismo bloque.

Java y Powerbuilder admiten la sobrecarga de funciones por lista de argumentos, es decir una clase puede tener varios métodos con el mismo nombre, pero con argumentos en distinta cantidad, o de distinto tipo. Otros lenguajes (notablemente PL/SQL) incluso admiten sobrecarga de funciones por nombre de argumento; es decir una clase puede tener varios métodos con el mismo nombre y número de argumentos de incluso el mismo tipo, pero con diferentes nombres de argumento. Python no admite ninguno de estos casos; no hay forma de sobrecarga de funciones. Los métodos se

definen sólo por su nombre, y hay un único método por clase con un nombre dado. De manera que si una clase sucesora tiene un método `__init__`, *siempre* sustituye al método `__init__` de su ancestro, incluso si éste lo define con una lista de argumentos diferentes. Y se aplica lo mismo a cualquier otro método.

Guido, el autor original de Python, explica el reemplazo de métodos así: "Las clases derivadas pueden reemplazar los métodos de sus clases base. Dado que los métodos no tienen privilegios especiales para llamar a otros métodos del mismo objeto, un método de una clase base que llama a otro método definido en la misma clase base, puede en realidad estar llamando a un método de una clase derivada que la reemplaza (para programadores de C++: todos los métodos de Python son virtuales a los efectos)". Si esto no tiene sentido para usted (a mí me confunde sobremanera), ignórelo. Sólo pensaba que debía comentarlo.

Asigne siempre un valor inicial a todos los atributos de datos de una instancia en el método `__init__`. Le quitará horas de depuración más adelante, en busca de excepciones `AttributeError` debido a que está haciendo referencia a atributos sin inicializar (y por tanto inexistentes).

Ejemplo 5.10. Métodos normales de `UserDict`

```
def clear(self): self.data.clear()          ❶
def copy(self):                             ❷
    if self.__class__ is UserDict:          ❸
        return UserDict(self.data)
    import copy                             ❹
    return copy.copy(self)
def keys(self): return self.data.keys()     ❺
def items(self): return self.data.items()
def values(self): return self.data.values()
```

❶ `clear` es un método normal de clase; está disponible públicamente para que cualquiera lo invoque en cualquier momento. Observe que `clear`, igual que

todos los métodos de una clase, tiene `self` como primer argumento (recuerde que no ha de incluir `self` al invocar el método; Python lo hace por usted). Fíjese también en la técnica básica de esta clase *wrapper*: almacena un diccionario real (`data`) como atributo, define todos los métodos que tiene un diccionario real, y cada uno lo redirige al correspondiente en el diccionario (en caso de que lo haya olvidado, el método `clear` de un diccionario borra todas sus claves y sus valores asociados).

❷ El método `copy` de un diccionario real devuelve un nuevo diccionario que es un duplicado exacto del original (los mismos pares clave-valor). Pero `UserDict` no se puede redirigir simplemente a `self.data.copy`, porque el método devuelve un diccionario real, y lo que queremos es devolver una nueva instancia que sea de la misma clase que `self`.

❸ Puede usar el atributo `__class__` para ver si `self` es un `UserDict`; si lo es, perfecto, porque sabemos cómo copiar un `UserDict`: simplemente creamos un nuevo `UserDict` y le proporcionamos el diccionario real que mantenemos en `self.data`. Entonces devolvemos de inmediato el nuevo `UserDict` sin llegar siquiera al `import copy` de la siguiente línea.

❹ Si `self.__class__` no es un `UserDict`, entonces `self` debe ser alguna subclase de `UserDict` (por ejemplo `FileInfo`), en cuyo caso la vida se hace un poco más dura. `UserDict` no sabe cómo hacer una copia exacta de uno de sus descendientes; podría haber, por ejemplo, otros atributos de datos definidos en la subclase, así que necesitaríamos iterar sobre ellos y asegurarnos de que los copiamos todos. Por suerte, Python tiene un módulo que hace exactamente esto, y se llama `copy`. No entraré en detalles (aunque es un módulo muy interesante, por si alguna vez se siente inclinado a sumergirse en él usted mismo). Baste decir que `copy` puede copiar objetos arbitrarios de Python, y que para eso lo estamos usando aquí.

❺ El resto de los métodos son triviales, y redireccionan las llamadas a los métodos incorporados en `self.data`.

En las versiones de Python previas a la 2.20, no podía derivar directamente

tipos de datos internos como cadenas, listas y diccionarios. Para compensarlo, Python proporcionaba clases encapsulantes que imitaban el comportamiento de estos tipos: `UserString`, `UserList`, y `UserDict`. Usando una combinación de métodos normales y especiales, la clase `UserDict` hace una excelente imitación de un diccionario. En Python 2.2 y posteriores, puede hacer que una clase herede directamente de tipos incorporados como `dict`. Muestro esto en los ejemplos que acompañan al libro, en `fileinfo_fromdict.py`.

En Python, puede heredar directamente del tipo incorporado `dict`, como se muestra en este ejemplo. Hay tres diferencias si lo comparamos con la versión que usa `UserDict`.

Ejemplo 5.11. Herencia directa del tipo de datos `dict`

```
class FileInfo(dict):                    ❶
    "store file metadata"
    def __init__(self, filename=None): ❷
        self["name"] = filename
```

❶ La primera diferencia es que no necesita importar el módulo `UserDict`, ya que `dict` es un tipo de dato incorportado y siempre está disponible. La segunda es que hereda directamente de `dict`, en lugar de `UserDict.UserDict`.

❷ La tercera diferencia es sutil pero importante. Debido a la manera en que funciona internamente `UserDict`, precisa de que usted llame manualmente a su método `__init__` para inicializar correctamente sus estructuras internas. `dict` no funciona de esta manera; no es una cápsula, y no requiere ninguna inicialización explícita.

Lecturas complementarias sobre `UserDict`

- La*Referencia de bibliotecas de Python* documenta el módulo `UserDict` y el módulo `copy`.

5.6. Métodos de clase especiales

Además de los métodos normales, existe un cierto número de métodos especiales que pueden definir las clases de Python. En lugar de llamarlos directamente desde su código (como los métodos normales), los especiales los invoca Python por usted en circunstancias particulares o cuando se use una sintaxis específica.

Como pudo ver en la sección anterior, los métodos normales van mucho más allá de simplemente actuar como cápsula de un diccionario en una clase. Pero los métodos normales por sí solos no son suficientes, porque hay muchas cosas que puede hacer con diccionarios aparte de llamar a sus métodos. Para empezar, en lugar de usar get y set para trabajar con los elmentos, puede hacerlo con una sintaxis que no incluye una invocación explícita a métodos. Aquí es donde entran los métodos especiales de clase: proporcionan una manera de convertir la sintaxis-que-no-llama-a-métodos en llamadas a métodos.

5.6.1. Consultar y modificar elementos

Ejemplo 5.12. El método especial `__getitem__`

```
    def __getitem__(self, key): return self.data[key]

>>> f = fileinfo.FileInfo("/music/_singles/kairo.mp3")

>>> f

{'name':'/music/_singles/kairo.mp3'}

>>> f.__getitem__("name") ❶

'/music/_singles/kairo.mp3'

>>> f["name"]             ❷
```

```
'/music/_singles/kairo.mp3'
```

❶ El método especial `__getitem__` parece bastante sencillo. Igual que los métodos normales `clear`, `keys` y `values`, simplemente se dirige al diccionario para devolver su valor. Pero, ¿cómo se le invoca? Bien, puede llamar a `__getitem__` diretamente, pero en la práctica no es lo que hará; lo hago aquí para mostrarle cómo trabaja. La manera adecuada de usar `__getitem__` es hacer que Python lo llame por usted.

❷ Se parece a la sintaxis que usaríamos para obtener un valor del diccionario, y de hecho devuelve el valor esperado. Pero he aquí el eslabón perdido: internamente, Python ha convertido esta sintaxis en una llamada al método `f.__getitem__("name")`. Por eso es `__getitem__` un método especial; no sólo puede invocarlo usted, sino que puede hacer que Python lo invoque usando la sintaxis adecuada.

Por supuesto, Python tiene un método especial `__setitem__` que acompaña a `__getitem__`, como mostramos en el siguiente ejemplo.

Ejemplo 5.13. El método especial `__setitem__`

```
    def __setitem__(self, key, item): self.data[key] = item

>>> f
{'name':'/music/_singles/kairo.mp3'}
>>> f.__setitem__("genre", 31) ❶
>>> f
{'name':'/music/_singles/kairo.mp3', 'genre':31}
>>> f["genre"] = 32            ❷
>>> f
{'name':'/music/_singles/kairo.mp3', 'genre':32}
```

❶ Igual que con el método `__getitem__`, `__setitem__` simplemente delega en el diccionario real `self.data` para hacer su trabajo. E igual que `__getitem__`, normalmente no lo invocará de forma ordinaria de esta manera; Python llama a `__setitem__` cuando usa la sintaxis correcta.

❷ Esto parece sintaxis normal de diccionario, excepto que por supuesto `f` realmente es una clase que intenta por todos los medios aparentar ser un diccionario, y `__setitem__` es parte esencial de la mascarada. Esta línea de código en realidad invoca a `f.__setitem__("genre", 32)` tras las cortinas.

`__setitem__` es un método especial de clase debido a que lo invocan por usted, pero sigue siendo un método de clase. Con la misma facilidad con que definió el método `__setitem__` en `UserDict`, puede redefinirlo en la clase descendiente para reemplezar el método del ancestro. Esto le permite definir clases que actúan como diccionarios de cierta manera pero con su propio comportamiento más allá del de un diccionario estándar.

Este concepto es la base de todo el *framework* que está estudiando en este capítulo. Cada tipo de fichero puede tener una clase de manejo que sepa cómo acceder a los metadatos de ese tipo particular de fichero. Una vez se conocen ciertos atributos (como el nombre del fichero y su emplazamiento), la clase manejadora sabe cómo derivar otros atributos de forma automática. Esto se hace reemplazando el método `__setitem__`, buscando ciertas claves, y añadiendo procesamiento adicional cuando se las encuentra.

Por ejemplo, `MP3FileInfo` desciende de `FileInfo`. Cuando se asigna el `name` de una `MP3FileInfo`, no nos limitamos a asignar el valor de la clave `name` (como hace el ancestro `FileInfo`); también se busca en el propio fichero etiquetas MP3 y da valor a todo un juego de claves. El siguiente ejemplo le muestra cómo funciona esto.

Ejemplo 5.14. Reemplazo de `__setitem__` en `MP3FileInfo`

```
    def __setitem__(self, key, item):          ❶
        if key == "name" and item:             ❷
```

```
            self.__parse(item)                  ❸
        FileInfo.__setitem__(self, key, item) ❹
```

❶ Observe que este método `__setitem__` se define exactamente de la misma manera que en el método ancestro. Esto es importante, ya que Python llamará al método por usted, y espera que esté definido con un cierto número de argumentos (técnicamente, los nombres de los argumentos no importan; sólo su número).

❷ Aquí está el quid de toda la clase `MP3FileInfo`: si asignamos un valor a la clave `name`, queremos que se hagan algunas cosas extra.

❸ El procesamiento extra que se hace para los `name` se encapsula en el método `__parse`. Éste es otro método de clase definido en `MP3FileInfo`, y cuando lo invoca, lo hace calificándolo con `self`. Una llamada simplemente a `__parse` hará que se busque una función normal definida fuera de la clase, que no es lo que deseamos. Invocar a `self.__parse` buscará un método definido en la clase. Esto no es nuevo; hemos hecho referencia a atributos de datos de la misma manera.

❹ Tras hacer este procesamiento extra, querremos llamar al método del ancestro. Recuerde que esto no lo hace Python por usted nunca; debe hacerlo manualmente. Fíjese en que está llamado al ancestro inmediato, `FileInfo`, incluso aunque éste no define un método `__setitem__`. Esto es correcto, ya que Python escalará en el árbol genealógico hasta que encuentre una clase con el método que busca, de manera que esta línea de código acabará encontrando e invocando al `__setitem__` definido en `UserDict`.

Cuando se accede a atributos de datos dentro de una clase, necesitamos calificar el nombre del atributo: `self.`*`atributo`*. Cuando llamamos a otros métodos dentro de una clase, necesitamos calificar el nombre del método: `self.`*`método`*.

Ejemplo 5.15. Dar valor al `name` de una `MP3FileInfo`

```
>>> import fileinfo
>>> mp3file = fileinfo.MP3FileInfo()                    ❶
```

```
>>> mp3file
{'name':None}
>>> mp3file["name"] = "/music/_singles/kairo.mp3"      ❷
>>> mp3file
{'album': 'Rave Mix', 'artist': '***DJ MARY-JANE***', 'genre': 31,
'title': 'KAIRO****THE BEST GOA', 'name': '/music/_singles/kairo.mp3',
'year': '2000', 'comment': 'http://mp3.com/DJMARYJANE'}
>>> mp3file["name"] = "/music/_singles/sidewinder.mp3" ❸
>>> mp3file
{'album': '', 'artist': 'The Cynic Project', 'genre': 18, 'title':
'Sidewinder',
'name': '/music/_singles/sidewinder.mp3', 'year': '2000',
'comment': 'http://mp3.com/cynicproject'}
```

❶ Primero, creamos una instancia de `MP3FileInfo`, sin pasarle el nombre de un fichero (podemos hacerlo así porque el argumento `filename` del método `__init__` es opcional). Como `MP3FileInfo` no tiene método `__init__` propio, Python escala en el árbol genealógico y encuentra el método `__init__` de `FileInfo`. Este método `__init__` llama manualmente al `__init__` de `UserDict` y establece el valor de la clave `name` como `filename`, que es `None`, ya que no pasó un nombre de fichero. Por tanto, `mp3file` aparenta ser un diccionario con una clave, `name`, cuyo valor es `None`.

❷ Ahora empieza lo divertido. Poner un valor a la clave `name` de `mp3file` dispara el método `__setitem__` de `MP3FileInfo` (no `UserDict`), lo que se da cuenta de que está dándole a `name` un valor real e invoca a `self.__parse`. Aunque no hemos mirado aún el método `__parse`, puede ver por la salida que establece varias otras claves: `album`, `artist`, `genre`, `title`, `year`, y `comment`.

❸ Modificar la clave `name` reproducirá este proceso: Python llama a `__setitem__`, que a su vez llama a `self.__parse`, que pone valor a las otras claves.

5.7. Métodos especiales avanzados

Python tiene más métodos especiales aparte de `__getitem__` y `__setitem__`. Algunos de ellos le permiten emular funcionalidad que puede que aún ni conozca.

Este ejemplo muestra algunos de los otros métodos especiales de `UserDict`.

Ejemplo 5.16. Más métodos especiales de `UserDict`

```
def __repr__(self): return repr(self.data)      ❶
def __cmp__(self, dict):                        ❷
    if isinstance(dict, UserDict):
        return cmp(self.data, dict.data)
    else:
        return cmp(self.data, dict)
def __len__(self): return len(self.data)        ❸
def __delitem__(self, key): del self.data[key] ❹
```

❶ `__repr__` es un método especial que se invoca cuando llamamos a `repr(`*`instancia`*`)`. La función `repr` viene incorporada y devuelve una representación textual de un objeto. Funciona sobre cualquier objeto, no sólo instancias de clases. Ya está familiarizado con `repr` aunque no lo supiera. En la ventana interactiva, cuando escribe un nombre de variable y pulsa **INTRO**, Python usa `repr` para mostrar su valor. Cree un diccionario `d` con algunos datos y entonces haga `print repr(d)` para comprobarlo usted mismo.

❷ `__cmp__` se invoca cuando compara instancias de clases. En general, puede ocmpara dos objetos cualquiera de Python, no sólo instancias de clases, usando ==. Hay reglas que definen cuándo se consideran iguales dos tipos de datos incorporados; por ejemplo, los diccionarios son iguales cuando tienen todas las mismas claves y valores, y las cadenas son iguales cuando tienen la misma longitud y contienen la misma secuencia de caracteres. Para instancias de clases, podemos definir el método `__cmp__` y programar la lógica de la comparación nosotros mismos, y entonces puede usar == para comparar instancias de sus clases y Python invocará a su método especial `__cmp__` por usted.

❸ `__len__` se invoca cuando llama a `len(`*`instancia`*`)`. La función incorporada `len` devuelve la longitud de un objeto. Funciona sobre cualquier objeto del que se pueda pensar razonablemente que tiene longitud. La `len` de una cadena es su número de caracteres; la `len` de un diccionario es su número de

claves; la `len` de una lista o tupla es su número de elementos. Para instancias de clases; defina el método `__len__` y programe usted mismo el cálculo de la longitud, y luego llame a `len(instancia)` para que Python invoque a su método epsecial `__len__`.

4. `__delitem__` se invoca cuando llama a `del instancia[clave]`, que como puede que recuerde es la manera de eliminar elementos individuales de un diccionario.. Cuando use `del` sobre una instancia de clase, Python invocará al método especial `__delitem__`.

En Java, determinamos si dos variables de cadena referencian la misma posición física de memoria usando `str1 == str2`. A esto se le denomina *identidad de objetos*, y en Python se escribe así: `str1 is str2`. Para comparar valores de cadenas en Java, usaríamos`str1.equals(str2)`; en Python, usaríamos `str1 == str2`. Los programadores de Java a los que se les haya enseñado a creer que el mundo es un lugar mejor porque == en Java compara la identidad en lugar del valor pueden tener dificultades ajustándose a la falta de este tipo de "*gotchas*" en Python.

A estas alturas, puede estar pensando, "Todo este trabajo sólo para hacer algo en una clase que podría hacer con un tipo de datos incorporado". Y es cierto que la vida sería más fácil (e innecesaria toda la clase `UserDict`) si pudiéramos heredar de tipos de datos incorporados como un diccionario. Pero incluso aunque pudiera, los métodos especiales seguirían siendo útiles, porque se pueden usar en cualquier clase, no sólo en las que envuelven a otras como `UserDict`.

Los métodos especiales implican que *cualquier clase* puede almacenar pares clave/valor como un diccionario. *Cualquier clase* puede actuar como una secuencia, simplemente definiendo el método `__getitem__`. Cualquier clase que defina el método `__cmp__` puede compararse con `==`. Y si su clase representa algo que tenga una longitud, no defina un método `GetLength`; defina el método `__len__` y use `len(instancia)`.

Mientras que otros lenguajes orientados a objeto sólo le permitirán definir el modelo físico de un objeto ("este objeto tiene un método `GetLength`"), los métodos especiales de Python como `__len__` le permiten definir el modelo lógico de un objeto ("este objeto tiene una longitud").

Python tiene muchos otros métodos especiales. Hay todo un conjunto de ellos para hacer que las clases actúen como números, permitiéndole sumar, sustraer y otras operaciones aritméticas sobre instancias de clases (el ejemplo canónico es la clase que representa a los números complejos, con componentes real e imaginario). El método `__call__` permite que una clase actúe como una función, permitiéndole invocar directamente a una instancia de la clase. Y hay otros métodos especiales que permiten a las clases tener atributos de datos de sólo lectura o sólo escritura; hablaremos más sobre ellos en otros capítulos.

Lecturas complementarias sobre métodos especiales de clase

- La *Referencia del lenguaje Python* documenta todos los métodos especiales de clase.

5.8. Presentación de los atributos de clase

Ya conoce los atributos de datos, que son variables que pertenecen a una instancia específica de una clase. Python también admite atributos de clase, que son variables que pertenecen a la clase en sí.

Ejemplo 5.17. Presentación de los atributos de clase

```
class MP3FileInfo(FileInfo):
    "store ID3v1.0 MP3 tags"
    tagDataMap = {"title"   : (  3,  33, stripnulls),
                  "artist"  : ( 33,  63, stripnulls),
                  "album"   : ( 63,  93, stripnulls),
                  "year"    : ( 93,  97, stripnulls),
                  "comment" : ( 97, 126, stripnulls),
```

```
                  "genre"   : (127, 128, ord)}
>>> import fileinfo
>>> fileinfo.MP3FileInfo            ❶
<class fileinfo.MP3FileInfo at 01257FDC>
>>> fileinfo.MP3FileInfo.tagDataMap ❷
{'title': (3, 33, <function stripnulls at 0260C8D4>),
'genre': (127, 128, <built-in function ord>),
'artist': (33, 63, <function stripnulls at 0260C8D4>),
'year': (93, 97, <function stripnulls at 0260C8D4>),
'comment': (97, 126, <function stripnulls at 0260C8D4>),
'album': (63, 93, <function stripnulls at 0260C8D4>)}
>>> m = fileinfo.MP3FileInfo()      ❸
>>> m.tagDataMap
{'title': (3, 33, <function stripnulls at 0260C8D4>),
'genre': (127, 128, <built-in function ord>),
'artist': (33, 63, <function stripnulls at 0260C8D4>),
'year': (93, 97, <function stripnulls at 0260C8D4>),
'comment': (97, 126, <function stripnulls at 0260C8D4>),
'album': (63, 93, <function stripnulls at 0260C8D4>)}
```

❶ `MP3FileInfo` es la clase en sí, no una instancia en particular de la clase.

❷ `tagDataMap` es un atributo de la clase: literalmente. Está disponible antes de crear cualquier instancia de la clase.

❸ Los atributos de clase están disponibles tanto haciendo referencia directa a la clase como a cualquiera de sus instancias.

En Java, tanto las variables estáticas (llamadas atributos de clase en Python) como las variables de instancia (llamadas atributos de datos en Python) se declaran inmediatamente en la definición de la clase (unas con la palabra clave `static`, otras sin ella). En Python, sólo se pueden definir aquí los atributos de clase; los atributos de datos se definen en el método `__init__`.

Los atributos de clase se pueden usar como constantes de la clase (que es para lo que las usamos en `MP3FileInfo`), pero no son constantes realmente. También puede cambiarlas.

No hay constantes en Python. Todo puede cambiar si lo intenta con ahínco. Esto se ajusta a uno de los principios básicos de Python: los comportamientos inadecuados sólo deben desaconsejarse, no prohibirse. Si en realidad quiere cambiar el valor de `None`, puede hacerlo, pero no venga luego llorando si es imposible depurar su código.

Ejemplo 5.18. Modificación de atributos de clase

```
>>> class counter:
...     count = 0                      ❶
...     def __init__(self):
...         self.__class__.count += 1  ❷
...
>>> counter
<class __main__.counter at 010EAECC>
>>> counter.count                      ❸
0
>>> c = counter()
>>> c.count                            ❹
1
>>> counter.count
1
>>> d = counter()                      ❺
>>> d.count
2
>>> c.count
2
>>> counter.count
2
```

❶ `count` es un atributo de clase de la clase `counter`.

❷ `__class__` es un atributo incorporado de cada instancia de la clase (de toda clase). Es una referencia a la clase de la que es instancia `self` (en este caso, la clase `counter`).

❸ Debido a que `count` es un atributo de clase, está disponible por referencia directa a la clase, antes de haber creado instancias de ella.

❹ Crear una instancia de la clase llama al método `__init__`, que incrementa el atributo de la clase `count` en `1`. Esto afecta a la clase en sí, no sólo a la instancia recién creada.

❺ Crear una segunda instancia incrementará el atributo de clase `count` de nuevo. Advierta cómo la clase y sus instancias comparten el atributo de clase.

5.9. Funciones privadas

Como muchos lenguajes, Python tiene el concepto de elementos privados:

- Funciones privadas, que no se pueden invocar desde fuera de su módulo
- Métodos privados de clase, que no se pueden invocar desde fuera de su clase
- Atributos privados, a los que no se puede acceder desde fuera de su clase.

Al contrario que en muchos otros lenguajes, la privacidad de una función, método o atributo de Python viene determinada completamente por su nombre.

Si el nombre de una función, método de clase o atributo en Python empieza con (pero no termina en) dos caracteres guión bajo (_), es privado; todo lo demás es público. Python no tiene concept de métodos de clase *protected* (accesibles sólo desde su propia clase y descendientes). Los métodos de clase son privados (accesibles sólo desde su clase) o públicos (accesibles a cualquiera).

En `MP3FileInfo`, hay dos métodos: `__parse` y `__setitem__`. Como ya hemos hablado, `__setitem__` es un método especial; normalmente, lo llamará de forma indirecta usando la sintaxis de diccionarios en una instancia de clase, pero es pública, y podría llamarla directamente (incluso desde fuera del módulo `fileinfo`) si tuviera una poderosa razón. Sin embargo, `__parse` es privada, porque tiene dos guiones bajos al principio de su nombre.

En Python, todos los métodos especiales (como `__setitem__`) y atributos incorporados (como `__doc__`) siguen una convención estándar: empiezan y terminan con dos guiones bajos. No ponga a sus propios métodos ni atributos nombres así, porque sólo le confudirán a usted (y otros) más adelante.

Ejemplo 5.19. Intento de invocación a un método privado

```
>>> import fileinfo
>>> m = fileinfo.MP3FileInfo()
>>> m.__parse("/music/_singles/kairo.mp3") ❶
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'MP3FileInfo' instance has no attribute '__parse'
```

❶ Si intenta invocar a un método privado, Python lanzará una expresión un tanto confusa, diciendo que el método no existe. Por supuesto que existe, pero es privado, y por tanto no es accesible fuera de la clase.Hablando estrictamente, sí se puede acceder desde fuera de la clase a un método privado, sólo que no es *fácil*. Nada en Python es realmente privado; internamente, los nombres de los métodos y atributos privados se manipulan sobre la marcha para que parezca que son inaccesibles mediante sus nombres. Puede acceder al método `__parse` de `MP3FileInfo` mediante el nombre `_MP3FileInfo__parse`. Entienda esto como detalle interesante, pero prometa que nunca, nunca, lo hará en código de verdad. Los métodos privados lo son por alguna razón, pero como muchas otras cosas en Python, su privacidad es cuestión de convención, no forzada.

Lecturas complementarias sobre funciones privadas

- El *Tutorial de Python* expone los detalles de las variables privadas.

5.10. Resumen

Esto es todo lo en cuanto a trucos místicos. Verá una aplicación en el mundo real de métodos especiales de clase en Capítulo 12, que usa `getattr` para crear un *proxy* a un servicio remoto por web.

El siguiente capítulo continuará usando este ejemplo de código para explorar otros conceptos de Python, como las excepciones, los objetos de fichero, y los bucles `for`.

Antes de sumergirnos en el siguiente capítulo, asegúrese de que se siente cómodo haciendo las siguientes cosas:

- Importar módulos usando tanto `import módulo` como `from módulo import`
- Definir e instanciar clases
- Definir métodos `__init__` y otros métodos especiales de clase, y comprender cuándo se les invoca
- Derivar `UserDict` para definir clases que actúen como diccionarios
- Definir atributos de datos y atributos de clase, y comprender las diferencias entre ellos
- Definir atributos y métodos privados

Capítulo 6. Excepciones y gestión de ficheros

En este capítulo, se sumergirá en las excepciones, objetos de fichero, bucles `for`, y los módulos `os` y `sys`. Si ha usado excepciones en otros lenguaje de programación, puede leer por encima la primera sección para hacerse con la sintaxis de Python. Asegúrese de volver a rendir plenamente para la gestión de ficheros.

6.1. Gestión de excepciones

Como muchos otros lenguajes de programación, Python gestiona excepciones mediante bloques `try...except`.

Python utiliza `try...except` para gestionar las excepciones y `raise` para generarlas. Java y C++ usan `try...catch` para gestionarlas, y `throw` para generarlas.

Las excepciones las encuentra en todos lados en Python. Prácticamente cada módulo estándar de Python hace uso de ellas, y el propio Python las lanzará en muchas circunstancias diferentes. Ya las ha visto varias veces a lo largo de este libro.

- El acceso a una clave de diccionario que no existe provocará una excepción `KeyError`.
- Buscar en una lista un valor que no existe provocará una excepción `ValueError`.
- Invocar a un método que no existe provocará una excepción `AttributeError`.
- Referirse a una variable que no existe provocará una excepción `NameError`.
- Mezclar tipos de datos sin convertirlos previamente provocará una excepción `TypeError`.

En cada uno de estos casos, simplemente estábamos jugando con el IDE de Python: ocurría un error, se mostraba la excepción (dependiendo del IDE, quizá en un tono de rojo intencionadamente irritante), y eso era todo. A esto se le denomina una excepción *sin gestionar* (*unhandled*). Cuando se lanzó la excepción, no había código explícito que lo advirtiese e hiciera algo al respecto, de manera que subió hasta la superficie y disparó el comportamiento por omisión de Python, que es mostrar algo de información para depuración y terminar la ejecución. En el IDE esto no es mucho problema, pero si ocurre mientras está ejecutándose un verdadero programa escrito en Python, todo el programa terminaría con un gran derrape.

Una excepción no tiene por qué resultar en un completo desastre, sin embargo. Las excepciones, tras ser lanzadas, pueden ser *controladas*. Algunas veces ocurren excepciones debido a fallos en el programa (como acceder a una variable que no existe), pero a menudo, sucede por algo que podemos anticipar. Si abrimos un fichero, puede ser que no exista. Si conectamos a una base de datos, puede que no esté disponible, o quizá no introdujimos las credenciales de

seguridad correctas al acceder. Si sabe qué línea de código lanzará la excepción, podría gestionarla utilizando un bloque `try...except`.

Ejemplo 6.1. Apertura de un fichero inexistente

```
>>> fsock = open("/notthere", "r")      ❶
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
IOError: [Errno 2] No such file or directory: '/notthere'
>>> try:
...     fsock = open("/notthere")       ❷
... except IOError:                     ❸
...     print "The file does not exist, exiting gracefully"
... print "This line will always print" ❹
The file does not exist, exiting gracefully
This line will always print
```

❶ Podemos intentar abrir un fichero para leerlo usando la función incorporada `open` (más sobre `open` en la próxima sección). Pero el fichero no existe, y esto provoca una excepción `IOError`. Como no hemos proporcionado una comprobación explícita en busca de excepciones `IOError`, Python se limita a imprimir algo de información de depuración sobre lo que sucedió, y luego se detiene.

❷ Intentamos abrir de nuevo el mismo fichero inexistente, pero esta vez lo hacemos dentro de un bloque `try...except`.

❸ Cuando el método `open` lanza una excepción `IOError`, estamos preparados para ello. Le línea `except IOError:` captura la excepción y ejecuta su bloque de código, que en este caso simplemente imprime un mensaje de error más elegante.

❹ Una vez controlada la excepción, el proceso continúa de forma normal en la primera línea tras el bloque `try...except`. Observe que esta línea se imprime siempre, haya o no ocurrido la excepción. Si realmente tuviera un fichero llamado `notthere` en el directorio raíz, la llamada a `open` tendría éxito, se ignoraría la cláusula `except`, pero esta línea se ejecutaría de todas maneras.

Las excepciones pueden parecer poco amigables (después de todo, si no la captura, el programa entero se va al traste), pero considere la alternativa. ¿Preferiría tener un objeto de fichero inútil sobre un fichero no existente? Necesitaría comprobar su validez de todas maneras, y si se olvidase, en algún momento por debajo de esa línea, el programa mostraría errores extraños que debería trazar en el código fuente. Estoy seguro de que lo ha experimentado alguna vez, y sabe que no es divertido. Con las excepciones, el error ocurre de inmediato, y puede controlarlo de una manera estándar en la fuente del problema.

6.1.1. Uso de excepciones para otros propósitos

Hay muchos otros usos para las excepciones aparte de controlar verdaderas condiciones de error. Un uso común en la biblioteca estándar de Python es intentar importar un módulo, y comprobar si funcionó. Importar un módulo que no existe lanzará una excepción `ImportError`. Puede usar esto para definir varios niveles de funcionalidad basándose en qué módulos están disponibles en tiempo de ejecución, o para dar soporte a varias plataformas (donde esté separado el código específico de cada plataforma en varios módulos).

También puede definir sus propias excepciones creando una clase que herede de la clase incorporada `Exception`, para luego lanzarlas con la orden `raise`. Vea la sección de lecturas complementarias si le interesa hacer esto.

El próximo ejemplo demuestra el uso de una excepción para dar soporte a funcionalidad específica de una plataforma. Este código viene en el módulo `getpass` module, un módulo accesorio para obtener la clave del usuario. Esto se hace de manera diferente en las plataformas UNIX, Windows y Mac OS, pero el código encapsula todas esas diferencias.

Ejemplo 6.2. Dar soporte a funcionalidad específica de una plataforma

```
# Bind the name getpass to the appropriate function
```

```
try:
    import termios, TERMIOS                          ❶
except ImportError:
    try:
        import msvcrt                                ❷
    except ImportError:
        try:
            from EasyDialogs import AskPassword      ❸
        except ImportError:
            getpass = default_getpass                ❹
        else:                                        ❺
            getpass = AskPassword
    else:
        getpass = win_getpass
else:
    getpass = unix_getpass
```

❶ `termios` es un módulo específico de UNIX que proporciona control de la terminal a bajo nivel. Si no está disponible este módulo (porque no está en el sistema, o el sistema no da soporte), la importación falla y Python genera una `ImportError`, que podemos capturar.

❷ OK, no tenemos `termios`, así que probemos con `msvcrt`, que es un módulo específico de Windows que propociona un API a muchas funciones útiles en los servicios de ejecución de Visual C++. Si falla esta importación, Python lanzará una `ImportError`, que capturaremos.

❸ Si las dos primeras fallaron, podemos tratar de importar una función de `EasyDialogs`, un módulo específico de Mac OS que proporciona funciones para mostrar ventanas de diálogo de varios tipos. Una vez más, si falla esta operación, Python lanzará una `ImportError`, que capturaremos.

❹ Ninguno de estos módulos específicos de la plataforma están disponibles (puede suceder, ya que Python ha sido portado a muchas plataformas), de manera que necesita acudir a una función de introducción de *password* por omisión (que está definida en otra parte del módulo `getpass`). Observe lo que estamos haciendo aquí: asignamos la función `default_getpass` a la variable `getpass`. Si lee la documentación oficial de `getpass`, le dirá que el módulo `getpass` define una función `getpass`. Lo hace asociando `getpass` a la función

adecuada para su plataforma. Cuando llama a la función `getpass`, realmente está invocando una función específica de la plataforma en la que se ejecuta su código (simplemente llame a `getpass`, y siempre hará lo correcto).

5. Un bloque `try...except` puede tener una cláusula `else`, como la sentencia `if`. Si no se lanza una excepción durante el bloque `try`, al final se ejecuta la cláusula `else`. En este caso, eso significa que funcionó la importación `from EasyDialogs import AskPassword`, de manera que deberíamos asociar `getpass` a la función `AskPassword`. Cada uno de los otros bloques `try...except` tiene cláusulas `else` similares para asociar `getpass` a la función apropiada cuando se encuentre un `import` que funcione.

Lecturas complementarias sobre la gestión de excepciones

- El *Tutorial de Python* expone la definición y lanzamiento de sus propias excepciones, y la gestión de varias al mismo tiempo.
- La *Referencia de bibliotecas de Python* enumera todas las excepciones incorporadas.
- La *Referencia de bibliotecas de Python* documenta el módulo getpass.
- La *Referencia de bibliotecas de Python* documenta el módulo `traceback`, que proporciona acceso a bajo nivel a los atributos de las excepciones tras haberse lanzado una.
- La *Referencia del lenguaje Python* discute los entresijos del bloque `try...except`.

6.2. Trabajo con objetos de fichero

Python incorpora una funcioón, `open`, para abrir ficheros de un disco. `open` devuelve un objeto de fichero, que tiene métodos y atributos para obtener información sobre y manipular el fichero abierto.

Ejemplo 6.3. Apertura de un fichero

```
>>> f = open("/music/_singles/kairo.mp3", "rb") ❶
>>> f                                           ❷
<open file '/music/_singles/kairo.mp3', mode 'rb' at 010E3988>
>>> f.mode                                      ❸
'rb'
>>> f.name                                      ❹
'/music/_singles/kairo.mp3'
```

❶ El método `open` puede tomar hasta tres parámetros: un nombre de fichero, un modo y un parámetro para búfer. Sólo el primero, el nombre, es obligatorio; los otros dos son opcionales. Si no lo especifica, el fichero se abrirá en modo lectura y texto. Aquí estamos abriendo el fichero en modo binario (`print open.__doc__` muestra una gran explicación de todos los modos posibles).

❷ La función `open` devuelve un objeto (a estas alturas, esto no debería sorprenderle). Un objeto de fichero tiene varios atributos útiles.

❸ El atributo `mode` de un objeto de fichero nos dice en qué modo se abrió.

❹ El atributo `name` de un objeto de fichero nos dice el nombre del fichero que el objeto ha abierto.

6.2.1. Lectura de un fichero

Tras abrir un fichero, lo más importante que querrá hacer es leerlo, como se muestra en el siguiente ejemplo.

Ejemplo 6.4. Lectura de un fichero

```
>>> f
<open file '/music/_singles/kairo.mp3', mode 'rb' at 010E3988>
>>> f.tell()             ❶
0
>>> f.seek(-128, 2)      ❷
```

```
>>> f.tell()                ❸
7542909
>>> tagData = f.read(128)   ❹
>>> tagData
'TAGKAIRO****THE BEST GOA          ***DJ MARY-JANE***
Rave Mix                       2000http://mp3.com/DJMARYJANE     \037'
>>> f.tell()                ❺
7543037
```

❶ Un objeto de fichero mantiene el estado del fichero que ha abierto. El método `tell` del objeto te indica la posición actual dentro del fichero abierto. Dado que no hemos hecho nada con este fichero aún, la posición actual es `0`, que es el comienzo del fichero.

❷ El método `seek` de un objeto de fichero mueve la posición actual a otro lugar en el fichero abierto. El segundo parámetro especifica cómo interpretar el primero; `0` significa moverse en una posición absoluta (contando desde el comienzo del fichero), `1` significa moverse a una posición relativa (desde la actual), y `2` significa moverse a una opsición relativa al final del fichero. Dado que las etiquetas de MP3 que buscamos están almacenadas al final del fichero, usaremos `2` y le diremos al objeto que se mueva a la posición `128` antes del final del fichero.

❸ El método `tell` confirma que la posición actual del fichero ha cambiado.

❹ El método `read` lee un número especificado de bytes del fichero abierto y devuelve una cadena con los datos leídos. El parámetro opcional especifica el número máximo de bytes a leer. Si no se especifica parámetro, `read` leerá hasta el final del fichero (podría haber dicho simplemente `read()` aquí, ya que sabemos exactamente el sitio del fichero donde nos encontramos y estamos, en realidad, leyendo los últimos 128 bytes). Los datos leídos se asignan a la variable `tagData`, y se actualiza la posición actual basándose en el número de bytes leídos.

❺ El método `tell` confirma que la posición actual ha cambiado. Si hace los cálculos, comprobará que tras leer 128 bytes, la posición se ha incrementado en 128.

6.2.2. Cerrar ficheros

Los ficheros abiertos consumen recursos del sistema, y dependiendo del modo del fichero, puede que otros programas no puedan acceder a ellos. Es importante cerrar los ficheros tan pronto como haya terminado con ellos.

Ejemplo 6.5. Cierre de un fichero

```
>>> f
<open file '/music/_singles/kairo.mp3', mode 'rb' at 010E3988>
>>> f.closed       ❶
False
>>> f.close()      ❷
>>> f
<closed file '/music/_singles/kairo.mp3', mode 'rb' at 010E3988>
>>> f.closed       ❸
True
>>> f.seek(0)      ❹
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: I/O operation on closed file
>>> f.tell()
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: I/O operation on closed file
>>> f.read()
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: I/O operation on closed file
>>> f.close()      ❺
```

❶ El atributo `closed` de un objeto de fichero indica si el objeto tiene abierto un fichero o no. En este caso, el fichero sigue abierto (`closed` es `False`).

❷ Para cerrar un fichero, llame al método `close` del objeto. Esto libera el bloqueo (si lo hubiera) que estaba manteniendo sobre el fichero, activa la escritura de búfers (si los hubiera) que el sistema aún no ha escrito realmente, y libera los recursos del sistema.

❸ El atributo `closed` confirma que el fichero está cerrado.

❹ Sólo porque un fichero esté cerrado no significa que el objeto deje de existir. La variable `f` continuará existiendo hasta que salga de ámbito o sea eliminada manualmente. Sin embargo, ninguno de los métodos que manipula ficheros abiertos funcionará una vez cerrado el fichero; todos lanzarán una excepción.

❺ Invocar `close` en un objeto de fichero cuyo fichero ya está cerrado *no* lanza una excepción; falla silenciosamente.

6.2.3. Gestión de errores de E/S

Ahora ya ha visto bastante para comprender el código de gestión de ficheros en el ejemplo `fileinfo.py` del capítulo anterior. Este ejemplo le muestra cómo abrir y leer un fichero de forma segura y lidiar con los errores elegantemente.

Ejemplo 6.6. Objetos de fichero en `MP3FileInfo`

```
try:                                    ❶
    fsock = open(filename, "rb", 0)     ❷
    try:
        fsock.seek(-128, 2)             ❸
        tagdata = fsock.read(128)       ❹
    finally:                            ❺
        fsock.close()
    .
    .
    .
except IOError:                         ❻
    pass
```

❶ Como abrir y leer ficheros tiene su riesgo y puede provocar una excepción, todo el código está encapsulado en un bloque `try...except`. (¡Eh!, ¿no es maravilloso el sangrado estándarizado? Aquí es donde empezará a apreciarlo).

❷ La función `open` puede lanzar una `IOError` (puede que el fichero no exista).

❸ El método `seek` puede lanzar una `IOError` (puede que el fichero sea más pequeño que 128 bytes).

❹ El método `read` puede lanzar una `IOError` (puede que el disco tenga sectores defectuosos, o esté en una unidad de red y ésta acabe de caerse).

❺ Esto es nuevo: un bloque `try...finally`. Una vez la función `open` consiga abrir el fichero con éxito, querrá estar absolutamente seguro de cerrarlo, incluso si los métodos `seek` o `read` lanzan una excepción. Para esto es el bloque `try...finally`: el código en el bloque `finally` se ejecutará *siempre*, incluso si algo en el bloque `try` lanza una excepción. Piense en ello como código que se ha de ejecutar al salir, independientemente de lo que haya sucedido en medio.

❻ Por fin, gestionamos la excepción `IOError`. Podría ser la `IOError` lanzada por la llamada a `open`, `seek` o `read`. En este caso no nos importa, porque todo lo que vamos a hacer es ignorarlo en silencio y continuar (recuerde, `pass` es una sentencia de Python que no hace nada). Eso es perfectamente válido; "gestionar" una excepción puede querer decir no hacer nada, pero explícitamente. Sigue contando como gestión, y el proceso continuará de forma normal en la siguiente línea de código tras el bloque `try...except`.

6.2.4. Escribir en ficheros

Como cabría esperar, también podemos escribir en ficheros de la misma manera que podemos leer de ellos. Hay dos modos básicos de fichero:

- El modo "*append*" añadirá datos al final del fichero.
- el modo "*write*" sobrescribirá el contenido del fichero.

Cualquiera de los modos creará el fichero automáticamente si no existía ya, de manera que no se necesita ningún tipo de molesta lógica "si el fichero de registro no existe aún, crea un fichero nuevo vacío de manera que puedas abrirlo por primera vez".

Ejemplo 6.7. Escribir en ficheros

```
>>> logfile = open('test.log', 'w') ❶
>>> logfile.write('test succeeded') ❷
```

```
>>> logfile.close()
>>> print file('test.log').read()    ❸
test succeeded
>>> logfile = open('test.log', 'a') ❹
>>> logfile.write('line 2')
>>> logfile.close()
>>> print file('test.log').read()    ❺
test succeededline 2
```

❶ Empezamos sin compasión, creando un nuevo fichero `test.log` o sobrescribiendo el que ya existía, y abriéndolo para escribir (el segundo parámetro `"w"` significa abrir el fichero para escribir). Sí, esto es tan peligroso como suena. Espero que le importase el contenido previo del fichero, porque ya ha desaparecido.

❷ Puede añadir datos al fichero recién abierto con el método `write` del objeto de fichero devuelto por `open`.

❸ `file` es un sinónimo de `open`. Este *one-liner* abre el fichero, lee su contenido y lo imprime.

❹ Usted sabe que `test.log` existe (ya que acaba de escribir en él), de manera que podemos abrirlo y añadir datos al final (el parámetro `"a"` implica abrir el fichero para adición). En realidad podría hacer esto incluso si el fichero no existiese, porque abrir el fichero para añadir lo creará en caso necesario. Pero este modo *nunca* dañará el contenido ya existente en el fichero.

❺ Como puede ver, tanto la línea original que escribió como la segunda que acaba de añadir están en el fichero `test.log`. Observe también que no se incluye el retorno de carro. Como no lo ha escrito explícitamente ninguna de las veces, el fichero no lo incluye. Puede escribir un retorno de carro con el carácter `"\n"`. Como no lo ha hecho, todo lo que escribió en el fichero ha acabado pegadas en la misma línea.

Lecturas complementarias sobre manipulación de ficheros

- El *Tutorial de Python* comenta la lectura y escritura de ficheros, incluyendo la manera de leer un fichero una línea por vez guardándolo en una lista.

- eff-bot comenta la eficiencia y rendimiento de varias maneras de leer un fichero.
- La Python Knowledge Base responde preguntas frecuentes sobre ficheros.
- La *Referencia de bibliotecas de Python* enumera todosl os métodos del objeto de fichero.

6.3. Iteración con bucles `for`

Como la mayoría de los otros lenguajes, Python cuenta con bucles `for`. La única razón por la que no los ha visto antes es que Python es bueno en tantas otras cosas que no los ha necesitado hasta ahora con tanta frecuencia.

La mayoría de los otros lenguajes no tiene el poderoso tipo de lista como Python, de manera que acaba haciendo mucho trabajo manual, especificando un inicio, fin y paso que define un rango de enteros o caracteres y otras entidades iterables. Pero en Python, un bucle `for` simplemente itera sobre una lista, de la misma manera que funcionan las listas por comprensión.

Ejemplo 6.8. Presentación del bucle `for`

```
>>> li = ['a', 'b', 'e']
>>> for s in li:          ❶
...     print s           ❷
a
b
e
>>> print "\n".join(li)   ❸
a
b
e
```

❶ La sintaxis del bucle `for` es similar a la de las listas por comprensión. `li` es una lista, y `s` tomará el valor de cada elemento por turnos, comenzando por el primero.

❷ Como una sentencia `if` o cualquier otro bloque sangrado, un bucle `for` puede

constar de cualquier número de líneas.

❸ Ésta es la razón por la que aún no ha visto aún el bucle `for`: todavía no lo habíamos necesitado. Es impresionante lo a menudo que se usan los bucles `for` en otros lenguajes cuando todo lo que desea realmente es un `join` o una lista por comprensión.

Hacer un bucle contador "normal" (según estándares de Visual Basic) también es sencillo.

Ejemplo 6.9. Contadores simples

```
>>> for i in range(5):                ❶
...     print i
0
1
2
3
4
>>> li = ['a', 'b', 'c', 'd', 'e']
>>> for i in range(len(li)):          ❷
...     print li[i]
a
b
c
d
e
```

❶ Como pudo ver en Ejemplo 3.20, "Asignación de valores consecutivos", `range` produce una lista de enteros, sobre la que puede iterar. Sé que parece un poco extraño, pero ocasionalmente (y subrayo ese *ocasionalmente*) es útil tener un bucle contador.

❷ Nunca haga esto. Esto es pensar estilo Visual Basic. Despréndase de eso. Simplemente, itere sobre la lista, como se mostró en el ejemplo anterior.

Los bucles `for` no son sólo simples contadores. Puede interar sobre todo tipo de cosas. Aquí tiene un ejemplo de uso de un bucle `for` para iterar sobre un diccionario.

Ejemplo 6.10. Iteración sobre un diccionario

```
>>> import os
>>> for k, v in os.environ.items():      ❶ ❷
...     print "%s=%s" % (k, v)
USERPROFILE=C:\Documents and Settings\mpilgrim
OS=Windows_NT
COMPUTERNAME=MPILGRIM
USERNAME=mpilgrim

[...snip...]
>>> print "\n".join(["%s=%s" % (k, v)
...     for k, v in os.environ.items()]) ❸
USERPROFILE=C:\Documents and Settings\mpilgrim
OS=Windows_NT
COMPUTERNAME=MPILGRIM
USERNAME=mpilgrim

[...snip...]
```

❶ `os.environ` es un diccionario de las variables de entorno definidas en su sistema. En Windows, son su usuario y las variables accesibles desde MS-DOS. En UNIX, son las variables que exportada por los *scripts* de inicio de su intérprete de órdenes. En Mac OS, no hay concepto de variables de entorno, de manera que el diccionario está vacío.

❷ `os.environ.items()` devuelve una lista detuplas: `[(`*`clave1`*`, `*`valor1`*`), (`*`clave2`*`, `*`valor2`*`), ...]`. El bucle `for` itera sobre esta lista. En la primera iteración, asigna *`clave1`* a `k` y *`valor1`* a `v`, de manera que `k` = `USERPROFILE` y `v` = `C:\Documents and Settings\mpilgrim`. En la segunda iteración, `k` obtiene la segunda clave, `OS`, y `v` el valor correspondiente, `Windows_NT`.

❸ Con la asignación de múltiples variables y las listas por comprensión, puede reemplazar todo el bucle `for` con una sola sentencia. Hacer o no esto en código real es una cuestión de estilo personal al programar. Me gusta porque deja claro que lo que hago es relacionar un diccionario en una lista, y luego unir la lista en una única cadena. Otros programadores prefieren escribir esto como un bucle `for`. La salida es la misma en cualquier caso, aunque esta

versión es ligeramente más rápidao, porque sólo hay una sentencia `print` en lugar de muchas.

Ahora podemos mirar el bucle `for` de `MP3FileInfo`, del programa de ejemplo `fileinfo.py` que presentamos en Capítulo 5.

Ejemplo 6.11. Bucle `for` en `MP3FileInfo`

```
    tagDataMap = {"title"   : (  3,  33, stripnulls),
                  "artist"  : ( 33,  63, stripnulls),
                  "album"   : ( 63,  93, stripnulls),
                  "year"    : ( 93,  97, stripnulls),
                  "comment" : ( 97, 126, stripnulls),
                  "genre"   : (127, 128, ord)}
❶
    .
    .
    .
            if tagdata[:3] == "TAG":
                for tag, (start, end, parseFunc) in
self.tagDataMap.items(): ❷
                    self[tag] = parseFunc(tagdata[start:end])
❸
```

❶ `tagDataMap` es un atributo de clase que define las etiquetas que está buscando en un fichero MP3. Las etiquetas se almacenan en campos de longitud fija. Una vez ha leído los últimos 128 bytes del fichero, los bytes 3 al 32 siempre corresponden al título de la canción, del 33 al 62 es siempre el nombre del artista, del 63 al 92 son el nombre del álbum, y así en adelante. Observe que `tagDataMap` es un diccionario de tuplas, y cada tupla contiene dos enteros y una referencia a una función.

❷ Parece complicado, pero no lo es. La estructura de las variables del `for` se corresponden a la estructura de los elementos de la lista devuelta por `items`. Recuerde que `items` devuelve una lista de tuplas de la forma `(clave, valor)`. El primer elemento de esa lista es `("title", (3, 33, <function stripnulls>))`, de manera que la primera iteración del bucle, `tag` contiene `"title"`, `start` contiene `3`, `end` obtiene `33`, y `parseFunc` tendrá asignada la

función `stripnulls`.

❸ Ahora que hemos extraído todos los parámetros de una etiqueta de MP3, guardar sus datos es sencillo. Haremos un *slice* de `tagdata` desde `start` hasta `end` para obtener el dato de esa etiqueta, llamaremos a `parseFunc` para postprocesar el dato, y lo asignaremos como valor de la clave `tag` en el pseudo-diccionario `self`. Tras iterar sobre todos los elementos de `tagDataMap`, `self` tiene los valores de todas las etiquetas, y ya sabe el aspecto que tiene eso.

6.4. Uso de `sys.modules`

Los módulos, como todo lo demás en Python son objetos. Una vez importados, siempre puede obtener una referencia a un módulo mediante el diccionario global `sys.modules`.

Ejemplo 6.12. Presentación de `sys.modules`

```
>>> import sys                              ❶
>>> print '\n'.join(sys.modules.keys())     ❷
win32api
os.path
os
exceptions
__main__
ntpath
nt
sys
__builtin__
site
signal
UserDict
stat
```

❶ El módulo `sys` contiene información sobre el sistema, tal como la versión de Python que ejecutamos (`sys.version` o `sys.version_info`), y opciones del sistema tales como el nivel de recursión máximo permitido (`sys.getrecursionlimit()` y `sys.setrecursionlimit()`).

❷ `sys.modules` es un diccionario que contiene todos los módulos que se han importado desde que arrancara Python; la clave es el nombre del módulo, el valor es el objeto del módulo. Advierta que aquí hay más módulos de los que *su* programa ha importado. Python carga algunos módulos durante el arranque, y si usa un IDE para Python, `sys.modules` contendrá todos los módulos importados por todos los programas que esté ejecutando dentro del IDE.

Este ejemplo demuestra el uso de `sys.modules`.

Ejemplo 6.13. Uso de `sys.modules`

```
>>> import fileinfo          ❶
>>> print '\n'.join(sys.modules.keys())
win32api
os.path
os
fileinfo
exceptions
__main__
ntpath
nt
sys
__builtin__
site
signal
UserDict
stat
>>> fileinfo
<module 'fileinfo' from 'fileinfo.pyc'>
>>> sys.modules["fileinfo"] ❷
<module 'fileinfo' from 'fileinfo.pyc'>
```

❶ Los módulos nuevos van siendo añadidos a `sys.modules` según son importados. Esto explica por qué importar un módulo por segunda vez es muy rápido: Python ya lo ha cargado y lo tiene en caché en `sys.modules`, de manera que importarlo la segunda vez no cuesta más que una búsqueda en un diccionario.

❷ Dado el nombre (como cadena) de cualquier módulo importado previamente, puede obtener una referencia al propio módulo mediante el diccionario `sys.modules`.

El siguiente ejemplo muestra el uso del atributo de clase `__module__` con el diccionario `sys.modules` para obtener una referencia al módulo en el que está definida una clase.

Ejemplo 6.14. El atributo de clase `__module__`

```
>>> from fileinfo import MP3FileInfo
>>> MP3FileInfo.__module__                ❶
'fileinfo'
>>> sys.modules[MP3FileInfo.__module__] ❷
<module 'fileinfo' from 'fileinfo.pyc'>
```

❶ Cada clase de Python tiene un atributo de clase `__module__` incorporado, que consiste en el nombre del módulo en que se definió la clase.

❷ Combinando esto con el diccionario `sys.modules`, podemos obtener una referencia al módulo en que se definió la clase.

Ahora ya está listo para comprobar el uso de `sys.modules` en `fileinfo.py`, el programa de ejemplo presentado en Capítulo 5. Este ejemplo muestra esa porción de código.

Ejemplo 6.15. `sys.modules` en `fileinfo.py`

```
    def getFileInfoClass(filename,
module=sys.modules[FileInfo.__module__]):       ❶
        "get file info class from filename extension"
        subclass = "%sFileInfo" %
os.path.splitext(filename)[1].upper()[1:]        ❷
        return hasattr(module, subclass) and getattr(module, subclass)
or FileInfo ❸
```

❶ Ésta es una función con dos argumentos; `filename` es obligatoria, pero `module` es opcional y por omisión es el módulo que contiene la clase `FileInfo`. Esto no parece eficiente, porque es de esperar que Python evalúe la expresión

`sys.modules` cada vez que se llama a la función. En realidad, Python evalúa las expresiones por omisión sólo una vez, la primera en que se importa el módulo. Como verá más adelante, nunca se llama a esta función con un argumento `module`, así que `module` sirve como constante en la función.

❷ Volverá a esta línea más adelante, tras sumergirse en el módulo `os`. Por ahora, creáse que `subclass` acaba siendo el nombre de una clase, como `MP3FileInfo`.

❸ Ya conoce `getattr`, que obtiene una referencia a un objeto por su nombre. `hasattr` es una función complementaria, que comprueba si un objeto tiene un atributo en particular; en este caso, si un módulo tiene una clase en particular (aunque funciona con cualquier objeto y cualquier atributo, igual que `getattr`). En español, el código de esta línea dice, "Si este módulo contiene la clase denominada `subclass` entonces devuélvela, en caso contrario devuelve la clase base `FileInfo`."

Lecturas complementarias sobre módulos

- El *Tutorial de Python* comenta exactamente cuándo y cómo se evalúan los argumentos por omisión.
- La *Referencia de bibliotecas de Python* documenta el módulo `sys`.

6.5. Trabajo con directorios

El módulo `os.path` tiene varias funciones para manipular ficheros y directorios. Aquí, queremos manuplar rutas y listar el contenido de un directorio.

Ejemplo 6.16. Construcción de rutas

```
>>> import os
>>> os.path.join("c:\\music\\ap\\", "mahadeva.mp3") ❶ ❷
'c:\\music\\ap\\mahadeva.mp3'
>>> os.path.join("c:\\music\\ap", "mahadeva.mp3")   ❸
'c:\\music\\ap\\mahadeva.mp3'
>>> os.path.expanduser("~")                         ❹
```

```
'c:\\Documents and Settings\\mpilgrim\\My Documents'
>>> os.path.join(os.path.expanduser("~"), "Python") ❺
'c:\\Documents and Settings\\mpilgrim\\My Documents\\Python'
```

❶ `os.path` es una referencia a un módulo (qué módulo exactamente, depende de su plataforma). Al igual que `getpass` encapsula las diferencias entre plataformas asociando `getpass` a una función específica, `os` encapsula las diferencias entre plataformas asociando `path` a un módulo específico par la suya.

❷ La función `join` de `os.path` construye el nombre de una ruta partiendo de una o más rutas parciales. En este caso, simplemente concatena cadenas (observe que trabajar con nombres de rutas en Windows es molesto debido a que hay que escapar la barra inversa).

❸ En este caso ligeramente menos trivial, `join` añadirá una barra inversa extra a la ruta antes de unirla al nombre de fichero. Quedé encantadísimo cuando descubrí esto, ya que `addSlashIfNecessary` es una de las pequeñas funciones estúpidas que siempre tengo que escribir cuando escribo mis propias herramientas en un nuevo lenguaje. *No* escriba esta pequeña estúpida función en Python hay gente inteligente que ya lo ha hecho por usted.

❹ `expanduser` expandirá un nombre de ruta que utilice ~ para representar el directorio personal del usuario actual. Esto funciona en cualquier plataforma donde los usuarios tengan directorios personales, como Windows, UNIX y Mac OS X; no tiene efecto en Mac OS.

❺ Combinando estas técnicas, puede construir fácilmente rutas para directorios y ficheros bajo el directorio personal del usuario.

Ejemplo 6.17. Dividir nombres de rutas

```
>>> os.path.split("c:\\music\\ap\\mahadeva.mp3")
❶
('c:\\music\\ap', 'mahadeva.mp3')
>>> (filepath, filename) =
os.path.split("c:\\music\\ap\\mahadeva.mp3") ❷
>>> filepath
❸
```

```
'c:\\music\\ap'
>>> filename
❹
'mahadeva.mp3'
>>> (shortname, extension) = os.path.splitext(filename)
❺
>>> shortname
'mahadeva'
>>> extension
'.mp3'
```

❶ La función `split` divide una ruta completa y devuelve una tupla que contiene la ruta y el nombre del fichero. ¿Recuerda cuando dije que podría usar la asignación de múltiples variables para devolver varios valores de una función? Bien, `split` es una de esas funciones.

❷ Asignamos el valor de retorno de la función `split` a una tupla con dos variables. Cada variable recibe el valor del elemento correspondiente de la tupla devuelta.

❸ La primera variable, `filepath`, recibe el valor del primer elemento de la tupla devuelta por `split`, la ruta hasta el fichero.

❹ La segunda variable, `filename`, recibe el valor del segundo elemento de la tupla devuelta por `split`, el nombre del fichero.

❺ `os.path` también contiene una función `splitext`, que divide un nombre de fichero y devuelve una tupla que contiene el nombre y la extensión. Usamos la misma técnica para asignar cada una de ellas a variables distintas.

Ejemplo 6.18. Listado de directorios

```
>>> os.listdir("c:\\music\\_singles\\")                ❶
['a_time_long_forgotten_con.mp3', 'hellraiser.mp3',
'kairo.mp3', 'long_way_home1.mp3', 'sidewinder.mp3',
'spinning.mp3']
>>> dirname = "c:\\"
>>> os.listdir(dirname)                                ❷
['AUTOEXEC.BAT', 'boot.ini', 'CONFIG.SYS', 'cygwin',
'docbook', 'Documents and Settings', 'Incoming', 'Inetpub', 'IO.SYS',
'MSDOS.SYS', 'Music', 'NTDETECT.COM', 'ntldr', 'pagefile.sys',
```

```
'Program Files', 'Python20', 'RECYCLER',
'System Volume Information', 'TEMP', 'WINNT']
>>> [f for f in os.listdir(dirname)
...     if os.path.isfile(os.path.join(dirname, f))] ❸
['AUTOEXEC.BAT', 'boot.ini', 'CONFIG.SYS', 'IO.SYS', 'MSDOS.SYS',
'NTDETECT.COM', 'ntldr', 'pagefile.sys']
>>> [f for f in os.listdir(dirname)
...     if os.path.isdir(os.path.join(dirname, f))]  ❹
['cygwin', 'docbook', 'Documents and Settings', 'Incoming',
'Inetpub', 'Music', 'Program Files', 'Python20', 'RECYCLER',
'System Volume Information', 'TEMP', 'WINNT']
```

❶ La función `listdir` toma una ruta y devuelve una lista con el contenido de ese directorio.

❷ `listdir` devuelve tanto ficheros como carpetas, sin indicar cual es cual.

❸ Puede usar el filtrado de listas y la función `isfile` del módulo `os.path` para separar los ficheros de las carpetas. `isfile` toma una ruta y devuelve 1 si representa un fichero, y un 0 si no. Aquí estamos usando `os.path.join` para asegurarnos de tener una ruta completa, pero `isfile` también funciona con rutas parciales, relativas al directorio actual de trabajo. Puede usar `os.getcwd()` para obtener el directorio de trabajo.

❹ `os.path` también tiene una función `isdir` que devuelve 1 si la ruta representa un directorio y 0 si no. Puede usarlo para obtener una lista de los subdirectorios dentro de un directorio.

Ejemplo 6.19. Listado de directorios en `fileinfo.py`

```
def listDirectory(directory, fileExtList):
    "get list of file info objects for files of particular extensions"
    fileList = [os.path.normcase(f)
                for f in os.listdir(directory)]           ❶ ❷
    fileList = [os.path.join(directory, f)
                for f in fileList
                if os.path.splitext(f)[1] in fileExtList]  ❸ ❹ ❺
```

❶ `os.listdir(directory)` devuelve una lista de todos los ficheros y carpetas en `directory`.

❷ Iterando sobre la lista con `f`, usamos `os.path.normcase(f)` para normalizar el caso de acuerdo a los valores por omisión del sistema operativo. `normcase` es una pequeña y útil función que compensa en los sistemas operativos insensibles al caso que piensan que `mahadeva.mp3` y `mahadeva.MP3` son el mismo fichero. Por ejemplo, en Windows y Mac OS, `normcase` convertirá todo el nombre del fichero a minúsculas; en los sistemas compatibles con UNIX, devolverá el nombre sin cambios.

❸ Iterando sobre la lista normalizada de nuevo con `f`, usamos `os.path.splitext(f)` para dividir cada nombre de fichero en nombre y extensión.

❹ Por cada fichero, vemos si la extensión está en la lista de extensiones de ficheros que nos ocupan (`fileExtList`, que se pasó a la función `listDirectory`).

❺ Por cada fichero que nos interesa, usamos `os.path.join(directory, f)` para construir la ruta completa hasta el fichero, y devolvemos una lista de las rutas absolutas.

Siempre que sea posible, debería usar las funciones de `os` y `os.path` para manipulaciones sobre ficheros, directorios y rutas. Estos módulos encapsulan los específicos de cada plataforma, de manera que funciones como `os.path.split` funcionen en UNIX, Windows, Mac OS y cualquier otra plataforma en que funcione Python.

Hay otra manera de obtener el contenido de un directorio. Es muy potente, y utiliza los comodines con los que posiblemente esté familiarizado al trabajar en la línea de órdenes.

Ejemplo 6.20. Listado de directorios con `glob`

```
>>> os.listdir("c:\\music\\_singles\\")                ❶
['a_time_long_forgotten_con.mp3', 'hellraiser.mp3',
'kairo.mp3', 'long_way_home1.mp3', 'sidewinder.mp3',
'spinning.mp3']
```

```
>>> import glob
>>> glob.glob('c:\\music\\_singles\\*.mp3')                    ❷
['c:\\music\\_singles\\a_time_long_forgotten_con.mp3',
'c:\\music\\_singles\\hellraiser.mp3',
'c:\\music\\_singles\\kairo.mp3',
'c:\\music\\_singles\\long_way_home1.mp3',
'c:\\music\\_singles\\sidewinder.mp3',
'c:\\music\\_singles\\spinning.mp3']
>>> glob.glob('c:\\music\\_singles\\s*.mp3')                   ❸
['c:\\music\\_singles\\sidewinder.mp3',
'c:\\music\\_singles\\spinning.mp3']
>>> glob.glob('c:\\music\\*\\*.mp3')                           ❹
```

❶ Como vimos anteriormente, `os.listdir` se limita a tomar una ruta de directorio y lista todos los ficheros y directorios que hay dentro suya.

❷ El módulo `glob`, por otro lado, toma un comodín y devuelve la ruta absoluta hasta todos los ficheros y directorios que se ajusten al comodín. Aquí estamos usando una ruta de directorio más "*.mp3", que coincidirá con todos los ficheros `.mp3`. Observe que cada elemento de la lista devuelta incluye la ruta completa hasta el fichero.

❸ Si quiere encontrar todos los ficheros de un directorio específico que empiecen con "s" y terminen en ".mp3", también puede hacerlo.

❹ Ahora imagine esta situación: tiene un directorio `music`, con varios subdirectorios, con ficheros `.mp3` dentro de cada uno. Podemos obtener una lista de todos ellos con una sola llamada a `glob`, usando dos comodines a la vez. Uno es el `"*.mp3"` (para capturar los ficheros `.mp3`), y el otro comodín está *dentro de la propia ruta al directorio*, para que capture todos los subdirectorios dentro de `c:\music`. ¡Una increíble cantidad de potencial dentro de una función de aspecto engañosamente simple!

Lecturas complementarias sobre el módulo `os`

- La Python Knowledge Base contesta preguntas sobre el módulo `os`.
- La *Referencia de bibliotecas de Python* documenta los módulos `os` y `os.path`.

6.6. Todo junto

Una vez más, todas las piezas de dominó están en su lugar. Ha visto cómo funciona cada línea de código. Ahora retrocedamos y veamos cómo encaja todo.

Ejemplo 6.21. `listDirectory`

```
def listDirectory(directory, fileExtList):
❶
    "get list of file info objects for files of particular extensions"
    fileList = [os.path.normcase(f)
                for f in os.listdir(directory)]
    fileList = [os.path.join(directory, f)
                for f in fileList
                if os.path.splitext(f)[1] in fileExtList]
❷
    def getFileInfoClass(filename,
module=sys.modules[FileInfo.__module__]):       ❸
        "get file info class from filename extension"
        subclass = "%sFileInfo" %
os.path.splitext(filename)[1].upper()[1:]        ❹
        return hasattr(module, subclass) and getattr(module, subclass)
or FileInfo ❺
    return [getFileInfoClass(f)(f) for f in fileList]
❻
```

❶ `listDirectory` es la atracción principal de todo este módulo. Toma un directorio (como `c:\music\_singles\` en mi caso) y una lista de extensiones de fichero interesantes (como `['.mp3']`), y devuelve una lista instancias de clases que actúan como diccionarios que contienen metadatos sobre cada fichero interesante de ese directorio. Y lo hace en unas pocas líneas de código bastante simples.

❷ Como pudo ver en la sección anterior, esta línea de código devuelve una lista de las rutas absolutas de todos los ficheros de `directory` que tienen una extensión interesante (especificada por `fileExtList`).

❸ Los programadores de Pascal de la vieja escuela estarán familiarizados con ellas, pero la mayoría de la gente se me queda mirando con cara de tonto cuando digo que Python admite *funciones anidada* (literalmente, una función

dentro de otra). Sólo se puede llamar a la función anidada `getFileInfoClass` desde dentro de la función en que está definida, `listDirectory`. Como con cualquier otra función, no necesita una interfaz de declaración ni nada parecido; limítese a definirla y programarla.

❹ Ahora que ha visto el módulo `os`, esta línea debería tener más sentido. Toma la extensión del fichero (`os.path.splitext(filename)[1]`), fuerza a convertirla en mayúsculas (`.upper()`), le quita el punto (`[1:]`), y construye el nombre de una clase usando cadenas de formato. Así que `c:\music\ap\mahadeva.mp3` se convierte en `.mp3`, y esto a su vez en `.MP3` que acaba siendo `MP3` para darnos `MP3FileInfo`.

❺ Habiendo construido el nombre de la clase controladora que manipulará este fichero, comprobamos si la clase existe realmente en este módulo. Si está, devuelve la clase, y si no, devuelve la clase base `FileInfo`. Éste es un detalle muy importante: *esta función devuelve una clase*. No una instancia de una clase, sino la clase en sí.

❻ Por cada fichero de la lista "ficheros interesantes" (`fileList`), llamamos a `getFileInfoClass` pasándole el nombre del fichero (`f`). La invocación `getFileInfoClass(f)` devuelve una clase; que no sabemos exactamente cual es, pero no nos importa. Entonces creamos una instancia de esta clase (la que sea) y pasamos el nombre del fichero (`f` de nuevo) al método `__init__`. Como vio anteriormente en este capítulo, el método `__init__` de `FileInfo` asigna `self["name"]`, lo que dispara `__setitem__`, que está reempleazada en la clase descendiente (`MP3FileInfo`) para que analice el fichero de forma apropiada sacando los metadatos de su interior. Esto lo hacemos por cada fichero interesante y devolvemos una lista de las instancias resultantes.

Observe que `listDirectory` es completamente genérica. No sabe nada con antelación sobre los tipos de ficheros con que va a trabajar, o qué clases están definidas que podrían potencialmente manipular estos ficheros. Inspecciona el directorio en busca de ficheros con que trabajar, y luego introspecciona en su propio módulo para ver qué clases manipuladoras (como `MP3FileInfo`) hay definidas. Puede extender este programa para manipular otros tipos de ficheros

con solo definir una clase con el nombre apropiado: `HTMLFileInfo` para ficheros HTML, `DOCFileInfo` para ficheros `.doc` de Word, *etc.*. `listDirectory` trabajará con todas, sin modificaciones, delegando el trabajo real a las clases apropiadas y recogiendo los resultados.

6.7. Resumen

El programa `fileinfo.py` que presentamos en Capítulo 5 debería ahora tener todo el sentido del mundo.

```
"""Framework for getting filetype-specific metadata.

Instantiate appropriate class with filename.  Returned object acts
like a
dictionary, with key-value pairs for each piece of metadata.
    import fileinfo
    info = fileinfo.MP3FileInfo("/music/ap/mahadeva.mp3")
    print "\\n".join(["%s=%s" % (k, v) for k, v in info.items()])

Or use listDirectory function to get info on all files in a directory.
    for info in fileinfo.listDirectory("/music/ap/", [".mp3"]):
        ...

Framework can be extended by adding classes for particular file types,
e.g.
HTMLFileInfo, MPGFileInfo, DOCFileInfo.  Each class is completely
responsible for
parsing its files appropriately; see MP3FileInfo for example.
"""
import os
import sys
from UserDict import UserDict

def stripnulls(data):
    "strip whitespace and nulls"
    return data.replace("\00", "").strip()

class FileInfo(UserDict):
    "store file metadata"
    def __init__(self, filename=None):
```

```
        UserDict.__init__(self)
        self["name"] = filename

class MP3FileInfo(FileInfo):
    "store ID3v1.0 MP3 tags"
    tagDataMap = {"title"   : (  3,  33, stripnulls),
                  "artist"  : ( 33,  63, stripnulls),
                  "album"   : ( 63,  93, stripnulls),
                  "year"    : ( 93,  97, stripnulls),
                  "comment" : ( 97, 126, stripnulls),
                  "genre"   : (127, 128, ord)}

    def __parse(self, filename):
        "parse ID3v1.0 tags from MP3 file"
        self.clear()
        try:
            fsock = open(filename, "rb", 0)
            try:
                fsock.seek(-128, 2)
                tagdata = fsock.read(128)
            finally:
                fsock.close()
            if tagdata[:3] == "TAG":
                for tag, (start, end, parseFunc) in
self.tagDataMap.items():
                    self[tag] = parseFunc(tagdata[start:end])
        except IOError:
            pass

    def __setitem__(self, key, item):
        if key == "name" and item:
            self.__parse(item)
        FileInfo.__setitem__(self, key, item)

def listDirectory(directory, fileExtList):
    "get list of file info objects for files of particular extensions"
    fileList = [os.path.normcase(f)
                for f in os.listdir(directory)]
    fileList = [os.path.join(directory, f)
               for f in fileList
                if os.path.splitext(f)[1] in fileExtList]
```

```
    def getFileInfoClass(filename,
module=sys.modules[FileInfo.__module__]):
        "get file info class from filename extension"
        subclass = "%sFileInfo" %
os.path.splitext(filename)[1].upper()[1:]
        return hasattr(module, subclass) and getattr(module, subclass)
or FileInfo
    return [getFileInfoClass(f)(f) for f in fileList]

if __name__ == "__main__":
    for info in listDirectory("/music/_singles/", [".mp3"]):
        print "\n".join(["%s=%s" % (k, v) for k, v in info.items()])
        print
```

Antes de sumergirnos en el próximo capítulo, asegúrese de que es capaz de hacer las siguientes cosas con comodidad:

- Capturar excepciones con `try...except`
- Proteger recursos externos con `try...finally`
- Leer ficheros
- Asignar múltiples valores a la vez en un bucle `for`
- Usar el módulo `os` para todas las manipulaciones de fichero que necesite de forma multiplataforma
- Instanciar clases de tipo desconocido de forma dinámica tratando las clases como objetos pasándolas a funciones

Capítulo 7. Expresiones regulares

Las expresiones regulares son una forma moderna y estandarizada de hacer búsquedas, reemplazos y análisis de texto usando patrones complejos de caracteres. Si ha usado las expresiones regulares en otros lenguajes (como Perl), la sintaxis le será muy familiar, y con sólo leer el resumen de módulo `re` puede hacerse una idea general de las funciones disponibles y sus parámetros.

7.1. Inmersión

Las cadenas tienen métodos para la búsqueda (`index`, `find`, y `count`), reemplazo (`replace`), y análisis (`split`), pero están limitadas a los casos más sencillos. Los métodos de búsqueda encuentran cadenas sencillas, fijas, y siempre distinguiendo las mayúsculas. Para hacer búsquedas de una cadena `s` sin que importen las mayúsculas, debe invocar `s.lower()` o `s.upper()` y asegurarse de que las cadenas que busca están en el caso adecuado. Los métodos `replace` y `split` tienen las mismas limitaciones.

Si lo que intentamos hacer se puede realizar con las funciones de cadenas, debería usarlas. Son rápidas y simples, y sencillas de entender, y todo lo bueno que se diga sobre el código rápido, simple y legible, es poco. Pero si se encuentra usando varias funciones de cadenas diferentes junto con sentencias

`if` para manejar casos especiales, o si las tiene que combinar con `split` y `join` y listas por comprensión de formas oscuras e ilegibles, puede que deba pasarse a las expresiones regulares.

Aunque la sintaxis de las expresiones regulares es compacta y diferente del código normal, el resultado puede acabar siendo *más* legible que una solución hecha a mano que utilice una larga cadena de funciones de cadenas. Hay incluso maneras de insertar comentarios dentro de una expresión regular para hacerlas prácticamente autodocumentadas.

7.2. Caso de estudio: direcciones de calles

Esta serie de ejemplos la inspiró un problema de la vida real que surgió en mi trabajo diario hace unos años, cuando necesité limpiar y estandarizar direcciones de calles exportadas de un sistema antiguo antes de importarlo a un nuevo sistema (vea que no me invento todo esto; es realmente útil). Este ejemplo muestra la manera en que me enfrenté al problema.

Ejemplo 7.1. Buscando el final de una cadena

```
>>> s = '100 NORTH MAIN ROAD'
>>> s.replace('ROAD', 'RD.')                    ❶
'100 NORTH MAIN RD.'
>>> s = '100 NORTH BROAD ROAD'
>>> s.replace('ROAD', 'RD.')                    ❷
'100 NORTH BRD. RD.'
>>> s[:-4] + s[-4:].replace('ROAD', 'RD.')      ❸
'100 NORTH BROAD RD.'
>>> import re                                   ❹
>>> re.sub('ROAD$', 'RD.', s)                   ❺ ❻
'100 NORTH BROAD RD.'
```

❶ Mi objetivo es estandarizar la dirección de calle de manera que `'ROAD'` siempre quede abreviado como `'RD.'`. A primera vista, pensé que era suficientemente simple como para limitarme a usar el método de cadena `replace`. Después de todo, los datos ya estaban en mayúsculas, así que no

habría problema con diferencias de mayúsculas. Y la cadena a buscar, `'ROAD'`, era constante. Y en este ejemplo engañosamente sencillo, `s.replace` funciona.

❷ La vida, desafortunadamente, está llena de contraejemplos, y descubrí éste enseguida. El problema aquí es que `'ROAD'` aparece dos veces en la dirección, una como parte del nombre de la calle `'BROAD'` y otra como la propia palabra. La función `replace` ve las dos apariciones y las reemplaza ambas sin mirar atrás; mientras tanto, yo veo cómo mi dirección queda destruida.

❸ Para resolver el problema de las direcciones con más de una subcadena `'ROAD'`, podríamos recurrir a algo como esto: buscar y reemplazar `'ROAD'` sólo en los últimos cuatro caracteres de la dirección (`s[-4:]`), y dejar el resto sin modificar (`s[:-4]`). Pero como puede ver, esto ya se está volviendo inmanejable. Por ejemplo, el patrón depende de la longitud de la cadena que estamos reemplazando (si quisiera sustituir `'STREET'` por `'ST.'`, necesitaría usar `s[:-6]` y `s[-6:].replace(...)`). ¿Le gustaría volver en seis meses y tener que depurar esto? Sé que a mí no me gustaría.

❹ Es hora de pasar a las expresiones regulares. En Python, toda la funcionalidad relacionada con las expresiones regulares está contenida en el módulo `re`.

❺ Eche un vistazo al primer parámetro: `'ROAD$'`. Ésta es una expresión regular sencilla que coincide con `'ROAD'` sólo cuando se encuentra al final de una cadena. El `$` significa “fin de la cadena” (existe el carácter correspondiente, el acento circunflejo `^`, que significa “comienzo de la cadena”).

❻ Usando la función `re.sub`, buscamos la expresión regular `'ROAD$'` dentro de la cadena `s` y la reemplazamos con `'RD.'`. Esto coincide con `ROAD` al final de la cadena `s`, pero *no* con la `ROAD` que es parte de la palabra `BROAD`, porque está en mitad de `s`.

Siguiendo con mi historia de adecentar las direcciones, pronto descubrí que el ejemplo anterior que se ajustaba a `'ROAD'` al final de las direcciones, no era suficientemente bueno, porque no todas las direcciones incluían la indicación del tipo de calle; algunas simplemente terminaban en el nombre de la calle. La

mayor parte de las veces, podía pasar con eso, pero si la calle se llamaba `'BROAD'`, entonces la expresión regular coincidiría con el `'ROAD'` del final de la cadena siendo parte de `'BROAD'`, que no es lo que yo quería.

Ejemplo 7.2. Coincidencia con palabras completas

```
>>> s = '100 BROAD'
>>> re.sub('ROAD$', 'RD.', s)
'100 BRD.'
>>> re.sub('\\bROAD$', 'RD.', s)  ❶
'100 BROAD'
>>> re.sub(r'\bROAD$', 'RD.', s)  ❷
'100 BROAD'
>>> s = '100 BROAD ROAD APT. 3'
>>> re.sub(r'\bROAD$', 'RD.', s)  ❸
'100 BROAD ROAD APT. 3'
>>> re.sub(r'\bROAD\b', 'RD.', s) ❹
'100 BROAD RD. APT 3'
```

❶ Lo que yo quería *de verdad* era una coincidencia con `'ROAD'` cuando estuviera al final de la cadena *y* fuera una palabra en sí misma, no parte de una mayor. Para expresar esto en una expresión regular, usamos `\b`, que significa "aquí debería estar el límite de una palabra". En Python, esto se complica debido al hecho de que el carácter `'\'` ha de escaparse si está dentro de una cadena. A veces a esto se le llama la plaga de la barra inversa, y es una de las razones por las que las expresiones regulares son más sencillas en Perl que en Python. Por otro lado, Perl mezcla las expresiones regulares con el resto de la sintaxis, de manera que si tiene un fallo, será difícil decidir si es a causa de la sintaxis o de la expresión regular.

❷ Para evitar la plaga de la barra inversa, puede usar lo que se denominan cadenas sin procesar[3], prefijando una letra `r` a la cadena. Esto le dice a Python que nada de esa cadena debe ser escapado; `'\t'` es un carácter de tabulador, pero `r'\t'` es en realidad el carácter de la barra `\` seguido de la letra `t`. Le recomiendo que siempre use estas cadenas cuando trabaje con expresiones regulares; de otro modo, las cosas pueden volverse confusas rápidamente (y las expresiones regulares ya se vuelven confusas

suficientemente rápido por sí mismas).

❸ **suspiro** Desafortunadamente, pronto encontré más casos que contradecían mi lógica. En este caso, la dirección de la calle contenía la palabra `'ROAD'` por separado, pero no estaba al final, ya que la dirección incluía un número de apartamento tras la indicación de la calle. Como `'ROAD'` no estaba justo al final de la cadena, no había coincidencia, de manera que la invocación a `re.sub` acababa por no reemplazar nada, y obtenía la cadena original, que no es lo que deseaba.

❹ Para resolver este problema, eliminé el carácter `$` y añadí otro `\b`. Ahora la expresión regular dice "busca una `'ROAD'` que sea una palabra por sí misma en cualquier parte de la cadena", ya sea al final, al principio, o en alguna parte por en medio.

Footnotes

[3] *raw strings*

7.3. Caso de estudio: números romanos

Seguramente ha visto números romanos, incluso si no los conoce. Puede que los haya visto en copyrights de viejas películas y programas de televisión ("Copyright `MCMXLVI`" en lugar de "Copyright `1946`"), o en las placas conmemorativas en bibliotecas o universidades ("inaugurada en `MDCCCLXXXVIII`" en lugar de "inaugurada en `1888`"). Puede que incluso los haya visto en índices o referencias bibliográficas. Es un sistema de representar números que viene de los tiempos del antiguo imperio romano (de ahí su nombre).

En los números romanos, existen siete caracteres que representan números al combinarlos y repetirlos de varias maneras.

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

Se aplican las siguientes reglas generales al construir números romanos:

- Los caracteres son aditivos. `I` es `1`, `II` es `2`, y `III` es `3`. `VI` es `6` (literalmente, "`5` y `1`"), `VII` es `7`, y `VIII` es `8`.
- Los caracteres de la unidad y decenas (`I`, `X`, `C`, y `M`) pueden repetirse hasta tres veces. Para el `4`, debe restar del cinco inmediatamente superior. No puede representar `4` como `IIII`; en lugar deeso, se representa con `IV` ("`1` menos que `5`"). El número `40` se escribe como `XL` (`10` menos que `50`), `41` como `XLI`, `42` como `XLII`, `43` como `XLIII`, y `44` como `XLIV` (`10` menos que `50`, y `1` menos que `5`).
- De forma similar, para el `9`, debe restar de la decena inmediata: `8` es `VIII`, pero `9` es `IX` (`1` menos que `10`), no `VIIII` (ya que la `I` no se puede repetir cuatro veces). El número `90` es `XC`, `900` es `CM`.
- Los caracteres para "cinco" no se pueden repetir. El número `10` siempre se representa como `X`, nunca como `VV`. El número `100` siempre es `C`, nunca `LL`.
- Los números romanos siempre se escriben del mayor al menor, y se leen de izquierda a derecha, de manera que el orden de los caracteres importa mucho. `DC` es `600`; `CD` es un número completamente diferente (`400`, `100` menos que `500`). `CI` es `101`; `IC` no es siquiera un número romano válido (porque no puede restar `1` directamente de `100`; tendrá que escribirlo como `XCIX`, que es `10` menos que `100`, y `1` menos que `10`).

7.3.1. Comprobar los millares

¿Qué se necesitaría para certificar una cadena arbitraria como número romano válido? Miremos un carácter cada vez. Como los números romanos se escriben siempre de mayor a menor, empecemos con el mayor: el lugar de los millares. Para números del 1000 en adelante, los millares se representan con series de caracteres `M` characters.

Ejemplo 7.3. Comprobación de los millares

```
>>> import re
>>> pattern = '^M?M?M?$'       ❶
>>> re.search(pattern, 'M')    ❷
<SRE_Match object at 0106FB58>
>>> re.search(pattern, 'MM')   ❸
<SRE_Match object at 0106C290>
>>> re.search(pattern, 'MMM')  ❹
<SRE_Match object at 0106AA38>
>>> re.search(pattern, 'MMMM') ❺
>>> re.search(pattern, '')     ❻
<SRE_Match object at 0106F4A8>
```

❶ Este patrón tiene tres partes:

- `^` para hacer que lo que sigue coincida sólo con el comienzo de la cadena. Si no se especificase, el patrón podría coincidir con cualquiera de los caracteres `M` que hubiese, que no es lo que deseamos. Quiere asegurarse de que los caracteres `M`, si los hay ahí, estén al principio de la cadena.
- `M?` para coincidir con un único carácter `M` de forma opcional. Como se repite tres veces, estamos buscando cualquier cosa entre cero y tres caracteres `M` seguidos.
- `$` para que lo anterior preceda sólo al fin de la cadena. Cuando se combina con el carácter `^` al principio, significa que el patrón debe coincidir con la cadena al completo, sin otros caracteres antes o después de las `M`.

❷ La esencia del módulo `re` es la función `search`, que toma una expresión

regular (`pattern`) y una cadena (`'M'`) para comprobar si coincide con la expresión regular. Si se encuentra una coincidencia, `search` devuelve un objeto que tiene varios métodos para describirla; si no hay coincidencia, `search` devuelve `None`, el valor nulo de Python. Todo lo que ha de preocuparnos por ahora es si el patrón coincide, cosa que podemos saber con sólo mirar el valor devuelto por `search`. `'M'` coincide con esta expresión regular, porque se ajusta a la primera `M` opcional, mientras que se ignoran la segunda y tercera `M` opcionales.

❸ `'MM'` también coincide porque se ajusta a los primeros dos caracteres `M` opcionales, mientras que se ignora el tercero.

❹ `'MMM'` coincide porque se ajusta a los tres caracteres `M` del patrón.

❺ `'MMMM'` no coincide. Hay coincidencia con los tres caracteres `M`, pero la expresión regular insiste en que la cadena debe terminar ahí (debido al carácter `$`), pero la cadena aún no ha acabado (debido a la cuarta `M`). Así que `search` devuelve `None`.

❻ Interesante: una cadena vacía también coincide con esta expresión regular, ya que todos los caracteres `M` son opcionales.

7.3.2. Comprobación de centenas

Las centenas son más complejas que los millares, porque hay varias maneras mutuamente exclusivas de expresarlas, dependiendo de su valor.

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

Así que hay cuatro patrones posibles:

- `CM`
- `CD`
- De cero a tres caracteres `C` (cero si hay un 0 en el lugar de las centenas)
- `D`, seguido opcionalmente de hasta tres caracteres `C`

Los últimos dos patrones se pueden combinar:

- una `D` opcional, seguida de hasta tres caracteres `C` (opcionales también)

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

Ejemplo 7.4. Comprobación de las centenas

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

❶ Este patrón empieza igual que el anterior, comprobando el principio de lacadena, (`^`), luego el lugar de los millares (`M?M?M?`). Aquí viene la parte nueva, entre paréntesis, que define un conjunto de tres patrones mutuamente exclusivos, separados por barras verticales: `CM`, `CD`, y `D?C?C?C?` (que es una `D` opcional seguida de cero a tres caracteres `C` opcionales). El analizador de expresiones regulares comprueba cada uno de estos patrones en orden (de izquierda a derecha), toma el primero que coincida, y descarta el resto.

❷ `'MCM'` coincide porque la primera `M` lo hace, ignorando los dos siguientes caracteres `M`, y `CM` coincide (así que no se llegan a considerar los patrones `CD` ni

`D?C?C?C?`). `MCM` es la representación en números romanos de `1900`.

❸ `'MD'` coincide con la primera `M`, ignorando la segunda y tercera `M`, y el patrón `D?C?C?C?` coincide con la `D` (cada uno de los tres caracteres `C` son opcionales así que se ignoran). `MD` es la representación en romanos de `1500`.

❹ `'MMMCCC'` coincide con los tres primeros caracteres `M`, y con el patrón `D?C?C?C?` coinciden `CCC` (la `D` es opcional y se ignora). `MMMCCC` es la representación en números romanos de `3300`.

❺ `'MCMC'` no coincide. La primera `M` lo hace, las dos siguientes se ignoran, y también coincide `CM`, pero `$` no lo hace, ya que aún no hemos llegado al final de la cadena de caracteres (todavía nos queda un carácter `C` sin pareja). La `C` *no* coincide como parte del patrón `D?C?C?C?`, ya que se encontró antes el patrón `CM`, mutuamente exclusivo.

❻ Es interesante ver que la cadena vacía sigue coincidiendo con este patrón, porque todas las `M` son opcionales y se ignoran, y porque la cadena vacía coincide con el patrón `D?C?C?C?`, en el que todos los caracteres son opcionales, y por tanto también se ignoran.

¡Vaya! ¿Ve lo rápido que las expresiones regulares se vuelven feas? Y sólo hemos cubierto los millares y las centenas de los números romanos. Pero si ha seguido el razonamiento, las decenas y las unidades son sencillas, porque siguen exactamente el mismo patrón de las centenas. Pero veamos otra manera de expresarlo.

7.4. Uso de la sintaxis `{n,m}`

En la sección anterior, tratamos con un patrón donde el mismo carácter podía repetirse hasta tres veces. Hay otra manera de expresar esto con expresiones regulares, que algunas personas encuentran más legible. Primero mire el método que hemos usado ya en los ejemplos anteriores.

Ejemplo 7.5. La manera antigua: cada carácter es opcional

```
>>> import re
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'M')    ❶
<_sre.SRE_Match object at 0x008EE090>
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'MM')   ❷
<_sre.SRE_Match object at 0x008EEB48>
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'MMM')  ❸
<_sre.SRE_Match object at 0x008EE090>
>>> re.search(pattern, 'MMMM') ❹
>>>
```

❶ Esto coincide con el inicio de la cadena, luego la primera `M` opcional, pero no la segunda y la tercera (pero no pasa nada, porque son opcionales), y luego el fin de la cadena.

❷ Esto coincide con el inicio de la cadena, luego la primera y segunda `M` opcionales, pero no la tercera `M` (pero no pasa nada, porque es opcional), y luego el fin de la cadena.

❸ Esto coincide con el principio de la cadena, y luego con las tres `M` opcionales, antes del fin de la cadena.

❹ Esto coincide con el principio de la cadena, y luego las tres `M` opcionales, pero no encontramos el fin de la cadena (porque aún hay una `M` sin emparejar), así que el patrón no coincide y devuelve `None`.

Ejemplo 7.6. La nueva manera: de `n` a `m`

```
>>> pattern = '^M{0,3}$'       ❶
>>> re.search(pattern, 'M')    ❷
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MM')   ❸
<_sre.SRE_Match object at 0x008EE090>
>>> re.search(pattern, 'MMM')  ❹
<_sre.SRE_Match object at 0x008EEDA8>
>>> re.search(pattern, 'MMMM') ❺
>>>
```

❶ Este patrón dice: "Coincide con el principio de la cadena, luego cualquier cosa entre cero y tres caracteres `M`, y luego el final de la cadena". 0 y 3 podrían ser números cualquiera; si queremos que haya al menos una `M` pero no más de tres, podríamos escribir `M{1,3}`.

❷ Esto coincide con el principio de la cadena, luego una de tres posibles `M`, y luego el final de la cadena.

❸ Esto coincide con el principio de la cadena, luego dos de tres posibles `M`, y luego el final de la cadena.

❹ Esto coincide con el principio de la cadena, luego tres de tres posibles `M`, y luego el final de la cadena.

❺ Esto coincide con el principio de la cadena, luego tres de tres posibles `M`, pero entonces *no encuentra* el final de la cadena. La expresión regular nos permitía sólo hasta tres caracteres `M` antes de encontrar el fin de la cadena, pero tenemos cuatro, así que el patrón no coincide y se devuelve `None`.

No hay manera de determinar programáticamente si dos expresiones regulares son equivalentes. Lo mejor que puede hacer es escribir varios casos de prueba para asegurarse de que se comporta correctamente con todos los tipos de entrada relevantes. Hablaremos más adelante en este mismo libro sobre la escritura de casos de prueba.

7.4.1. Comprobación de las decenas y unidades

Ahora expandiremos la expresión regular de números romanos para cubrir las decenas y unidades. Este ejemplo muestra la comprobación de las decenas.

Ejemplo 7.7. Comprobación de las decenas

```
>>> pattern = '^M?M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)$'
>>> re.search(pattern, 'MCMXL')    ❶
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCML')     ❷
<_sre.SRE_Match object at 0x008EEB48>
```

```
>>> re.search(pattern, 'MCMLX')    ❸
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLXXX')  ❹
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLXXXX') ❺
>>>
```

❶ Esto coincide con el principio de la cadena, luego la primera `M` opcional, después con `CM`, y `XL`, para coincidir por último con el fin de la cadena. Recuerde, la sintaxis `(A|B|C)` significa "coincide exactamente con un patrón de entre A, B o C". Encontramos `XL`, así que se ignoran las posibilidades `XC` y `L?X?X?X?`, y después pasamos al final de la cadena. `MCML` es el número romano que representa `1940`.

❷ Esto coincide con el principio de la cadena, y la primera `M` opcional, después `CM`, y `L?X?X?X?`. De `L?X?X?X?`, coincide con la `L` y descarta los tres caracteres opcionales `X`. Ahora pasamos al final de la cadena. `MCML` es la representación en romanos de `1950`.

❸ Esto coincide con el principio de la cadena, luego con la primera `M` opcional, y con `CM`, después con la `L` opcional y con la primera `X` opcional, ignora las otras dos `X` opcionales y luego encuentra el final de la cadena. `MCMLX` representa en números romanos `1960`.

❹ Esto encuentra el principio de la cadena, luego la primera `M` opcional, después `CM`, luego la `L` opcional y las tres `X` opcionales, y por último el fin de la cadena. `MCMLXXX` es el número romano que representa `1980`.

❺ Esto coincide con el principio de la cadena, y la primera `M` opcional, luego `CM`, y después la `L` y las tres `X` opcionales, para entonces *fallar al intentar coincidir* con el fin de la cadena, debido a que aún resta una `X` sin justificar. Así que el patrón al completo falla, y se devuelve `None`. `MCMLXXXX` no es un número romano válido.

La expresión de las unidades sigue el mismo patrón. Le ahorraré los detalles mostrándole el resultado final.

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

¿Cómo se vería esto usando su sintaxis alternativa `{n,m}`? El ejemplo nos muestra la nueva sintaxis.

Ejemplo 7.8. Validación de números romanos con `{n,m}`

```
>>> pattern =
'^M{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$'
>>> re.search(pattern, 'MDLV')              ❶
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MMDCLXVI')          ❷
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MMMMDCCCLXXXVIII') ❸
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'I')                 ❹
<_sre.SRE_Match object at 0x008EEB48>
```

❶ Esto coincide con el principio de la cadena, después con uno de las cuatro `M` posibles, luego con `D?C{0,3}`. De estos, coincide con la `D` opcional y ninguna de las tres posibles `C`. Seguimos, y coincide con `L?X{0,3}` con la `L` opcional y ninguna de las tres posibles `X`. Entonces coincide con `V?I{0,3}` al encontrar la `V` opcional y ninguna de las tres posibles `I`, y por último el fin de la cadena. `MDLV` es la representación en romanos de `1555`.

❷ Esto coincide con el principio de la cadena, luego dos de las cuatro posibles `M`, entonces con `D?C{0,3}` por una `D` y una `C` de tres posibles; luego con `L?X{0,3}` por la `L` y una de tres `X` posibles; después con `V?I{0,3}` por una `V` y una de tres posibles `I`; y por último, con el fin de la cadena. `MMDCLXVI` es la representación en números romanos de `2666`.

❸ Esto coincide con el principio de la cadena, luego con cuatro de cuatro `M` posibles, después con `D?C{0,3}` por una `D` y tres de tres posibles `C`; entonces con `L?X{0,3}` por la `L` y las tres `X` de tres; luego con `V?I{0,3}` por la `V` y tres de tres `I`; y por último con el final de la cadena. `MMMMDCCCLXXXVIII` es la representación en números romanos de `3888`, y es el número romano más largo que se puede escribir sin usar una sintaxis extendida.

❹ Observe atentamente (me siento como un mago. “Observen atentamente, niños, voy a sacar un conejo de mi sombrero.”). Esto coincide con el principio

de la cadena, y con ninguna de las cuatro posibles `M`, luego con `D?C{0,3}` al ignorar la `D` y coincidiendo con cero de tres `C`, después con `L?X{0,3}` saltándose la `L` opcional y con cero de tres `X`, seguidamente con `V?I{0,3}` ignorando la `V` opcional y una de tres posibles `I`. Y por último, el fin de la cadena. ¡Guau!.

Si ha conseguido seguirlo todo y lo ha entendido a la primera, ya lo hizo mejor que yo. Ahora imagínese tratando de entender las expresiones regulares de otra persona, en mitad de una parte crítica de un programa grande. O incluso imagínese encontrando de nuevo sus propias expresiones regulares unos meses más tarde. Me ha sucedido, y no es una visión placentera.

En la siguiente sección exploraremos una sintaxis alternativa que le puede ayudar a hacer mantenibles sus expresiones.

7.5. Expresiones regulares prolijas

Hasta ahora sólo hemos tratado con lo que llamaremos expresiones regulares "compactas". Como verá, son difíciles de leer, e incluso si uno sabe lo que hacen, eso no garantiza que seamos capaces de comprenderlas dentro de seis meses. Lo que estamos necesitando es documentación en línea.

Python le permite hacer esto con algo llamado *expresiones regulares prolijas*. Una expresión regular prolija es diferente de una compacta de dos formas:

- Se ignoran los espacios en blanco. Los espacios, tabuladores y retornos de carro no coinciden como tales. De hecho, no coinciden con nada (si quiere hacer coincidir un espacio en una expresión prolija, necesitará escaparlo poniendo una barra inversa delante).
- Se ignoran los comentarios. Un comentario de una expresión regular prolija es exactamente como uno en el código de Python: comienza con un carácter # y continúa hasta el final de la línea. En este caso es un comentario dentro de una cadena de múltiples líneas en lugar de dentro de su propio código, pero funciona de la misma mamera.

Esto quedará más claro con un ejemplo. Volvamos sobre la expresión regular compacta con la que hemos estado trabajando, y convirtámosla en una prolija. Este ejemplo le muestra cómo.

Ejemplo 7.9. Expresiones regulares con comentarios en línea

```
>>> pattern = """
    ^                   # beginning of string
    M{0,4}              # thousands - 0 to 4 M's
    (CM|CD|D?C{0,3})    # hundreds - 900 (CM), 400 (CD), 0-300 (0 to 3
C's),
                        #            or 500-800 (D, followed by 0 to 3
C's)
    (XC|XL|L?X{0,3})    # tens - 90 (XC), 40 (XL), 0-30 (0 to 3 X's),
                        #        or 50-80 (L, followed by 0 to 3 X's)
    (IX|IV|V?I{0,3})    # ones - 9 (IX), 4 (IV), 0-3 (0 to 3 I's),
                        #        or 5-8 (V, followed by 0 to 3 I's)
    $                   # end of string
    """
>>> re.search(pattern, 'M', re.VERBOSE)                ❶
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLXXXIX', re.VERBOSE)        ❷
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MMMMDCCCLXXXVIII', re.VERBOSE) ❸
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'M')                            ❹
```

❶ Lo más importante que ha de recordar cuando use expresiones regulares prolijas es que debe pasar un argumento extra cuando trabaje con ellas: `re.VERBOSE` es una constante definida en el módulo `re` que indica que un patrón debe ser tratado como una expresión prolija. Como puede ver, este patrón tiene bastante espacio en blanco (que se ignora por completo), y varios comentarios (que también se ignoran). Una vez ignorado el espacio en blanco y los comentarios, es exactamente la misma expresión regular que vimos en la sección anterior, pero es mucho más legible.

❷ Esto coincide con el inicio de la cadena, luego una de cuatro posibles `M`, despuéscon `CM`, `L` y tres de tres `X`, `IX`, y el final de la cadena.

❸ Esto coincide con el principio de lacadena, las cuatro posibles `M`, `D` y tres de

tres `C`, `L` y las tres posibles `X`, `V` y las tres `I` disponibles, y el final de la cadena.

❹ Esto no coincide. ¿Por qué? Porque no tiene el indicador `re.VERBOSE`, de manera que la función `re.search` está tratando el patrón como una expresión regular compacta, donde los espacios y las marcas # tienen significado. Python no puede detectar por sí solo si una expresión regular es prolija o no. Python asume que cada expresión es compacta a menos que le indice explícitamente que es prolija.

7.6. Caso de estudio: análisis de números de teléfono

Por ahora se ha concentrado en patrones completos. Cada patrón coincide, o no. Pero las expresiones regulares son mucho más potentes que eso. Cuando una expresión regular *coincide*, puede extraer partes concretas. Puede saber qué es lo que causó la coincidencia.

Este ejemplo sale de otro problema que he encontrado en el mundo real, de nuevo de un empleo anterior. El problema: analizar un número de teléfono norteamericano. El cliente quería ser capaz de introducir un número de forma libre (en un único campo), pero quería almacenar por separado el código de área, la troncal, el número y una extensión opcional en la base de datos de la compañía. Rastreando la Web encontré muchos ejemplo de expresiones regulares que supuestamente conseguían esto, pero ninguna era lo suficientemente permisiva.

Éstos son los números de teléfono que había de poder aceptar:

- `800-555-1212`
- `800 555 1212`
- `800.555.1212`
- `(800) 555-1212`
- `1-800-555-1212`
- `800-555-1212-1234`
- `800-555-1212x1234`

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

¡Qué gran variedad! En cada uno de estos casos, necesitaba saber que el código de área era `800`, la troncal `555`, y el resto del número de teléfono era `1212`. Para aquellos con extensión, necesitaba saber que ésta era `1234`.

Vamos a desarrollar una solución para el análisis de números de teléfono. Este ejemplo le muestra el primer paso.

Ejemplo 7.10. Finding Numbers

```
>>> phonePattern = re.compile(r'^(\d{3})-(\d{3})-(\d{4})$') ❶
>>> phonePattern.search('800-555-1212').groups()            ❷
('800', '555', '1212')
>>> phonePattern.search('800-555-1212-1234')                ❸
>>>
```

❶ Lea siempre una expresión regular de izquierda a derecha. Ésta coincide con el comienzo de la cadena, y luego `(\d{3})`. ¿Qué es `\d{3}`? Bien, el `{3}` significa "coincidir con exactamente tres caracteres"; es una variante de la sintaxis `{n,m}` que vimos antes. `\d` significa "un dígito numérico" (de `0` a `9`). Ponerlo entre paréntesis indica "coincide exactamente con tres dígitos numéricos *y recuérdalos como un grupo que luego te los voy a pedir*". Luego coincide con un guión. Después con otro grupo de exactamente tres dígitos. Luego con otro guión. Entonces con otro grupo de exactamente cuatro dígitos. Y ahora con el final de la cadena.

❷ Para acceder a los grupos que ha almacenado el analizador de expresiones por el camino, utilice el método `groups()` del objeto que devuelve la función `search`. Obtendrá una tupla de cuantos grupos haya definido en la expresión regular. En este caso, hemos definido tres grupos, un con tres dígitos, otro con tres dígitos, y otro más con cuatro dígitos.

❸ Esta expresión regular no es la respuesta final, porque no trabaja con un número de teléfono con extensión al final. Para eso, hace falta aumentar la expresión.

Ejemplo 7.11. Búsqueda de la extensión

```
>>> phonePattern = re.compile(r'^(\d{3})-(\d{3})-(\d{4})-(\d+)$') ❶
>>> phonePattern.search('800-555-1212-1234').groups()             ❷
('800', '555', '1212', '1234')
>>> phonePattern.search('800 555 1212 1234')                      ❸
>>>
>>> phonePattern.search('800-555-1212')                           ❹
>>>
```

❶ Esta expresión regular es casi idéntica a la anterior. Igual que antes buscamos el comienzo de la cadena, luego recordamos un grupo de tres dígitos, luego un guión, un grupo de tres dígitos a recordar, un guión, un grupo de cuatro dígitos a recordar. Lo nuevo es que ahora buscamos otro guión, y un grupo a recordar de uno o más dígitos, y por último el final de la cadena.

❷ El método `groups()` devuelve ahora una tupla de cuatro elementos, ya que la expresión regular define cuatro grupos a recordar.

❸ Desafortunadamente, esta expresión regular tampoco es la respuesta definitiva, porque asume que las diferentes partes del número de teléfono las separan guiones. ¿Qué pasa si las separan espacios, comas o puntos? Necesita una solución más general para coincidir con varios tipos de separadores.

❹ ¡Vaya! No sólo no hace todo lo que queremos esta expresión, sino que incluso es un paso atrás, porque ahora no podemos reconocer números *sin* una extensión. Eso no es lo que queríamos; si la extensión está ahí, queremos saberla, pero si no, aún queremos saber cuales son las diferentes partes del número principal.

El siguiente ejemplo muestra la expresión regular que maneja separadores entre diferentes partes del número de teléfono.

Ejemplo 7.12. Manejo de diferentes separadores

```
>>> phonePattern =
re.compile(r'^(\d{3})\D+(\d{3})\D+(\d{4})\D+(\d+)$') ❶
```

```
>>> phonePattern.search('800 555 1212 1234').groups()
❷
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212-1234').groups()
❸
('800', '555', '1212', '1234')
>>> phonePattern.search('80055512121234')
❹
>>>
>>> phonePattern.search('800-555-1212')
❺
>>>
```

❶ Agárrese el sombrero. Estamos buscando el comienzo de la cadena, luego un grupo de tres dígitos, y después `\D+`. ¿Qué diantre es eso? Bien, `\D` coincide con cualquier carácter *excepto* un dígito numérico, y `+` significa "1 o más". Así que `\D+` coincide con uno o más caracteres que no sean dígitos. Esto es lo que vamos a usar en lugar de un guión, para admitir diferentes tipos de separadores.

❷ Usar `\D+` en lugar de `-` implica que ahora podemos reconocer números de teléfonos donde los números estén separados por espacios en lugar de guiones.

❸ Por supuesto, los números separados por guión siguen funcionando.

❹ Desafortunadamente, aún no hemos terminado, porque asume que hay separadores. ¿Qué pasa si se introduce el número de teléfono sin ningún tipo de separador?

❹ ¡Vaya! Aún no hemos corregido el problema de la extensión obligatoria. Ahora tenemos dos problemas, pero podemos resolverlos ambos con la misma técnica.

El siguiente ejemplo muestra la expresión regular para manejar números de teléfonos *sin* separadores.

Ejemplo 7.13. Manejo de números sin separadores

```
>>> phonePattern =
re.compile(r'^(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$') ❶
>>> phonePattern.search('80055512121234').groups()
❷
('800', '555', '1212', '1234')
>>> phonePattern.search('800.555.1212 x1234').groups()
❸
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212').groups()
❹
('800', '555', '1212', '')
>>> phonePattern.search('(800)5551212 x1234')
❺
>>>
```

❶ La única modificación que hemos hecho desde el último paso es cambiar todos los `+` por `*`. En lugar de buscar `\D+` entre las partes del número de teléfono, ahora busca `\D*`. ¿Recuerda que `+` significa "1 o más"? Bien, `*` significa "cero o más". Así que ahora es capaz de reconocer números de teléfono incluso si no hay separadores de caracteres.

❷ Espere un momento, esto funciona. ¿Por qué? Coincidimos con el principio de la cadena, y luego recordamos un grupo de tres dígitos (`800`), después cero caracteres no numéricos, luego un grupo de tres dígitos (`555`), ahora cero caracteres no numéricos, un grupo de cuatro dígitos (`1212`), cero caracteres no numéricos y un grupo arbitrario de dígitos (`1234`), y después el final de la cadena.

❸ También funcionan otras variantes: puntos en lugar de guiones, y tanto espacios como una `x` antes de la extensión.

❹ Por último, hemos resuelto el otro problema que nos ocupaba: las extensiones vuelven a ser opcionales. Si no se encuentra una extensión, el método `groups()` sigue devolviendo una tupla de 4 elementos, pero el cuarto es simplemente una cadena vacía.

❺ Odio ser portador de malas noticias, pero aún no hemos terminado. ¿Cual es el problema aquí? Hay un carácter adicional antes del código de área, pero la expresión regular asume que el código de área es lo primero que hay al

empezar la cadena. No hay problema, podemos usar la misma técnica de "cero o más caracteres no numéricos" para eliminar los caracteres del que hay antes del código de área.

El siguiente ejemplo muestra cómo manejar los caracteres antes del número de teléfono.

Ejemplo 7.14. Manipulación de caracteres iniciales

```
>>> phonePattern =
re.compile(r'^\D*(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$') ❶
>>> phonePattern.search('(800)5551212 ext. 1234').groups()
❷
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212').groups()
❸
('800', '555', '1212', '')
>>> phonePattern.search('work 1-(800) 555.1212 #1234')
❹
>>>
```

❶ Esto es lo mismo que en el ejemplo anterior, excepto que ahora empezamos con `\D*`, cero o más caracteres no numéricos, antes del primer grupo que hay que recordar (el código de área). Observe que no estamos recordando estos caracteres no numéricos (no están entre paréntesis). Si los encontramos, simplemente los descartaremos y empezaremos a recordar el código de área en cuanto lleguemos a él.

❷ Puede analizar con éxito el número de teléfono, incluso con el paréntesis abierto a la izquierda del código de área. (El paréntesis de la derecha también se tiene en cuenta; se le trata como un separador no numérico y coincide con el `\D*` tras el primer grupo a recordar).

❸ Un simple control para asegurarnos de no haber roto nada que ya funcionase. Como los caracteres iniciales son totalmente opcionales, esto coincide con el principio de la cadena, luego vienen cero caracteres no numéricos, después un grupo de tres dígitos a recordar (`800`), luego un carácter no numérico (el guión), un grupo de tres dígitos a recordar (`555`), un

carácter no numérico (el guión), un grupo de cuatro dígitos a recordar (`1212`), cero caracteres no numéricos, un grupo a recordar de cero dígitos, y el final de la cadena.

❹ Aquí es cuando las expresiones regulares me hacen desear sacarme los ojos con algún objeto romo. ¿Por qué no funciona con este número de teléfono? Porque hay un `1` antes del código de área, pero asumimos que todos los caracteres antes de ese código son no numéricos (`\D*`). Aggghhh.

Parémonos a pensar por un momento. La expresión regular hasta ahora ha buscado coincidencias partiendo siempre del inicio de la cadena. Pero ahora vemos que hay una cantidad indeterminada de cosas al principio de la cadena que queremos ignorar. En lugar de intentar ajustarlo todo para simplemente ignorarlo, tomemos un enfoque diferente: no vamos a buscar coincidencias explícitamente desde el principio de la cadena. Esto lo mostramos en el siguiente ejemplo.

Ejemplo 7.15. Número de teléfono, dónde te he de encontrar

```
>>> phonePattern = re.compile(r'(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$')
❶
>>> phonePattern.search('work 1-(800) 555.1212 #1234').groups()
❷
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212')
❸
('800', '555', '1212', '')
>>> phonePattern.search('80055512121234')
❹
('800', '555', '1212', '1234')
```

❶ Observe la ausencia de `^` en esta expresión regular. Ya no buscamos el principio de la cadena. No hay nada que diga que tenemos que hacer que la entrada completa coincida con nuestra expresión regular. El motor de expresiones regulares hará el trabajo duro averiguando desde dónde ha de empezar a comparar en la cadena de entrada, y seguirá de ahí en adelante.

❷ Ahora ya podemos analizar con éxito un número teléfono que contenga otros

caracteres y dígitos antes, además de cualquier cantidad de cualquier tipo de separadores entre cada parte del número de teléfono.

❸ Comprobación de seguridad. Esto aún funciona.

❹ Esto también funciona.

¿Ve lo rápido que pueden descontrolarse las expresiones regulares? Eche un vistazo rápido a cualquiera de las iteraciones anteriores. ¿Podría decir la diferencia entre ésa y la siguiente?

Aunque entienda la respuesta final (y es la definitiva; si ha descubierto un caso que no se ajuste, yo no quiero saber nada), escribámoslo como una expresión regular prolija, antes de que olvidemos por qué hicimos cada elección.

Ejemplo 7.16. Análisis de números de teléfono (versión final)

```
>>> phonePattern = re.compile(r'''
                # don't match beginning of string, number can start
anywhere
    (\d{3})     # area code is 3 digits (e.g. '800')
    \D*         # optional separator is any number of non-digits
    (\d{3})     # trunk is 3 digits (e.g. '555')
    \D*         # optional separator
    (\d{4})     # rest of number is 4 digits (e.g. '1212')
    \D*         # optional separator
    (\d*)       # extension is optional and can be any number of
digits
    $           # end of string
    ''', re.VERBOSE)
>>> phonePattern.search('work 1-(800) 555.1212 #1234').groups()
❶
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212')
❷
('800', '555', '1212', '')
```

❶ Aparte de ocupar varias líneas, ésta es exactamente la misma expresión regular del paso anterior, de manera que no sorprende que analice las mismas entradas.

❷ Comprobación de seguridad final. Sí, sigue funcionando. Lo hemos conseguido.

Lecturas complementarias sobre expresiones regulares

- El Regular Expression HOWTO le instruye sobre expresiones regulares y su uso en Python.
- La *Referencia de bibliotecas de Python* expone el módulo `re`.

7.7. Resumen

Ésta es sólo la minúscula punta del iceberg de lo que pueden hacer las expresiones regulares. En otras palabras, incluso aunque esté completamente saturado con ellas ahora mismo, créame, todavía no ha visto nada.

Ahora deberían serle familiares las siguientes técnicas:

- `^` coincide con el principio de una cadena.
- `$` coincide con el final de una cadena.
- `\b` coincide con el límite de una palabra.
- `\d` coincide con cualquier dígito numérico.
- `\D` coincide con cualquier carácter no numérico.
- `x?` coincide con un carácter `x` opcional (en otras palabras, coincide con `x` una o ninguna vez).
- `x*` coincide con `x` cero o más veces.
- `x+` coincide con `x` una o más veces.
- `x{n,m}` coincide con un carácter `x` al menos `n` pero no más de `m` veces.
- `(a|b|c)` coincide sólo con una entre `a`, `b` o `c`.
- `(x)` en general es un *grupo a recordar*. Puede obtener el valor de la coincidencia usando el método `groups()` del objeto devuelto por `re.search`.

Las expresiones regulares son muy potentes, pero no son la solución adecuada para todos los problema. Debería aprender de ellas lo suficiente como para

saber cuándo es apropiado usarlas, cuándo resolverán sus problemas, y cuándo causarán más problemas de los que resuelven.

> Algunas personas, cuando se enfrentan a un problema, piensan "ya lo sé, voy a usar expresiones regulares". Ahora tienen dos problemas.
>
> --Jamie Zawinski, en comp.emacs.xemacs

Capítulo 8. Procesamiento de HTML

8.1. Inmersión

A menudo veo preguntas en comp.lang.python parecidas a "¿Cómo puedo obtener una lista de todas las [cabeceras | imágenes | enlaces] en mi documento HTML?" "¿Cómo analizo/traduzco/manipulo el texto de mi documento HTML pero sin tocar las etiquetas?" "¿Cómo puedo añadir/eliminar/poner comillas a los atributos de mis etiquetas HTML de una sola vez?" Este capítulo responderá todas esas preguntas.

Aquí tiene un programa en Python completo y funcional, en dos partes. La primera, `BaseHTMLProcessor.py`, es una herramienta genérica para ayudarle a procesar ficheros HTML iterando sobre las etiquetas y los bloques de texto. La segunda parte, `dialect.py`, es un ejemplo de uso de `BaseHTMLProcessor.py` para traducir el texto de un documento HTML pero sin tocar las etiquetas. Lea las `cadenas de documentación` y los comentarios para hacerse una idea de lo que está sucediendo. La mayoría parecerá magia negra, porque no es obvia la manera en que se invoca a los métodos de estas clases. No se preocupe, todo se revelará a su debido tiempo.

Ejemplo 8.1. `BaseHTMLProcessor.py`

Si aún no lo ha hecho, puede descargar éste ejemplo y otros usados en este libro.

```
from sgmllib import SGMLParser
import htmlentitydefs

class BaseHTMLProcessor(SGMLParser):
    def reset(self):
        # extend (called by SGMLParser.__init__)
        self.pieces = []
        SGMLParser.reset(self)

    def unknown_starttag(self, tag, attrs):
        # called for each start tag
        # attrs is a list of (attr, value) tuples
        # e.g. for <pre class="screen">, tag="pre", attrs=[("class",
"screen")]
        # Ideally we would like to reconstruct original tag and
attributes, but
        # we may end up quoting attribute values that weren't quoted
in the source
        # document, or we may change the type of quotes around the
attribute value
        # (single to double quotes).
        # Note that improperly embedded non-HTML code (like client-
side Javascript)
        # may be parsed incorrectly by the ancestor, causing runtime
script errors.
        # All non-HTML code must be enclosed in HTML comment tags (<!-
- code -->)
        # to ensure that it will pass through this parser unaltered
(in handle_comment).
        strattrs = "".join([' %s="%s"' % (key, value) for key, value
in attrs])
        self.pieces.append("<%(tag)s%(strattrs)s>" % locals())

    def unknown_endtag(self, tag):
        # called for each end tag, e.g. for </pre>, tag will be "pre"
        # Reconstruct the original end tag.
        self.pieces.append("</%(tag)s>" % locals())
```

```
    def handle_charref(self, ref):
        # called for each character reference, e.g. for " ", ref
will be "160"
        # Reconstruct the original character reference.
        self.pieces.append("&#%(ref)s;" % locals())

    def handle_entityref(self, ref):
        # called for each entity reference, e.g. for "&copy;", ref
will be "copy"
        # Reconstruct the original entity reference.
        self.pieces.append("&%(ref)s" % locals())
        # standard HTML entities are closed with a semicolon; other
entities are not
        if htmlentitydefs.entitydefs.has_key(ref):
            self.pieces.append(";")

    def handle_data(self, text):
        # called for each block of plain text, i.e. outside of any tag
and
        # not containing any character or entity references
        # Store the original text verbatim.
        self.pieces.append(text)

    def handle_comment(self, text):
        # called for each HTML comment, e.g. <!-- insert Javascript
code here -->
        # Reconstruct the original comment.
        # It is especially important that the source document enclose
client-side
        # code (like Javascript) within comments so it can pass
through this
        # processor undisturbed; see comments in unknown_starttag for
details.
        self.pieces.append("<!--%(text)s-->" % locals())

    def handle_pi(self, text):
        # called for each processing instruction, e.g. <?instruction>
        # Reconstruct original processing instruction.
        self.pieces.append("<?%(text)s>" % locals())

    def handle_decl(self, text):
        # called for the DOCTYPE, if present, e.g.
```

```
        # <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN"
        #     "http://www.w3.org/TR/html4/loose.dtd">
        # Reconstruct original DOCTYPE
        self.pieces.append("<!%(text)s>" % locals())

    def output(self):
        """Return processed HTML as a single string"""
        return "".join(self.pieces)
```

Ejemplo 8.2. `dialect.py`

```
import re
from BaseHTMLProcessor import BaseHTMLProcessor

class Dialectizer(BaseHTMLProcessor):
    subs = ()

    def reset(self):
        # extend (called from __init__ in ancestor)
        # Reset all data attributes
        self.verbatim = 0
        BaseHTMLProcessor.reset(self)

    def start_pre(self, attrs):
        # called for every <pre> tag in HTML source
        # Increment verbatim mode count, then handle tag like normal
        self.verbatim += 1
        self.unknown_starttag("pre", attrs)

    def end_pre(self):
        # called for every </pre> tag in HTML source
        # Decrement verbatim mode count
        self.unknown_endtag("pre")
        self.verbatim -= 1

    def handle_data(self, text):
        # override
        # called for every block of text in HTML source
        # If in verbatim mode, save text unaltered;
        # otherwise process the text with a series of substitutions
```

```
        self.pieces.append(self.verbatim and text or
self.process(text))

    def process(self, text):
        # called from handle_data
        # Process text block by performing series of regular
expression
        # substitutions (actual substitions are defined in descendant)
        for fromPattern, toPattern in self.subs:
            text = re.sub(fromPattern, toPattern, text)
        return text

class ChefDialectizer(Dialectizer):
    """convert HTML to Swedish Chef-speak

    based on the classic chef.x, copyright (c) 1992, 1993 John
Hagerman
    """
    subs = ((r'a([nu])', r'u\1'),
            (r'A([nu])', r'U\1'),
            (r'a\B', r'e'),
            (r'A\B', r'E'),
            (r'en\b', r'ee'),
            (r'\Bew', r'oo'),
            (r'\Be\b', r'e-a'),
            (r'\be', r'i'),
            (r'\bE', r'I'),
            (r'\Bf', r'ff'),
            (r'\Bir', r'ur'),
            (r'(\w*?)i(\w*?)$', r'\1ee\2'),
            (r'\bow', r'oo'),
            (r'\bo', r'oo'),
            (r'\bO', r'Oo'),
            (r'the', r'zee'),
            (r'The', r'Zee'),
            (r'th\b', r't'),
            (r'\Btion', r'shun'),
            (r'\Bu', r'oo'),
            (r'\BU', r'Oo'),
            (r'v', r'f'),
            (r'V', r'F'),
            (r'w', r'w'),
```

```
            (r'W', r'W'),
            (r'([a-z])[.]', r'\1.  Bork Bork Bork!'))

class FuddDialectizer(Dialectizer):
    """convert HTML to Elmer Fudd-speak"""
    subs = ((r'[rl]', r'w'),
            (r'qu', r'qw'),
            (r'th\b', r'f'),
            (r'th', r'd'),
            (r'n[.]', r'n, uh-hah-hah-hah.'))

class OldeDialectizer(Dialectizer):
    """convert HTML to mock Middle English"""
    subs = ((r'i([bcdfghjklmnpqrstvwxyz])e\b', r'y\1'),
            (r'i([bcdfghjklmnpqrstvwxyz])e', r'y\1\1e'),
            (r'ick\b', r'yk'),
            (r'ia([bcdfghjklmnpqrstvwxyz])', r'e\1e'),
            (r'e[ea]([bcdfghjklmnpqrstvwxyz])', r'e\1e'),
            (r'([bcdfghjklmnpqrstvwxyz])y', r'\1ee'),
            (r'([bcdfghjklmnpqrstvwxyz])er', r'\1re'),
            (r'([aeiou])re\b', r'\1r'),
            (r'ia([bcdfghjklmnpqrstvwxyz])', r'i\1e'),
            (r'tion\b', r'cioun'),
            (r'ion\b', r'ioun'),
            (r'aid', r'ayde'),
            (r'ai', r'ey'),
            (r'ay\b', r'y'),
            (r'ay', r'ey'),
            (r'ant', r'aunt'),
            (r'ea', r'ee'),
            (r'oa', r'oo'),
            (r'ue', r'e'),
            (r'oe', r'o'),
            (r'ou', r'ow'),
            (r'ow', r'ou'),
            (r'\bhe', r'hi'),
            (r've\b', r'veth'),
            (r'se\b', r'e'),
            (r"'s\b", r'es'),
            (r'ic\b', r'ick'),
            (r'ics\b', r'icc'),
            (r'ical\b', r'ick'),
```

```
            (r'tle\b', r'til'),
            (r'll\b', r'l'),
            (r'ould\b', r'olde'),
            (r'own\b', r'oune'),
            (r'un\b', r'onne'),
            (r'rry\b', r'rye'),
            (r'est\b', r'este'),
            (r'pt\b', r'pte'),
            (r'th\b', r'the'),
            (r'ch\b', r'che'),
            (r'ss\b', r'sse'),
            (r'([wybdp])\b', r'\1e'),
            (r'([rnt])\b', r'\1\1e'),
            (r'from', r'fro'),
            (r'when', r'whan'))

def translate(url, dialectName="chef"):
    """fetch URL and translate using dialect

    dialect in ("chef", "fudd", "olde")"""
    import urllib
    sock = urllib.urlopen(url)
    htmlSource = sock.read()
    sock.close()
    parserName = "%sDialectizer" % dialectName.capitalize()
    parserClass = globals()[parserName]
    parser = parserClass()
    parser.feed(htmlSource)
    parser.close()
    return parser.output()

def test(url):
    """test all dialects against URL"""
    for dialect in ("chef", "fudd", "olde"):
        outfile = "%s.html" % dialect
        fsock = open(outfile, "wb")
        fsock.write(translate(url, dialect))
        fsock.close()
        import webbrowser
        webbrowser.open_new(outfile)

if __name__ == "__main__":
```

```
    test("http://diveintopython.org/odbchelper_list.html")
```

Ejemplo 8.3. Salida de `dialect.py`

La ejecución de este script traducirá Sección 3.2, "Presentación de las listas" a la jerga del Cocinero Sueco (de los Teleñecos), la de Elmer Fudd (de Bugs Bunny), y a un Inglés Medieval en broma (basado lejanamente en los *Cuentos de Canterbury* de Chaucer). Si observa el código HTML de las página de salida, observará que no se han tocado las etiquetas HTML ni sus atributos, pero se ha "traducido" el texto que hay entre ellas al idioma bufonesco. Si mira más atentamente, verá que, de hecho, sólo se tradujeron los títulos y párrafos; sin tocar los listados de código y ejemplos de pantalla.

```
<div class="abstract">
<p>Lists awe <span class="application">Pydon</span>'s wowkhowse
datatype.
If youw onwy expewience wif wists is awways in
<span class="application">Visuaw Basic</span> ow (God fowbid) de
datastowe
in <span class="application">Powewbuiwdew</span>, bwace youwsewf fow
<span class="application">Pydon</span> wists.</p>
</div>
```

8.2. Presentación de `sgmllib.py`

El procesamiento de HTML se divide en tres pasos: obtener del HTML sus partes constitutivas, manipular las partes y reconstituirlas en un documento HTML. El primero paso lo realiza `sgmllib.py`, una parte de la biblioteca estándar de Python.

La clave para comprender este capítulo es darse cuenta de que HTML no es sólo texto, sino texto estructurado. La estructura se deriva de una secuencia más o menos jerárquica de etiquetas de inicio y de final. Normalmente no trabajará con HTML de esta manera; trabajará con él *de forma textual* en un editor de texto, o *visualmente* en un navegador o herramienta de autoedición. `sgmllib.py` presenta HTML de forma *estructural*.

`sgmllib.py` contiene una clase importante: `SGMLParser`. `SGMLParser` disgrega HTML en partes útiles, como etiquetas de inicio y fin. Tan pronto como consigue obtener algunos datos, llama a un método de sí misma basándose en lo que encontró. Para utilizar este analizador, derivará la clase `SGMLParser` para luego reemplazar estos métodos. A esto me refería cuando dije que presenta HTML de forma *estructural*: la estructura de HTML determina la secuencia de llamadas a métodos y los argumentos que se pasarán a cada uno.

`SGMLParser` disgrega HTML en 8 tipos de dato, e invoca métodos diferentes para cada uno de ellos:

Etiqueta de inicio
: Una etiqueta HTML que inicia un bloque, como `<html>`, `<head>`, `<body>`, o `<pre>`, o una etiqueta individual como `<br>` o `<img>`. Cuando encuentra una etiqueta de inicio buscará un método llamado `start_`*`nombre_etiqueta`* o `do_`*`nombre_etiqueta`*. Por ejemplo, cuando encuentra una etiqueta `<pre>`, busca un método llamado `start_pre` o `do_pre`. Si encuentra el método, `SGMLParser` lo invoca pasando una lista de los atributos de la etiqueta; en caso contrario, llama a `unknown_starttag` pasándole el nombre de la etiqueta y una lista de sus atributos.

Etiqueta de fin
: Una etiqueta de HTML que termina un bloque, como `</html>`, `</head>`, `</body>`, o `</pre>`. Cuando encuentra una etiqueta de fin, `SGMLParser` busca un método llamado `end_`*`tagname`*. Si lo encuentra, `SGMLParser` lo invoca, y si no llama a `unknown_endtag` pasándole el nombre de la etiqueta.

Referencia a carácter
: Un carácter escapado referenciado por su equivalente decimal o hexadecimal, como ` `. Cuando se le encuentra, `SGMLParser` invoca a `handle_charref` pasándole el texto del equivalente decimal o hexadecimal del carácter.

Referencia a entidad

Una entidad HTML, como `&copy;`. Cuando `SGMLParser` las encuentra, invoca a `handle_entityref` pasándole el nombre de la entidad HTML.

Comentario

Un comentario limitado por `<!-- ... -->`. Cuando `SGMLParser` lo encuentra, invoca a `handle_comment` pasándole el cuerpo del comenario.

Instrucción de proceso

Una instrucción de procesamiento[4] limitada por `<? ... >`. Cuando `SGMLParser` la encuentra, invoca a `handle_pi` pasándole el cuerpo de la instrucción.

Declaración

Una declaración de HTML, como `DOCTYPE`, limitada por `<! ... >`. Cuando `SGMLParser` la encuentra, invoca a `handle_decl` pasándole el cuerpo de la declaración.

Datos textuales

Un bloque de texto. Cualquier cosa que no caiga en ninguna de las otras 7 categorías. Cuando `SGMLParser` encuentra esto, invoca a `handle_data` pasándole el texto.

Python 2.0 sufría un fallo debido al que `SGMLParser` no podía reconocer declaraciones (no se llamaba nunca a `handle_decl`), lo que quiere decir que se ignoraban los `DOCTYPE` sin advertirlo. Esto quedó corregido en Python 2.1.

`sgmllib.py` incluye una batería de pruebas para ilustrarlo. Puede ejecutar `sgmllib.py` pasando el nombre de un documento HTML en la línea de órdenes y esto imprimirá las etiquetas y otros elementos a medida que los reconozca. Esto lo hace derivando la clase `SGMLParser` y definiendo `unknown_starttag`, `unknown_endtag`, `handle_data` y otros métodos para que se limiten a imprimir sus argumentos.

En el IDE ActivePython para Windows puede especificar argumentos en la

línea de órdenes desde el cuadro de diálogo “Run script”. Si incluye varios argumentos sepárelos con espacios.

Ejemplo 8.4. Prueba de ejemplo de `sgmllib.py`

Aquí hay un fragmento de la tabla de contenidos de la versión HTML de este libro. Por supuesto las rutas de usted pueden variar (si no ha descargado la versión HTML del libro, puede verla en http://diveintopython.org/).

```
c:\python23\lib> type
"c:\downloads\diveintopython\html\toc\index.html"

<!DOCTYPE html
  PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html lang="en">
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=ISO-
8859-1">

      <title>Dive Into Python</title>
      <link rel="stylesheet" href="diveintopython.css"
type="text/css">

... se omite el resto del fichero para abreviar ...
```

Hacer pasar esto por la batería de pruebas de `sgmllib.py` resulta en esta salida:

```
c:\python23\lib> python sgmllib.py
"c:\downloads\diveintopython\html\toc\index.html"
data: '\n\n'
start tag: <html lang="en" >
data: '\n   '
start tag: <head>
data: '\n      '
start tag: <meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1" >
data: '\n   \n      '
start tag: <title>
data: 'Dive Into Python'
end tag: </title>
```

```
data: '\n      '
start tag: <link rel="stylesheet" href="diveintopython.css"
type="text/css" >
data: '\n      '

... se omite el resto de la salida para abreviar ...
```

Aquí tiene la planificación del resto del capítulo:

- Derivar `SGMLParser` para crear clases que extraigan datos interesantes de documentos HTML.
- Derivar `SGMLParser` para crear `BaseHTMLProcessor`, que sustituye los 8 métodos de manipulación y los utiliza para reconstruir el HTML original partiendo de las partes.
- Derivar `BaseHTMLProcessor` para crear `Dialectizer`, que añade algunos métodos para procesar de forma especial algunas etiquetas específicas de HTML, y sustituye el método `handle_data` para proporcionar un marco de procesamiento de los bloques de texto que hay entre las etiquetas HTML.
- Derivar `Dialectizer` para crear clases que definan las reglas de procesamiento de texto que usa `Dialectizer.handle_data`.
- Escribir una batería de pruebas que tomen una página web real de `http://diveintopython.org/` y la procesen.

Por el camino aprenderá también que existen `locals`, `globals` y cómo dar formato a cadenas usando un diccionario.

Footnotes

[4] *processing instruction (PI)*

8.3. Extracción de datos de documentos HTML

Para extraer datos de documentos HTML derivaremos la clase `SGMLParser` y definiremos métodos para cada etiqueta o entidad que queramos capturar.

El primer paso para extrar datos de un documento HTML es obtener un HTML. Si tiene algún documento en su disco duro, use las funciones de ficheros para leerlo, pero lo divertido empezará cuando lo usemos sobre HTML sacado directamente de páginas web.

Ejemplo 8.5. Presentación de `urllib`

```
>>> import urllib                                              ❶
>>> sock = urllib.urlopen("http://diveintopython.org/")        ❷
>>> htmlSource = sock.read()                                   ❸
>>> sock.close()                                               ❹
>>> print htmlSource                                           ❺
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd"><html><head>
      <meta http-equiv='Content-Type' content='text/html; charset=ISO-
8859-1'>
   <title>Dive Into Python</title>
<link rel='stylesheet' href='diveintopython.css' type='text/css'>
<link rev='made' href='mailto:mark@diveintopython.org'>
<meta name='keywords' content='Python, Dive Into Python, tutorial,
object-oriented, programming, documentation, book, free'>
<meta name='description' content='a free Python tutorial for
experienced programmers'>
</head>
<body bgcolor='white' text='black' link='#0000FF' vlink='#840084'
alink='#0000FF'>
<table cellpadding='0' cellspacing='0' border='0' width='100%'>
<tr><td class='header' width='1%' valign='top'>diveintopython.org</td>
<td width='99%' align='right'><hr size='1' noshade></td></tr>
<tr><td class='tagline'
colspan='2'>Python for experienced programmers</td></tr
>

[...cortamos...]
```

❶ El módulo `urllib` es parte de la biblioteca estándar de Python. Contiene funciones para obtener información sobre URLs (páginas web, principalmente) y para descargar datos desde ellas.

❷ El uso más simple de `urllib` es la descarga del texto completo de una página

web usando la función `urlopen`. Abrir una URL es similar a abrir un fichero. El valor de retorno de `urlopen` es objeto parecido al de un fichero, que tiene algunos de sus mismos métodos.

❸ La operación más sencilla que puede realizar con el objeto devuelto por `urlopen` es `read`, que lee el HTML completo de la página web y lo almacena en una cadena de texto. El objeto también admite `readlines`, que devuelve el texto línea por línea en una lista.

❹ Cuando haya terminado con el objeto, asegúrese de cerrarlo (`close`), igual que con un objeto de fichero normal.

❺ Ahora tenemos el HTML completo de la página principal de `http://diveintopython.org/` en una cadena, preparado para su análisis.

Ejemplo 8.6. Presentación de `urllister.py`

Si aún no lo ha hecho, puede descargar éste ejemplo y otros usados en este libro.

```
from sgmllib import SGMLParser

class URLLister(SGMLParser):
    def reset(self):                              ❶
        SGMLParser.reset(self)
        self.urls = []

    def start_a(self, attrs):                     ❷
        href = [v for k, v in attrs if k=='href'] ❸ ❹
        if href:
            self.urls.extend(href)
```

❶ `reset` lo invoca el método `__init__` de `SGMLParser` y también se le puede llamar de forma manual una vez se ha creado la instancia del analizador. Si necesita hacer cualquier tipo de inicialización póngalo en `reset`, no en `__init__`, de manera que se reinicialice adecuadamente cuando alguien reutilice una instancia del analizador.

❷ `start_a` lo invoca `SGMLParser` cada vez que encuentra una etiqueta `<a>`. La etiqueta puede contener un atributo `href` y otros, como `name` o `title`. El parámetro `attrs` es una lista de tuplas `[(atributo, valor), (atributo, valor), ...]`. O puede que sea sólo una `<a>`, que es una etiqueta HTML válida (aunque inútil), en cuyo caso `attrs` será una lista vacía.

❸ Puede averiguar si esta etiqueta `<a>` tiene un atributo `href` con una simple lista por comprensión multivariable.

❹ Las comparaciones de cadenas como `k=='href'` siempre distiguen las mayúsculas, pero en este caso no hay peligro porque `SGMLParser` convierte los nombres de los atributos a minúsculas cuando construye `attrs`.

Ejemplo 8.7. Uso de `urllister.py`

```
>>> import urllib, urllister
>>> usock = urllib.urlopen("http://diveintopython.org/")
>>> parser = urllister.URLLister()
>>> parser.feed(usock.read())          ❶
>>> usock.close()                      ❷
>>> parser.close()                     ❸
>>> for url in parser.urls: print url  ❹
toc/index.html
#download
#languages
toc/index.html
appendix/history.html
download/diveintopython-html-5.0.zip
download/diveintopython-pdf-5.0.zip
download/diveintopython-word-5.0.zip
download/diveintopython-text-5.0.zip
download/diveintopython-html-flat-5.0.zip
download/diveintopython-xml-5.0.zip
download/diveintopython-common-5.0.zip


... se omite el resto de la salida para abreviar ...
```

❶ Invoque el método `feed`, definido en `SGMLParser`, para alimentar el HTML al analizador.[5] Acepta una cadena, que es lo que devuelve `usock.read()`.

❷ Igual que con los objetos, debería cerrar (`close`) sus objetos URL tan pronto como acabe de usarlos.

❸ Debería cerrar (`close`) también el objeto analizador, pero por una razón diferente. Ha leído todos los datos y se los ha dado al analizador, pero el método `feed` no garantiza que se haya procesado todo el HTML que le pasó; puede haberlo almacenado en un búfer, y estar esperando por más. Asegúrese de invocar `close` para forzar el vaciado del búfer y que se analice todo por completo.

❹ Una vez cerrado el analizado queda completo el análisis, y `parser.urls` contiene una lista de todas las URLs con las que enlaza el documento HTML. (Su salida puede ser diferente, si se han actualizado los enlaces de descarga para cuando usted lea esto).

Footnotes

[5] El término técnico para un analizador como `SGMLParser` es *consumidor*: consume HTML y lo separa en partes. Es presumible que el nombre `feed` se escogiera para ajustarse a la idea de "consumidor". Personalmente, me hace pensar en una atracción en el zoológico donde sólo hay una jaula oscura sin árboles ni plantas ni evidencia de vida de ninguna clase, pero si uno se queda perfectamente quieto y observa desde muy cerca, puede llegar a ver dos ojos ovalados que le devuelven la mirada desde la esquina más alejada, aunque uno se convence de que sólo es su imaginación que le está gastando una broma, y lo única manera de que uno pueda decir que eso no es sólo una jaula vacía es una pequeña e inocua señal en las rejas que reza, "No dé de comer al analizador". Pero quizá es sólo cosa mía. De cualquier manera, es una imagen mental interesante.

8.4. Presentación de `BaseHTMLProcessor.py`

`SGMLParser` no produce nada por sí mismo. Analiza, analiza y analiza, e invoca métodos por cada cosa interesante que encuentra, pero los métodos no hacen

nada. `SGMLParser` es un *consumidor* de HTML: toma un HTML y lo divide en trozos pequeños y estructurados. Como vio en la sección anterior, puede derivar `SGMLParser` para definir clases que capturen etiquetas específicas y produzcan cosas útiles, como una lista de todos los enlaces en una página web. Ahora llevará esto un paso más allá, definiendo una clase que capture todo lo que `SGMLParser` le lance, reconstruyendo el documento HTML por completo. En términos técnicos, esta clase será un *productor* de HTML.

`BaseHTMLProcessor` deriva de `SGMLParser` y proporciona los 8 métodos de manipulación esenciales: `unknown_starttag`, `unknown_endtag`, `handle_charref`, `handle_entityref`, `handle_comment`, `handle_pi`, `handle_decl` y `handle_data`.

Ejemplo 8.8. Presentación de `BaseHTMLProcessor`

```
class BaseHTMLProcessor(SGMLParser):
    def reset(self):                                  ❶
        self.pieces = []
        SGMLParser.reset(self)

    def unknown_starttag(self, tag, attrs):          ❷
        strattrs = "".join([' %s="%s"' % (key, value) for key, value
in attrs])
        self.pieces.append("<%(tag)s%(strattrs)s>" % locals())

    def unknown_endtag(self, tag):                    ❸
        self.pieces.append("</%(tag)s>" % locals())

    def handle_charref(self, ref):                    ❹
        self.pieces.append("&#%(ref)s;" % locals())

    def handle_entityref(self, ref):                  ❺
        self.pieces.append("&%(ref)s" % locals())
        if htmlentitydefs.entitydefs.has_key(ref):
            self.pieces.append(";")

    def handle_data(self, text):                      ❻
        self.pieces.append(text)
```

```
    def handle_comment(self, text):          ❼
        self.pieces.append("<!--%(text)s-->" % locals())

    def handle_pi(self, text):               ❽
        self.pieces.append("<?%(text)s>" % locals())

    def handle_decl(self, text):
        self.pieces.append("<!%(text)s>" % locals())
```

❶ `reset`, invocado por `SGMLParser.__init__`, inicializa `self.pieces` como una lista vacía antes de llamar al método del ancestro. `self.pieces` es un atributo de datos que contendrá las partes del documento HTML que usted está construyendo. Cada método manejador reconstruirá el HTML analizado por `SGMLParser` añadiendo esa cadena a `self.pieces`. Observe que `self.pieces` es una lista. Podría tentarle definirla como una cadena y limitarse a concatenar cada parte. Esto funcionaría, pero Python es mucho más eficiente tratando listas.[6]

❷ Dado que `BaseHTMLProcessor` no define métodos para etiquetas específicas (como el método `start_a` de `URLLister`), `SGMLParser` invocará a `unknown_starttag` por cada etiqueta de inicio. Este método toma la etiqueta (`tag`) y la lista de pares nombre/valor de atributos (`attrs`), reconstruye el HTML original y lo añade a `self.pieces`. La cadena de formato que vemos es un poco extraña; la desentrañará (y también esa función `locals` de pinta tan extraña) más adelante en este capítulo.

❸ Reconstruir las etiquetas de final es mucho más simple; basta tomar el nombre de la etiqueta y encerrarlo entre `</...>`.

❹ Cuando `SGMLParser` encuentra una referencia a un carácter, llama a `handle_charref` pasándole la referencia sin delimitadores. Si el documento HTML contiene la referencia ` `, `ref` será `160`. Reconstruir la referencia original completa implica sólo encapsular `ref` con los caracteres `&#...;`.

❺ Las referencias a entidades son similares a las referencias a caracteres, pero sin la marca `#`. Reconstruir la referencia original implica encapsular `ref` dentro de `&...;`. (En realidad, como un erudito lector me ha señalado, es ligeramente más complicado que esto. Sólo ciertas entidades estándar de

HTML terminan en punto y coma; el resto de entidades similares no lo hacen. Por suerte para nosotros, el conjunto de entidades estándar de HTML está definido en un diccionario del módulo de Python llamado `htmlentitydefs`. De ahí la sentencia `if` adicional)

6. Los bloques de texto se añaden a `self.pieces` sin alterar, simplemente.
7. Los comentarios de HTML van escritos entre los caracteres `<!--...-->`.
8. Las instrucciones de procesamiento van escritas entre los caracteres `<?...>` .

La especificación de HTML precisa que todo lo que no sea HTML (como JavaScript para el cliente) debe estar encerrado dentro de comentarios de HTML, pero no todas las páginas web lo hacen correctamente (y todos los navegadores modernos hacen la vista gorda en ese caso). `BaseHTMLProcessor` no es tan permisivo; si un *script* no está adecuadamente embebido, será analizado como si fuera HTML. Por ejemplo, si el script contiene símbolos "igual" o "menor que", `SGMLParser` puede entender de incorrectamente que ha encontrado etiquetas y atributos. `SGMLParser` siempre convierte los nombres de etiquetas y atributos a minúsculas, lo que puede inutilizar el *script*, y `BaseHTMLProcessor` siempre encierra los valores de atributos dentro de comillas dobles (incluso si el documento original utiliza comillas simples o ningún tipo de comillas), lo que hará inútil el *script* con certeza. Proteja siempre sus *script* dentro de comentarios de HTML.

Ejemplo 8.9. Salida de `BaseHTMLProcessor`

```
def output(self):                    ❶
    """Return processed HTML as a single string"""
    return "".join(self.pieces)      ❷
```

❶ Éste es el método de `BaseHTMLProcessor` que el ancestro `SGMLParser` no invoca nunca. Dado que los otros métodos manejadores almacenan su HTML reconstruido en `self.pieces` se necesita esta función para juntar todas las partes en una sola cadena. Como indicamos antes, Python es muy bueno con

las listas y mediocre con las cadenas, así que sólo crearemos la cadena completa cuando alguien nos la solicite explícitamente.

❷ Si lo prefiere puede utilizar el método `join` del módulo `string` en lugar de: `string.join(self.pieces, "")`

Lecturas complementarias

- W3C expone las referencias a caracteres y entidades.
- La *Referencia de bibliotecas de Python* confirmará sus sospechas al respecto de que el módulo `htmlentitydefs` es exactamente lo que parece.

Footnotes

[6] La razón de que Python sea mejor con las listas que con las cadenas es que las listas son mutables pero las cadenas son inmutables. Esto significa que añadir a una lista simplemente agrega el elemento y actualiza el índice. Como las cadenas no se pueden modificar tras haber sido creadas, un código como `s = s + nuevaparte` creará una cadena complemente nueva partiendo de la concatenación de la original y de la nueva parte, para asignarla luego a la variable de la cadena original. Esto implica mucha gestión de memoria costosa, y la cantidad de esfuerzo empleado aumenta cuando la cadena va creciendo, de manera que hacer `s = s + nuevaparte` en un bucle es mortal. En términos técnicos, agregar `n` elementos a una lista es `O(n)`, mientras que añadir `n` elementos a una cadena es $O(n^2)$.

8.5. `locals` y `globals`

Hagamos por un minuto un inciso entre tanto procesamiento de HTML y hablemos sobre la manera en que Python gestiona las variables. Python incorpora dos funciones, `locals` y `globals`, que proporcionan acceso de tipo diccionario a las variables locales y globales.

¿Se acuerda de `locals`? La vio por primera vez aquí:

```
    def unknown_starttag(self, tag, attrs):
        strattrs = "".join([' %s="%s"' % (key, value) for key, value
in attrs])
        self.pieces.append("<%(tag)s%(strattrs)s>" % locals())
```

No, espere, todavía no puede aprender cosas sobre `locals`. Antes debe aprender sobre espacios de nombres. Es un material árido pero es importante, así que preste atención.

Python utiliza lo que llamamos espacios de nombres[7] para llevar un seguimiento de las variables. Un espacio de nombres es como un diccionario donde las claves son los nombres de las variables y los valores del diccionario son los de esas variables. En realidad, puede acceder a un espacio de nombres de Python como a un diccionario, como veremos en breve.

En cualquier momento en particular en un programa de Python se puede acceder a varios espacios de nombres. Cada función tiene su propio espacio de nombres, que llamamos espacio local, que contiene las variables que define la función, incluyendo sus argumentos y las variables definidas de forma local. Cada módulo tiene su propio espacio de nombres, denominado espacio global, que contiene las variables del módulo, incluyendo sus funciones, clases y otros módulos que haya importado, así como variables y constantes del módulo. Y hay un espacio de nombres incorporado, accesible desde cualquier módulo, que contiene funciones del lenguaje y excepciones.

Cuando una línea de código solicita el valor de una variable `x`, Python busca esa variable en todos los espacios de nombres disponibles, por orden:

1. espacio local, específico a la función o método de clase actual. Si la función define una variable local `x`, o tiene un argumento `x`, Python usará ésta y dejará de buscar.
2. espacio global, específico al módulo actual. Si el módulo ha definido una variable, función o clase llamada `x`, Python ésta y dejará de buscar.

3. espacio incorporado, global a todos los módulos. Como último recurso, Python asumirá que `x` es el nombre de una función o variable incorporada por el lenguaje.

Si Python no encuentra `x` en ninguno de estos espacios, se rendirá y lanzará una `NameError` con el mensaje `There is no variable named 'x'`, que ya vio en su momento en Ejemplo 3.18, "Referencia a una variable sin asignar", pero no podía aún apreciar todo el trabajo que Python estaba realizando antes de mostrarle ese error.

Python 2.2 introdujo un cambio sutil pero importante que afecta al orden de búsqueda en los espacios de nombre: los ámbitos anidados. En las versiones de Python anteriores a la 2.2, cuando hace referencia a una variable dentro de una función anidda o función `lambda`, Python buscará esa variable en el espacio de la función actual (anidada o `lambda`) y después en el espacio de nombres del módulo. Python 2.2 buscará la variable en el espacio de nombres de la función actual (anidada o `lambda`), *después en el espacio de su función madre*, y luego en el espacio del módulo. Python 2.1 puede funcionar de ambas maneras; por omisión lo hace como Python 2.0, pero puede añadir la siguiente línea al código al principio de su módulo para hacer que éste funcione como Python 2.2:

```
from __future__ import nested_scopes
```

¿Ya está confundido? ¡No desespere! Esto es realmente bueno, lo prometo. Como muchas cosas en Python, a los espacios de nombre *se puede acceder directamente durante la ejecución*. ¿Cómo? Bien, al espacio local de nombres se puede acceder mediante la función incorporada `locals`, y el espacio global (del módulo) es accesible mediante la función `globals`.

Ejemplo 8.10. Presentación de `locals`

```
>>> def foo(arg): ❶
```

```
...     x = 1
...     print locals()
...
>>> foo(7)        ❷
{'arg': 7, 'x': 1}
>>> foo('bar')    ❸
{'arg': 'bar', 'x': 1}
```

❶ La función `foo` tiene dos variables en su espacio de nombres local: `arg`, cuyo valor se pasa a la función y `x`, que se define dentro de la función.

❷ `locals` devuelve un diccionario de pares nombre/valor. Las claves de este diccionario son los nombres de las variables como cadenas; los valores del diccionario son los auténticos valores de las variables. De manera que invocar a `foo` con un `7` imprime el diccionario que contiene las dos variables locales de la función: `arg` (`7`) y `x` (`1`).

❸ Recuerde, Python es de tipado dinámico, así que podría pasar una cadena en `arg` con la misma facilidad; la función (y la llamada a `locals`) funcionará igual de bien. `locals` funciona con todas las variables de todos los tipos.

Lo que hace `locals` con los espacios de nombres locales (función), lo hace `globals` para el espacio de nombres global (módulo). `globals` es más interesante, sin embargo, porque el espacio un módulo es más interesante.[8] El espacio de nombres del módulo no sólo incluye variables y constantes del módulo, también incluye todas las funciones y clases definidas en él. Además, incluye cualquier cosa que se haya importado dentro del módulo.

¿Recuerda la diferencia entre `from módulo import` e `import módulo`? Con `import módulo` se importa el módulo en sí, pero retiene su espacio de nombres, y esta es la razón por la que necesita usar el nombre del módulo para acceder a cualquiera de sus funciones o atributos: *`módulo.función`*. Pero con `from módulo import`, en realidad está importando funciones y atributos específicos de otro módulo al espacio de nombres del suyo propio, que es la razón por la que puede acceder a ellos directamente sin hacer referencia al módulo del que vinieron originalmente. Con la función `globals` puede ver esto en funcionamiento.

Ejemplo 8.11. Presentación de `globals`

Mire el siguiente bloque de código al final de `BaseHTMLProcessor.py`:

```
if __name__ == "__main__":
    for k, v in globals().items():              ❶
        print k, "=", v
```

❶ Para que no se sienta intimidado, recuerde que ya ha visto todo esto antes. La función `globals` devuelve un diccionario y estamos iterando sobre él usando el método `items` y una asignación multivariable. La única cosa nueva es la función `globals`.

Ahora ejecutar el *script* desde la línea de órdenes nos da esta salida (tenga en cuenta que la suya puede ser ligeramente diferente, dependiendo de la plataforma en que haya instalado Python):

```
c:\docbook\dip\py> python BaseHTMLProcessor.py

SGMLParser = sgmllib.SGMLParser                 ❶

htmlentitydefs = <module 'htmlentitydefs' from
'C:\Python23\lib\htmlentitydefs.py'> ❷

BaseHTMLProcessor = __main__.BaseHTMLProcessor ❸

__name__ = __main__                             ❹

... se omite el resto de la salida por abreviar ...
```

❶ `SGMLParser` se importó desde `sgmllib`, usando `from` *`módulo`* `import`. Esto significa que lo importamos directamente en el espacio de nombres del módulo, y aquí está.

❷ En contraste con esto tenemos `htmlentitydefs`, que importamos usando `import`. Esto significa que es el módulo `htmlentitydefs` quien se encuentra en el espacio de nombres, pero la variable `entitydefs` definida en su interior, no.

❸ Este módulo sólo define una clase, `BaseHTMLProcessor`, y aquí la tenemos. Observe que el valor aquí es la propia clase, no una instancia específica de la clase.

❹ ¿Recuerda el truco `if __name__`? Cuando ejecuta un módulo (en lugar de importarlo desde otro), el atributo `__name__` que incorpora contiene un valor especial, `__main__`. Dado que ejecutamos el módulo desde la línea de órdenes, `__name__` es `__main__`, y por eso se ejecuta el pequeño código de prueba que imprime `globals`.

Puede obtener dinámicamente el valor de variables arbitrarias usando las funciones `locals` y `globals`, proporcionando el nombre de la variable en una cadena. Esto imita la funcionalidad de la función `getattr`, que le permite acceder a funciones arbitrarias de forma dinámica proporcionando el nombre de la función en una cadena.

Hay otra diferencia importante entre las funciones `locals` y `globals` que deberá aprender antes de que se pille los dedos con ella. Y se los pillará de todas maneras, pero al menos recordará haberlo aprendido.

Ejemplo 8.12. `locals` es de sólo lectura, `globals` no

```
def foo(arg):
    x = 1
    print locals()      ❶
    locals()["x"] = 2   ❷
    print "x=",x        ❸

z = 7
print "z=",z
foo(3)
globals()["z"] = 8      ❹
print "z=",z            ❺
```

❶ Dado que llamamos a `foo` con `3`, esto imprimirá `{'arg': 3, 'x': 1}`. No debería sorprenderle.

❷ `locals` es una función que devuelve un diccionario, y aquí estamos asignando un valor a ese diccionario. Puede que piense que esto cambiará el valor de la variable local `x` a `2`, pero no es así. `locals` no devuelve el verdadero espacio de nombres local, sino una copia. Así que cambiarlo no hace nada con los valores de las variables del espacio local.

❸ Esto imprime `x= 1`, no `x= 2`.

❹ Tras habernos quemado con `locals` podríamos pensar que esto *no* cambiaría el valor de `z`, pero sí lo hace. Debido a las diferencias internas en la manera que está implementado Python (en las que no voy a entrar, ya que no las comprendo completamente), `globals` devuelve el verdadero espacio de nombres global, no una copia: justo lo opuesto que `locals`. Así que los cambios que haga en el diccionario devuelto por `globals` afectan directamente a las variables globales.

❺ Esto imprime `z= 8`, no `z= 7`.

Footnotes

[7] *namespaces*

[8] La verdad es que no salgo mucho a la calle.

8.6. Cadenas de formato basadas en diccionarios

¿Por qué le he enseñado que existen `locals` y `globals`? Para que pueda aprender a dar formato a cadenas usando diccionarios. Como recordará, la cadena de formato normal proporciona una manera sencilla de insertar valores en cadenas. Los valores se listan en una tupla y se insertan por orden en la cadena en lugar de cada marcador de formato. Aunque esto es eficiente, no siempre produce el código más fácil de leer, especialmente cuando ha de insertar múltiples valores. No se puede simplemente echar un vistazo a la

cadena y comprender cual será el resultado; alternará constantemente entre leer la cadena y la tupla de valores.

Hay una forma alternativa de dar formato a cadenas que usa diccionarios en lugar de tuplas de valores.

Ejemplo 8.13. Presentación de la cadena de formato basada en diccionarios

```
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa",
"pwd":"secret"}
>>> "%(pwd)s" % params                                          ❶
'secret'
>>> "%(pwd)s is not a good password for %(uid)s" % params       ❷
'secret is not a good password for sa'
>>> "%(database)s of mind, %(database)s of body" % params       ❸
'master of mind, master of body'
```

❶ En lugar de una tupla de valores explícitos, esta forma de cadena de formato utiliza un diccionario, `params`. Y en lugar de un ser un simple `%s` en la cadena, el marcador contendrá un nombre entre paréntesis. Este nombre se utilizar como clave para acceder al diccionario `params` y se inserta el valor correspondiente, `secret`, en lugar del marcador `%(pwd)s`.

❷ La cadena de formato con diccionarios funciona con cualquier cantidad de claves por nombre. Cada clave debe existir en el diccionario dado, o el formato fallará con una `KeyError`.

❸ Incluso puede especificar la misma clave dos veces; cada una será sustituida por el mismo valor.

Así que, ¿por qué usar una cadena de formato con diccionario? Bien, parece matar moscas a cañonazos construir un diccionario de claves y valores sólo para dar formato a una cadena en la siguiente línea; es mucho más útil cuando resulta que ya tiene un diccionario lleno de claves significativas con sus respectivos valores. Como `locals`.

Ejemplo 8.14. Formato basado en diccionarios en `BaseHTMLProcessor.py`

```
    def handle_comment(self, text):
        self.pieces.append("<!--%(text)s-->" % locals()) ❶
```

❶ El uso más común de la cadena de formato con diccionarios es acompañarla de la función incorporada `locals`. Significa que puede usar los nombres de las variables locales dentro de la cadena (en este caso, `text`, que se le pasó al método de la clase como argumento) y cada variable nombrada será sustituida por su valor. Si `text` es `'Begin page footer'`, el formato `"<!--%(text)s-->" % locals()` producirá la cadena `'<!--Begin page footer-->'`.

Ejemplo 8.15. Más cadenas de formato basadas en diccionarios

```
    def unknown_starttag(self, tag, attrs):
        strattrs = "".join([' %s="%s"' % (key, value) for key, value
in attrs]) ❶
        self.pieces.append("<%(tag)s%(strattrs)s>" % locals())
❷
```

❶ Cuando se invoca este método, `attrs` es una lista de tuplas clave/valor igual que los `items` de un diccionario, lo que significa que puede usar la asignación multivariable para iterar sobre ella. Este patrón debería resultarle familiar ya, pero se está haciendo mucho aquí, así que vamos a hacer una disección:

a. Suponga que `attrs` es `[('href', 'index.html'), ('title', 'Go to home page')]`.
b. En la primera iteración de la lista por comprensión, `key` tendrá `'href'` y `value` tendrá `'index.html'`.
c. La cadena de formato `' %s="%s"' % (key, value)` producirá `' href="index.html"'`. Esta cadena se convierte en el primer elemento del valor devuelto por la lista por comprensión.
d. En la segunda iteración, `key` tendrá `'title'`, y `value` tendrá `'Go to home page'`.

e. La cadena de formato producirá `' title="Go to home page"'`.
f. La lista por comprensión devuelve una lista con estas dos cadenas producto, y `strattrs` juntará ambos elementos de esta lista para formar `' href="index.html" title="Go to home page"'`.

2. Ahora insertará el valor de `tag` y `strattrs` en una cadena usando la cadena de formato basada en diccionarios. De manera que si `tag` es `'a'`, el resultado final será `'<a href="index.html" title="Go to home page">'`, y eso es lo que se agrega a `self.pieces`.

Usar cadenas de formato basadas en diccionarios con `locals` es una manera conveniente de hacer más legibles expresiones de cadenas de formato, pero tiene un precio. Llamar a locals tiene una ligera influencia en el rendimiento, ya que `locals` construye una copia del espacio de nombres local.

8.7. Poner comillas a los valores de los atributos

Una pregunta habitual en comp.lang.python es "Tengo unos documentos HTML con valores de atributos sin comillas, y quiero corregirlos todos. ¿Cómo puedo hacerlo?"[9] (Generalmente sucede cuando un jefe de proyecto que ha abrazado la religión HTML-es-un-estándar se une a un proyecto y proclama que todas las páginas deben pasar el validador de HTML. Los valores de atributos sin comillas son una violación habitual del estándar HTML). Cualquiera que sea la razón, es fácil corregir el problema de los valores de atributo sin comillas alimentando a `BaseHTMLProcessor` con HTML.

`BaseHTMLProcessor` consume HTML (ya que desciende de `SGMLParser`) y produce HTML equivalente, pero la salida no es idéntica a la entrada. Las etiquetas y los nombres de atributos terminarán en minúscula, incluso si al principio estaban en mayúscula o todo mezclado, y los valores de los atributos

estarán entre comillas dobles, incluso si estaban entre comillas simples, o sin comillas. Es éste último efecto secundario el que podemos aprovechar.

Ejemplo 8.16. Poner comillas a valores de atributo

```
>>> htmlSource = """                    ❶
...     <html>
...     <head>
...     <title>Test page</title>
...     </head>
...     <body>
...     <ul>
...     <li><a href=index.html>Home</a></li>
...     <li><a href=toc.html>Table of contents</a></li>
...     <li><a href=history.html>Revision history</a></li>
...     </body>
...     </html>
...     """
>>> from BaseHTMLProcessor import BaseHTMLProcessor
>>> parser = BaseHTMLProcessor()
>>> parser.feed(htmlSource)             ❷
>>> print parser.output()               ❸
<html>
<head>
<title>Test page</title>
</head>
<body>
<ul>
<li><a href="index.html">Home</a></li>
<li><a href="toc.html">Table of contents</a></li>
<li><a href="history.html">Revision history</a></li>
</body>
</html>
```

❶ Observe que los valores de los atributos `href` de las etiquetas `<a>` no tienen las comillas adecuadas. (Advierta también que está usando comillas triples para algo diferente a una `cadena de documentación`. Y directamente en el IDE, nada menos. Son muy útiles).

❷ Alimentar al analizador.

❸ Usando la función `output` definida en `BaseHTMLProcessor` obtenemos la salida como una única cadena, completa con comillas en los valores sus atributos. Aunque esto pueda parecer anticlimático, piense sobre todo lo que ha sucedido aquí: `SGMLParser` analizó el documento HTML entero, reduciéndolo a etiquetas, referencias, datos, *etc.*; `BaseHTMLProcessor` usó esos elementos para reconstruir las partes de HTML (que aún están almacenadas en `parser.pieces`, por si quiere verlas); y por último, ha llamado a `parser.output`, que juntó todas las partes del HTML en una única cadena.

Footnotes

[9] Vale, no es una pregunta tan frecuente. No se acerca a "¿Qué editor debería usar para escribir código de Python?" (respuesta: Emacs - el traductor disiente) o "¿Python es mejor o peor que Perl?" (respuesta: "Perl es peor que Python porque la gente quiere que sea peor." -Larry Wall, 10/14/1998) Pero aparecen preguntas sobre procesamiento de HTML de una u otra forma una vez al mes, y entre estas preguntas, esta es bastante popular.

8.8. Presentación de `dialect.py`

`Dialectizer` es una descendiente sencilla (y tonta) de `BaseHTMLProcessor`. Hace una serie de sustituciones sobre bloques de texto, pero se asegura de que cualquier cosa dentro de un bloque `<pre>...</pre>` pase sin alteraciones.

Para manejar los bloques `<pre>` definimos dos métodos en `Dialectizer`: `start_pre` y `end_pre`.

Ejemplo 8.17. Manipulación de etiquetas específicas

```
    def start_pre(self, attrs):                 ❶
        self.verbatim += 1                      ❷
        self.unknown_starttag("pre", attrs)     ❸

    def end_pre(self):                          ❹
        self.unknown_endtag("pre")              ❺
```

```
        self.verbatim -= 1                  ❻
```

❶ Se invoca `start_pre` cada vez que `SGMLParser` encuentra una etiqueta `<pre>` en la fuente HTML (en breve, veremos exactamente cómo sucede esto). El método toma un único parámetro, `attrs` que contiene los atributos de la etiqueta (si los hay). `attrs` es una lista de tuplas clave/valor, igual que la que toma `unknown_starttag`.

❷ En el método `reset`, inicializamos un atributo de datos que sirve como contador para las etiquetas `<pre>`. Cada vez que encontramos una etiqueta `<pre>`, incrementamos el contador; cada vez que encontramos una etiqueta `</pre>`, decrementamos el contador. Podríamos usar esto como indicador y ponerlo a `1` y reiniciarlo a `0` pero es igual de sencillo de esta manera, y además previene el caso extraño (pero posible) de etiquetas `<pre>` anidadas. En un minuto, veremos cómo darle uso a este contador.

❸ Esto es el único procesamiento especial que hacemos para las etiquetas `<pre>`. Ahora pasamos la lista de atributos a `unknown_starttag` de manera que pueda hacer el procesamiento por omisión.

❹ Se invoca `end_pre` cada vezq ue `SGMLParser` encuentra una etiqueta `</pre>`. Dado que las etiquetas de final no pueden contener atributos, el método no admite parámetros.

❺ Primero querrá hacer el procesamiento por omisión, igual que con cualquier otra etiqueta de final.

❻ En segundo luegar, decrementamos el contador para avisar que se ha cerrado este bloque `<pre>`.

Llegados aquí, merece la pena ahondar un poco más en `SGMLParser`. He asegurado repetidamente (y por ahora lo ha tomado a fe) que `SGMLParser` busca e invoca métodos específicos para cada etiqueta, si existen. Por ejemplo, acaba de ver la definición de `start_pre` y `end_pre` para manejar `<pre>` y `</pre>`. ¿Pero cómo sucede esto? Bueno, no es magia, es sólo buena programación en Python.

Ejemplo 8.18. `SGMLParser`

```
def finish_starttag(self, tag, attrs):                    ❶
    try:
        method = getattr(self, 'start_' + tag)            ❷
    except AttributeError:                                ❸
        try:
            method = getattr(self, 'do_' + tag)           ❹
        except AttributeError:
            self.unknown_starttag(tag, attrs)             ❺
            return -1
        else:
            self.handle_starttag(tag, method, attrs)      ❻
            return 0
    else:
        self.stack.append(tag)
        self.handle_starttag(tag, method, attrs)
        return 1                                          ❼

def handle_starttag(self, tag, method, attrs):
    method(attrs)                                         ❽
```

❶ En este lugar, `SGMLParser` ya ha encontrado una etiqueta de inicio y ha obtenido la lista de atributos. La única cosa que le queda por hacer es averiguar si hay algún método manejador especial para esta etiqueta, o si debería acudir en el método por omisión (`unknown_starttag`).

❷ La "magia" de `SGMLParser` no es nada más que su vieja amiga, `getattr`. Lo que puede que no haya advertido hasta ahora es que `getattr` buscará los métodos definidos en los descendientes de un objeto, aparte de en los del propio objeto. Aquí el objeto es `self`, la implementación en sí. De manera que si `tag` es `'pre'`, esta llamada a `getattr` buscará un método `start_pre` en la instancia actual, que es una instancia de la clase `Dialectizer`.

❸ `getattr` lanza una `AttributeError` si el método que busca no existe en el objeto (o en alguno de sus descendientes), pero esto no es malo, ya que encerró la llamada a `getattr` dentro de un bloque `try...except` y se captura de forma explícita la `AttributeError`.

❹ Dado que no se encontró un método `start_xxx`, también buscaremos el método `do_xxx` antes de rendirnos. Este esquema alternativo de nombres se

usa generalmente para etiquetas autocontenidas, como `<br>`, que no tienen etiqueta de cierre correspondiente. Pero puede utilizar cualquiera de los dos sistemas de nombres. Como puede ver, `SGMLParser` prueba ambos por cada etiqueta (sin embargo no debería definir ambos métodos manejadores, `start_xxx` y `do_xxx`; sólo se llamará al método `start_xxx`).

❺ Otra `AttributeError`, lo que significa que la llamada a `getattr` para buscar `do_xxx` falló. Como no hemos encontrado un método `start_xxx` ni `do_xxx` para esta etiqueta, capturamos la excepción y recurrimos al método por omisión, `unknown_starttag`.

❻ Recuerde, los bloques `try...except` pueden tener una cláusula `else`, que se invoca si no se lanza ninguna excepción dentro del bloque `try...except`. Lógicamente, esto significa que *encontramos* un método `do_xxx` para esta etiqueta, así que vamos a invocarla.

❼ Por cierto, no se preocupe por los diferentes valores de retorno; en teoría significan algo, pero nunca se utilizan. No se preocupe tampoco por el `self.stack.append(tag)`; `SGMLParser` lleva un seguimiento interno de si las etiquetas de inicio están equilibradas con las etiquetas de cierre adecuadas, pero no hace nada con esta información. En teoría, podría utilizar este módulo para validar si las etiquetas están equilibradas, pero probablemente no merece la pena, y es algo que se sale del ámbito de este capítulo. Tiene mejores cosas de qué preocuparse ahora mismo.

❽ No se invoca directamente a los métodos `start_xxx` y `do_xxx`; se pasan la etiqueta, método y atributos a esta función, `handle_starttag`, de manera que los descendientes puedan reemplazarla y cambiar la manera en que se despachan *todas* las etiquetas de inicio. No necesita ese nivel de control, así que vamos a limitarnos a usar este método para hacer eso, o sea llamar al método (`start_xxx` o `do_xxx`) con la lista de atributos. Recuerde, `method` es una función devuelta por `getattr` y las funciones son objetos (sé que se está cansando de oír esto, y prometo que dejaré de hacerlo en cuanto se me acaben las ocasiones de usarlo a mi favor). Aquí se pasa el objeto de la función como argumento al método de despacho, y a su vez este método

llama a la función. En este momento, no hace falta saber qué función es, cómo se llama o dónde se define; sólo hace falta saber que la función se invoca con un argumento, `attrs`.

Volvamos ahora a nuestro programa estipulado: `Dialectizer`. Cuando lo dejamos, estábamos en proceso de definir métodos manejadores específicos para las etiquetas `<pre>` y `</pre>`. Sólo queda una cosa por hacer, y es procesar bloques de texto con las sustituciones predefinidas. Para hacer esto, necesita reemplazar el método `handle_data`.

Ejemplo 8.19. Sustitución del método `handle_data`

```
    def handle_data(self, text):
❶
        self.pieces.append(self.verbatim and text or
self.process(text)) ❷
```

❶ Se invoca a `handle_data` con un único argumento, el texto a procesar.

❷ En el ancestro `BaseHTMLProcessor` el método simplemente agregaba el texto al búfer de salida, `self.pieces`. Aquí la lógica es sólo un poco más complicada. Si estamos en mitad de un bloque `<pre>...</pre>`, `self.verbatim` tendrá un valor mayor que `0`, y querremos poner el texto sin alterar en el búfer de salida. En otro caso, querremos invocar a otro método para procesar las sustituciones, y luego poner el resultado de eso en el búfer de salida. En Python esto se hace en una sola línea utilizando el truco `and-or`.

Está cerca de comprender completamente `Dialectizer`. El único eslabón perdido es la naturaleza de las propias sustituciones de texto. Si sabe algo sobre Perl, sabrá que cuando se precisan sustituciones complicadas de texto, la única solución es usar expresiones regulares. Las clases de `dialect.py` definen una serie de expresiones regulares que operan sobre el texto entre las etiquetas HTML. Pero usted acaba de leer un capítulo entero sobre expresiones regulares. No querrá volver a pelearse con las expresiones regulares, ¿verdad? Dios sabe que no. Pienso que ya ha aprendido bastante en este capítulo.

8.9. Todo junto

Es hora de darle buen uso a todo lo que ha aprendido hasta ahora. Espero que haya prestado atención.

Ejemplo 8.20. La función `translate`, parte 1

```
def translate(url, dialectName="chef"): ❶
    import urllib                       ❷
    sock = urllib.urlopen(url)          ❸
    htmlSource = sock.read()
    sock.close()
```

❶ La función `translate` tiene un argumento opcional `dialectName`, que es una cadena que especifica el dialecto que va a usar. Verá cómo se usa esto en un minuto.

❷ ¡Eh!, espere un momento, ¡hay una sentencia `import` en esta función! Esto es perfectamente válido en Python. Está acostumbrado a ver sentencias `import` al comienzo de un programa, lo que significa que lo importado está disponible para todo el programa. Pero también puede importar módulos dentro de una función, lo que quiere decir que el módulo importado sólo está disponible dentro de la función. Si tiene un módulo que sólo se usa una vez en una función, ésta es una manera sencilla de hacer el código más modular (cuando se dé cuenta de que su *hack* del fin de semana se ha convertido en una obra de arte de 800 líneas y decida convertirlo en una docena de módulos reutilizables, agradecerá esto).

❸ Ahora obtenemos la fuente de una URL dada.

Ejemplo 8.21. La función `translate`, parte 2: curiorífico y curiorífico

```
    parserName = "%sDialectizer" % dialectName.capitalize() ❶
    parserClass = globals()[parserName]                     ❷
    parser = parserClass()                                  ❸
```

❶ `capitalize` es un método de cadenas que aún no hemos visto; simplemente pone en mayúscula la primera letra de una cadena y obliga a que el resto esté en minúscula. Combinándola con algo de formato de cadenas, hemos tomado el nombre de un dialecto y lo transformamos en el nombre de la clase Dialectizer correspondiente. Si `dialectName` es la cadena `'chef'`, `parserName` será la cadena `'ChefDialectizer'`.

❷ Tiene el nombre de la clase en una cadena (`parserName`), y tiene el espacio global de nombres como diccionario (`globals()`). Combinándolos, podemos obtener una referencia a la clase que indica la cadena (recuerde, las clases son objetos y se pueden asignar a variables como cualquier otro objeto). Si `parserName` es la cadena `'ChefDialectizer'`, `parserClass` será la clase `ChefDialectizer`.

❸ Por último, tiene el objeto de una clase (`parserClass`), y quiere una instancia de esa clase. Bien, ya sabe cómo hacerlo: invoque a la clase como si fuera una función. El hecho de que la clase esté almacenada en una variable local no supone diferencia alguna; simplemente invoque la variable como si fuera una función, y lo que obtendrá es una instancia de la clase. Si `parserClass` es la clase `ChefDialectizer`, `parser` tendrá una instancia de la clase `ChefDialectizer`.

¿Por qué molestarse? Al fin y al cabo, sólo hay tres clases `Dialectizer`; ¿por qué no usar simplemente una sentencia `case`? (bien, no hay sentencia `case` en Python pero, ¿no es lo mismo usar una serie de sentencias `if`?) Una razón: extensibilidad. La función `translate` no tiene la menor idea de cuántas clases Dialectizer ha definido. Imagine que mañana define una nueva `FooDialectizer`; `translate` funcionará correctamente pasando `'foo'` como `dialectName`.

Incluso mejor, imagine que sitúa `FooDialectizer` en un módulo aparte, y que la importa con `from` *`módulo`* `import`. Ya ha visto que esto la incluye en `globals()`, así que `translate` funcionará aún sin modificaciones, aunque `FooDialectizer` esté en otro fichero.

Ahora imagine que el nombre del dialecto viene de alguna parte de fuera del programa, quizá de una base de datos o de un valor introducido por el usuario en un formulario. Puede usar cualquier cantidad de arquitecturas con Python como lenguaje para *server-side scripting* que generen páginas web de forma dinámica; esta función podría tomar una URL y el nombre de un dialecto (ambos cadenas) en la cadena de consulta de una petición de página web, y mostrar la página web "traducida".

Por último, imagine un *framework* con arquitectura de *plug-in*. Podría colocar cada clase `Dialectizer` en un fichero aparte, dejando sólo la función `translate` en `dialect.py`. Asumiendo un esquema de nombres consistente, la función `translate` podría importar de forma dinámica la clase adecuada del fichero adecuado, sin darle nada excepto el nombre del dialecto (no ha visto el importado dinámico aún, pero prometo que lo veremos en el siguiente capítulo). Para añadir un nuevo dialecto, se limitaría a añadir un fichero con el nombre apropiado en el directorio de plug-ins (como `foodialect.py` qque contendría la clase `FooDialectizer`). Llamar a la función `translate` con el nombre de dialecto `'foo'` encontraría el módulo `foodialect.py`, importaría la clase `FooDialectizer`, y ahí continuaría.

Ejemplo 8.22. La función `translate`, parte 3

```
    parser.feed(htmlSource) ❶
    parser.close()          ❷
    return parser.output()  ❸
```

❶ Tras tanta imaginación, esto va a parecer bastante aburrido, pero la función `feed` es la que hace toda la transformación. Tiene todo el HTML fuente en una única cadena, así que sólo tiene que llamar a `feed` una vez. Sin embargo, podemos llamar a `feed` con la frecuencia que queramos, y el analizador seguirá analizando. Así que si está preocupado por el uso de la memoria (o sabe que va a tratar con un número bien grande de páginas HTML), podría hacer esto dentro de un bucle, en el que leería unos pocos bytes de HTML para dárselo al analizador. El resultado sería el mismo.

❷ Como `feed` mantiene un búfer interno, deberíamos llamar siempre al método `close` del analizador cuando hayamos acabado (incluso si alimentó todo de una vez, como hemos hecho). De otra manera, puede encontrarse conque a la salida le faltan los últimos bytes.

❸ Recuerde, `output` es la función que definió en `BaseHTMLProcessor` que junta todas las piezas de saliad que ha introducido en el búfer y los devuelve en una única cadena.

Y de la misma manera, ha "traducido" una página web, dando sólo una URL y el nombre de un dialdecto.

Lecturas complementarias

- A lo mejor pensaba que estaba de broma con la idea del *server-side scripting*. También lo creía yo hasta que encontré esta web dialectizadora. Por desgracia, no parece estar disponible el código fuente.

8.10. Resumen

Python le proporciona una herramienta potente, `sgmllib.py`, para manipular HTML volviendo su estructura en un modelo de objeto. Puede usar esta herramienta de diferentes maneras.

- análisis de HTML en busca de algo específico
- agregar los resultados, como en el listador de URLs
- alterar la estructura, igual que con el entrecomillador de atributos
- transformar el HTML en algo diferente manipulando el texto sin tocar las etiquetas, como el `Dialectizer`

Según estos ejemplos, debería sentirse cómodo haciendo las siguientes cosas:

- Usar `locals()` y `globals()` para acceder al espacio de nombres
- Dar formato a cadenas usando sustituciones basadas en un diccionario

Capítulo 9. Procesamiento de XML

9.1. Inmersión

Los dos capítulos siguientes hablan sobre el procesamiento de XML en Python. Sería de ayuda que ya conociese el aspecto de un documento XML, que está compuesto por etiquetas estructuradas que forman una jerarquía de elementos. Si esto no tiene sentido para usted, hay muchos tutoriales sobre XML que pueden explicarle lo básico.

Aunque no esté particularmente interesado en XML debería leer estos capítulos, que cubren aspectos importantes como los paquetes de Python, Unicode, los argumentos de la línea de órdenes y cómo usar `getattr` para despachar métodos.

No se precisa tener la carrera de filosofía, aunque si tiene la poca fortuna de haber sido sometido a los escritos de Immanuel Kant, apreciará el programa de ejemplo mucho más que si estudió algo útil, como ciencias de la computación.

Hay dos maneras básicas de trabajar con XML. Una se denomina SAX ("Simple API for XML"), y funciona leyendo un poco de XML cada vez, invocando un método por cada elemento que encuentra (si leyó Capítulo 8, *Procesamiento de HTML*, esto debería serle familiar, porque es la manera en que trabaja el módulo `sgmllib`). La otra se llama DOM ("Document Object Model"), y funciona leyendo el documento XML completo para crear una representación

interna utilizando clases nativas de Python enlazadas en una estructura de árbol. Python tiene módulos estándar para ambos tipos de análisis, pero en este capítulo sólo trataremos el uso de DOM.

Lo que sigue es un programa de Python completo que genera una salida pseudoaleatoria basada en una gramática libre de contexto definida en formato XML. No se preocupe si aún no ha entiende lo que esto significa; examinaremos juntos la entrada y salida del program en más profundidad durante los próximos dos capítulos.

Ejemplo 9.1. `kgp.py`

Si aún no lo ha hecho, puede descargar éste ejemplo y otros usados en este libro.

```
"""Kant Generator for Python

Generates mock philosophy based on a context-free grammar

Usage: python kgp.py [options] [source]

Options:
  -g ..., --grammar=...   use specified grammar file or URL
  -h, --help              show this help
  -d                      show debugging information while parsing

Examples:
  kgp.py                  generates several paragraphs of Kantian
philosophy
  kgp.py -g husserl.xml   generates several paragraphs of Husserl
  kpg.py "<xref id='paragraph'/>"  generates a paragraph of Kant
  kgp.py template.xml     reads from template.xml to decide what to
generate
"""
from xml.dom import minidom
import random
import toolbox
import sys
import getopt
```

```
_debug = 0

class NoSourceError(Exception): pass

class KantGenerator:
    """generates mock philosophy based on a context-free grammar"""

    def __init__(self, grammar, source=None):
        self.loadGrammar(grammar)
        self.loadSource(source and source or self.getDefaultSource())
        self.refresh()

    def _load(self, source):
        """load XML input source, return parsed XML document

        - a URL of a remote XML file
("http://diveintopython.org/kant.xml")
        - a filename of a local XML file
("~/diveintopython/common/py/kant.xml")
        - standard input ("-")
        - the actual XML document, as a string
        """
        sock = toolbox.openAnything(source)
        xmldoc = minidom.parse(sock).documentElement
        sock.close()
        return xmldoc

    def loadGrammar(self, grammar):
        """load context-free grammar"""
        self.grammar = self._load(grammar)
        self.refs = {}
        for ref in self.grammar.getElementsByTagName("ref"):
            self.refs[ref.attributes["id"].value] = ref

    def loadSource(self, source):
        """load source"""
        self.source = self._load(source)

    def getDefaultSource(self):
        """guess default source of the current grammar
```

```
        The default source will be one of the <ref>s that is not
        cross-referenced.  This sounds complicated but it's not.
        Example: The default source for kant.xml is
        "<xref id='section'/>", because 'section' is the one <ref>
        that is not <xref>'d anywhere in the grammar.
        In most grammars, the default source will produce the
        longest (and most interesting) output.
        """
        xrefs = {}
        for xref in self.grammar.getElementsByTagName("xref"):
            xrefs[xref.attributes["id"].value] = 1
        xrefs = xrefs.keys()
        standaloneXrefs = [e for e in self.refs.keys() if e not in
xrefs]
        if not standaloneXrefs:
            raise NoSourceError, "can't guess source, and no source
specified"
        return '<xref id="%s"/>' % random.choice(standaloneXrefs)

    def reset(self):
        """reset parser"""
        self.pieces = []
        self.capitalizeNextWord = 0

    def refresh(self):
        """reset output buffer, re-parse entire source file, and
return output

        Since parsing involves a good deal of randomness, this is an
        easy way to get new output without having to reload a grammar
file
        each time.
        """
        self.reset()
        self.parse(self.source)
        return self.output()

    def output(self):
        """output generated text"""
        return "".join(self.pieces)

    def randomChildElement(self, node):
```

```
        """choose a random child element of a node

        This is a utility method used by do_xref and do_choice.
        """
        choices = [e for e in node.childNodes
                   if e.nodeType == e.ELEMENT_NODE]
        chosen = random.choice(choices)
        if _debug:
            sys.stderr.write('%s available choices: %s\n' % \
                (len(choices), [e.toxml() for e in choices]))
            sys.stderr.write('Chosen: %s\n' % chosen.toxml())
        return chosen

    def parse(self, node):
        """parse a single XML node

        A parsed XML document (from minidom.parse) is a tree of nodes
        of various types.  Each node is represented by an instance of
the
        corresponding Python class (Element for a tag, Text for
        text data, Document for the top-level document).  The
following
        statement constructs the name of a class method based on the
type
        of node we're parsing ("parse_Element" for an Element node,
        "parse_Text" for a Text node, etc.) and then calls the method.
        """
        parseMethod = getattr(self, "parse_%s" %
node.__class__.__name__)
        parseMethod(node)

    def parse_Document(self, node):
        """parse the document node

        The document node by itself isn't interesting (to us), but
        its only child, node.documentElement, is: it's the root node
        of the grammar.
        """
        self.parse(node.documentElement)

    def parse_Text(self, node):
        """parse a text node
```

```
        The text of a text node is usually added to the output buffer
        verbatim.  The one exception is that <p class='sentence'> sets
        a flag to capitalize the first letter of the next word.  If
        that flag is set, we capitalize the text and reset the flag.
        """
        text = node.data
        if self.capitalizeNextWord:
            self.pieces.append(text[0].upper())
            self.pieces.append(text[1:])
            self.capitalizeNextWord = 0
        else:
            self.pieces.append(text)

    def parse_Element(self, node):
        """parse an element

        An XML element corresponds to an actual tag in the source:
        <xref id='...'>, <p chance='...'>, <choice>, etc.
        Each element type is handled in its own method.  Like we did
in
        parse(), we construct a method name based on the name of the
        element ("do_xref" for an <xref> tag, etc.) and
        call the method.
        """
        handlerMethod = getattr(self, "do_%s" % node.tagName)
        handlerMethod(node)

    def parse_Comment(self, node):
        """parse a comment

        The grammar can contain XML comments, but we ignore them
        """
        pass

    def do_xref(self, node):
        """handle <xref id='...'> tag

        An <xref id='...'> tag is a cross-reference to a <ref
id='...'>
        tag.  <xref id='sentence'/> evaluates to a randomly chosen
child of
```

```
        <ref id='sentence'>.
        """
        id = node.attributes["id"].value
        self.parse(self.randomChildElement(self.refs[id]))

    def do_p(self, node):
        """handle <p> tag

        The <p> tag is the core of the grammar.  It can contain almost
        anything: freeform text, <choice> tags, <xref> tags, even
other
        <p> tags.  If a "class='sentence'" attribute is found, a flag
        is set and the next word will be capitalized.  If a
"chance='X'"
        attribute is found, there is an X% chance that the tag will be
        evaluated (and therefore a (100-X)% chance that it will be
        completely ignored)
        """
        keys = node.attributes.keys()
        if "class" in keys:
            if node.attributes["class"].value == "sentence":
                self.capitalizeNextWord = 1
        if "chance" in keys:
            chance = int(node.attributes["chance"].value)
            doit = (chance > random.randrange(100))
        else:
            doit = 1
        if doit:
            for child in node.childNodes: self.parse(child)

    def do_choice(self, node):
        """handle <choice> tag

        A <choice> tag contains one or more <p> tags.  One <p> tag
        is chosen at random and evaluated; the rest are ignored.
        """
        self.parse(self.randomChildElement(node))

def usage():
    print __doc__

def main(argv):
```

```
    grammar = "kant.xml"
    try:
        opts, args = getopt.getopt(argv, "hg:d", ["help", "grammar="])
    except getopt.GetoptError:
        usage()
        sys.exit(2)
    for opt, arg in opts:
        if opt in ("-h", "--help"):
            usage()
            sys.exit()
        elif opt == '-d':
            global _debug
            _debug = 1
        elif opt in ("-g", "--grammar"):
            grammar = arg

    source = "".join(args)

    k = KantGenerator(grammar, source)
    print k.output()

if __name__ == "__main__":
    main(sys.argv[1:])
```

Ejemplo 9.2. `toolbox.py`

```
"""Miscellaneous utility functions"""

def openAnything(source):
    """URI, filename, or string --> stream

    This function lets you define parsers that take any input source
    (URL, pathname to local or network file, or actual data as a
string)
    and deal with it in a uniform manner.  Returned object is
guaranteed
    to have all the basic stdio read methods (read, readline,
readlines).
    Just .close() the object when you're done with it.

    Examples:
    >>> from xml.dom import minidom
```

```
    >>> sock = openAnything("http://localhost/kant.xml")
    >>> doc = minidom.parse(sock)
    >>> sock.close()
    >>> sock = openAnything("c:\\inetpub\\wwwroot\\kant.xml")
    >>> doc = minidom.parse(sock)
    >>> sock.close()
    >>> sock = openAnything("<ref
id='conjunction'><text>and</text><text>or</text></ref>")
    >>> doc = minidom.parse(sock)
    >>> sock.close()
    """
    if hasattr(source, "read"):
        return source

    if source == '-':
        import sys
        return sys.stdin

    # try to open with urllib (if source is http, ftp, or file URL)
    import urllib
    try:
        return urllib.urlopen(source)
    except (IOError, OSError):
        pass

    # try to open with native open function (if source is pathname)
    try:
        return open(source)
    except (IOError, OSError):
        pass

    # treat source as string
    import StringIO
    return StringIO.StringIO(str(source))
```

La ejecución del programa `kgp.py` analizará la gramática basada en XML por omisión, de `kant.xml`, y mostrará varios párrafos de filosofía al estilo de Immanuel Kant.

Ejemplo 9.3. Ejemplo de la salida de `kgp.py`

```
[usted@localhost kgp]$ python kgp.py
     As is shown in the writings of Hume, our a priori concepts, in
reference to ends, abstract from all content of knowledge; in the
study
of space, the discipline of human reason, in accordance with the
principles of philosophy, is the clue to the discovery of the
Transcendental Deduction.  The transcendental aesthetic, in all
theoretical sciences, occupies part of the sphere of human reason
concerning the existence of our ideas in general; still, the
never-ending regress in the series of empirical conditions constitutes
the whole content for the transcendental unity of apperception.  What
we have alone been able to show is that, even as this relates to the
architectonic of human reason, the Ideal may not contradict itself,
but
it is still possible that it may be in contradictions with the
employment of the pure employment of our hypothetical judgements, but
natural causes (and I assert that this is the case) prove the validity
of the discipline of pure reason.  As we have already seen, time (and
it is obvious that this is true) proves the validity of time, and the
architectonic of human reason, in the full sense of these terms,
abstracts from all content of knowledge.  I assert, in the case of the
discipline of practical reason, that the Antinomies are just as
necessary as natural causes, since knowledge of the phenomena is a
posteriori.
    The discipline of human reason, as I have elsewhere shown, is by
its very nature contradictory, but our ideas exclude the possibility
of
the Antinomies.  We can deduce that, on the contrary, the pure
employment of philosophy, on the contrary, is by its very nature
contradictory, but our sense perceptions are a representation of, in
the case of space, metaphysics.  The thing in itself is a
representation of philosophy.  Applied logic is the clue to the
discovery of natural causes.  However, what we have alone been able to
show is that our ideas, in other words, should only be used as a canon
for the Ideal, because of our necessary ignorance of the conditions.

[...corte...]
```

This is, of course, complete gibberish. Well, not complete gibberish. It is syntactically and grammatically correct (although very verbose -- Kant wasn't what you would call a get-to-the-point kind of guy). Some of it may actually be

true (or at least the sort of thing that Kant would have agreed with), some of it is blatantly false, and most of it is simply incoherent. But all of it is in the style of Immanuel Kant.

Let me repeat that this is much, much funnier if you are now or have ever been a philosophy major.

The interesting thing about this program is that there is nothing Kant-specific about it. All the content in the previous example was derived from the grammar file, `kant.xml`. If you tell the program to use a different grammar file (which you can specify on the command line), the output will be completely different.

Ejemplo 9.4. Salida de `kgp.py`, más simple

```
[usted@localhost kgp]$ python kgp.py -g binary.xml
00101001
[usted@localhost kgp]$ python kgp.py -g binary.xml
10110100
```

Daremos un vistazo más de cerca a la estructura del fichero de gramática más adelante. Por ahora, todo lo que ha de saber es que el fichero de gramática define la estructura de la salida, y que el programa `kgp.py` lee esa gramática y toma decisiones al azar sobre qué palabras poner.

9.2. Paquetes

Analizar un documentl XML es algo muy sencillo: una línea de código. Sin embargo, antes de llegar a esa línea de código hará falta dar un pequeño rodeo para hablar sobre los paquetes.

Ejemplo 9.5. Carga de un documento XML (vistazo rápido)

```
>>> from xml.dom import minidom ❶
>>> xmldoc =
minidom.parse('~/diveintopython/common/py/kgp/binary.xml')
```

❶ Esta sintaxis no la ha visto antes. Se parece al `from *módulo* import` que ya conoce y adora, pero el `"."` lo hace parecer algo más que un simple import. De hecho, `xml` es lo que conocemos como un paquete, `dom` es un paquete anidado en `xml`y `minidom` es un módulo dentro de `xml.dom`.

Suena complicado pero en realidad no lo es. Ver la implementación puede que ayude. Los paquetes son poco más que directorios de módulos; los paquetes anidados son subdirectorios. Los módulos dentro de un paquete (o paquete anidado) siguen siendo sólo ficheros `.py`, como siempre, excepto que están en un subdirectorio en lugar del directorio `lib/` principal de la instalación de Python.

Ejemplo 9.6. Estructura de ficheros de un paquete

```
Python21/           instalación raíz de Python (directorio del
ejecutable)
|
+--lib/             directorio de bibliotecas (lugar de los módulos
estándar)
   |
   +-- xml/         paquete xml (un simple directorio con otras cosas
dentro)
       |
       +--sax/      paquete xml.sax (de nuevo, sólo un directorio)
       |
       +--dom/      paquete xml.dom (contiene minidom.py)
       |
       +--parsers/  paquet xml.parsers (de uso interno)
```

Así que cuando decimos `from xml.dom import minidom`, Python interpreta eso como "busca el directorio `dom` en `xml`, y luego busca el módulo `minidom` *ahí*, e impórtalo como `minidom`". Pero Python es incluso más inteligente; no sólo puede importar módulos enteros contenidos en un paquete; puede importar deforma selectiva clases o funciones específicas de un módulo contenido en un paquete. También puede importar el paquete en sí como un módulo. La sintaxis es la misma; Python averigua lo que usted quiere basándose en la estructura de ficheros del paquete, y hace lo correcto de forma automática.

Ejemplo 9.7. Los paquetes también son módulos

```
>>> from xml.dom import minidom          ❶
>>> minidom
<module 'xml.dom.minidom' from 'C:\Python21\lib\xml\dom\minidom.pyc'>
>>> minidom.Element
<class xml.dom.minidom.Element at 01095744>
>>> from xml.dom.minidom import Element ❷
>>> Element
<class xml.dom.minidom.Element at 01095744>
>>> minidom.Element
<class xml.dom.minidom.Element at 01095744>
>>> from xml import dom                  ❸
>>> dom
<module 'xml.dom' from 'C:\Python21\lib\xml\dom\__init__.pyc'>
>>> import xml                           ❹
>>> xml
<module 'xml' from 'C:\Python21\lib\xml\__init__.pyc'>
```

❶ Aquí estamos importando un módulo (`minidom`) de un paquete anidado (`xml.dom`). El resultado es que se importa `minidom` en el espacio de nombres, y para referirnos a las clases dentro del módulo `minidom` habrá que prefijarlos con el nombre del módulo.

❷ Aquí estamos importando una clase (`Element`) de un módulo (`minidom`) que pertenece a un paquete anidado (`xml.dom`). El resultado es que se importa `Element` directamente en el espacio de nombres. Observe que esto no interfiere con la importación anterior; ahora nos podemos referir a la clase `Element` de dos maneras (pero siempre es la misma clase).

❸ Aquí estamos importando el paquete `dom` (un paquete anidado en `xml`) como si fuera un módulo. A un paquete de cualquier nivel se le puede tratar como un módulo, como verá enseguida. Incluso puede tener sus propios atributos y métodos, igual que los módulos que ya hemos visto.

❹ Aquí estamos importando el paquete raíz `xml` como módulo.

Entonces, ¿cómo puede ser que importemos y tratemos un paquete (que es sólo un directorio en el disco) como un módulo (que siempre es un fichero)? La palabra es el fichero mágico `__init__.py`. Como ve, los paquetes no son simples

directorios; son directorios con un fichero especial dentro, `__init__.py`. Este fichero define los atributos y métodos del paquete. Por ejemplo, `xml.dom` contiene una clase `Node` que está definida en `xml/dom/__init__.py`. Cuando importa un paquete como módulo (como `dom` de `xml`), en realidad está importando su fichero `__init__.py`.

Un paquete es un directorio que contiene el fichero especial `__init__.py`. El fichero `__init__.py` define los atributos y métodos del paquete. No tiene por qué definir nada; puede ser un fichero vacío, pero ha de existir. Pero si no existe `__init__.py` el directorio es sólo eso, un directorio, no un paquete, y no puede ser importado o contener módulos u otros paquetes.

Así que, ¿por qué usar paquetes? Bueno, proporcionan una manera lógica de agrupar módulos relacionados. En lugar de tener un paquete `xml` que contenga los paquetes `sax` y `dom`, los autores podrían haber escogido poner toda la funcionalidad de `sax` en `xmlsax.py` y toda la funcionalidad de `dom` en `xmldom.py`, o incluso ponerlo todo en un único módulo. Pero eso hubiera sido horrible (mientras escribo esto, el paquete XML contiene más de 3000 líneas de código) y difícil de mantener (los ficheros fuente separados implican que varias personas pueden trabajar simultáneamente en aspectos diferentes).

Si alguna vez se encuentra escribiendo un subsistema grande en Python (o más bien, cuando se dé cuenta de que su pequeño subsistema ha crecido mucho), invierta algo de tiempo en diseñar una buena arquitectura de paquete. Ésta es una de las muchas cosas en que Python destaca, así que aprovéchelo.

9.3. Análisis de XML

Como iba diciendo, analizar un documento XML es muy sencillo: una línea de código. A dónde ir partiendo de eso es cosa suya.

Ejemplo 9.8. Carga de un documento XML (ahora de verdad)

```
>>> from xml.dom import minidom
❶
>>> xmldoc =
minidom.parse('~/diveintopython/common/py/kgp/binary.xml')  ❷
>>> xmldoc
❸
<xml.dom.minidom.Document instance at 010BE87C>
>>> print xmldoc.toxml()
❹
<?xml version="1.0" ?>
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
</grammar>
```

❶ Como vio en la sección anterior, esto importa el módulo `minidom` del paquete `xml.dom`.

❷ Ésta es la línea de código que hace todo el trabajo: `minidom.parse` toma un argumento y devuelve una representación analizada del documento XML. El argumento puede ser muchas cosas; en este caso, es sólo el nombre del fichero de un documento XML en mi disco (para poder seguir, tendrá que cambiar la ruta para que coincida con el directorio de ejemplos que habrá descargado). Pero también puede pasarle un objeto de fichero, o incluso un objeto que simule un fichero. Aprovecharemos esta flexibilidad más adelante.

❸ El objeto devuelto por `minidom.parse` es un objeto de tipo `Document`, una clase descendiente de `Node`. Este objeto `Document` es la raíz de una estructura compleja de tipo árbol de objetos de Python coordinados de manear que representan completamente el documento XML que le indicó a `minidom.parse`.

❹ `toxml` es un método de la clase `Node` (y por tanto está disponible en el objeto `Document` que obtuvo de `minidom.parse`). `toxml` imprime el XML que

representa este `Node`. En el caso del nodo `Document`, imprime el documento XML entero.

Ahora que tiene el documento XML en memoria, puede empezar a recorrerlo.

Ejemplo 9.9. Obtener nodos hijos

```
>>> xmldoc.childNodes     ❶
[<DOM Element: grammar at 17538908>]
>>> xmldoc.childNodes[0] ❷
<DOM Element: grammar at 17538908>
>>> xmldoc.firstChild     ❸
<DOM Element: grammar at 17538908>
```

❶ Cada `Node` tiene un atributo `childNodes`, que es una lista de objetos `Node`. Un `Document` siempre tiene un único nodo hijo, el elemento raíz del documento XML (en este caso, el elemento `grammar`).

❷ Para obtener el primer nodo hijo (en este caso, el único), use la sintaxis normal de las listas. Recuerde, no hay nada especial aquí; es sólo una lista normal de Python que contiene objetos normales.

❸ Como obtener el primer nodo hijo de un nodo es una actividad útil y común, la clase `Node` cuenta con el atributo `firstChild`, que es sinónimo de `childNodes[0]` (también tiene un atributo `lastChild`, que es sinónimo de `childNodes[-1]`).

Ejemplo 9.10. `toxml` funciona en cualquier nodo

```
>>> grammarNode = xmldoc.firstChild
>>> print grammarNode.toxml() ❶
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
```

```
</grammar>
```

❶ Como el método `toxml` está definido en la clase `Node`, está disponible en cualquier nodo de XML, no sólo en el elemento `Document`.

Ejemplo 9.11. Los nodos hijos pueden ser un texto

```
>>> grammarNode.childNodes                    ❶
[<DOM Text node "\n">, <DOM Element: ref at 17533332>, \
<DOM Text node "\n">, <DOM Element: ref at 17549660>, <DOM Text node
"\n">]
>>> print grammarNode.firstChild.toxml()      ❷



>>> print grammarNode.childNodes[1].toxml() ❸
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
>>> print grammarNode.childNodes[3].toxml() ❹
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
>>> print grammarNode.lastChild.toxml()       ❺
```

❶ Viendo el XML de `binary.xml`, podría pensar que el nodo `grammar` sólo tiene dos hijos, los dos elementos `ref`. Pero está olvidándose de algo: ¡los retornos de carro! Tras `'<grammar>'` y antes del primer `'<ref>'` hay un retorno de carro, y este texto cuenta como nodo hijo del elemento `grammar`. De forma similar, hay un retorno de carro tras cada `'</ref>'`; que también cuentan como nodos hijos. Así que `grammar.childNodes` en realidad es una lista de 5 objetos: 3 objetos `Text` y 2 `Element`.

❷ El primer hijo es un objeto `Text` que representa el retorno de carro tras la etiqueta `'<grammar>'` y antes de la primera etiqueta `'<ref>'`.

❸ El segundo hijo es un objeto `Element` que representa el primer elemento `ref`.

❹ El cuarto hijo es un objeto `Element` que representa el segundo elemento `ref`.

❺ El último hijo es un objeto `Text` que representa el retorno de carro que hay antes de la etiqueta de fin `'</ref>'` antes de la etiqueta `'</grammar>'`.

Ejemplo 9.12. Explorando en busca del texto

```
>>> grammarNode
<DOM Element: grammar at 19167148>
>>> refNode = grammarNode.childNodes[1] ❶
>>> refNode
<DOM Element: ref at 17987740>
>>> refNode.childNodes                  ❷
[<DOM Text node "\n">, <DOM Text node "  ">, <DOM Element: p at
19315844>, \
<DOM Text node "\n">, <DOM Text node "  ">, \
<DOM Element: p at 19462036>, <DOM Text node "\n">]
>>> pNode = refNode.childNodes[2]
>>> pNode
<DOM Element: p at 19315844>
>>> print pNode.toxml()                 ❸
<p>0</p>
>>> pNode.firstChild                    ❹
<DOM Text node "0">
>>> pNode.firstChild.data               ❺
u'0'
```

❶ Como vio en los ejemplos anteriores, el primer elemento `ref` es `grammarNode.childNodes[1]`, ya que childNodes[0] es el nodo `Text` del retorno de carro.

❷ El elemento `ref` tiene su propio juego de nodos hijos, uno por el retorno de carro, otro para los espacios, otro más para el elemento `p`, *etc.*.

❸ Puede usar el método `toxml` incluso aquí, en las profundidades del documento.

❹ El elemento `p` tiene un solo nodo hijo (no podría adivinarlo por este ejemplo, pero mire `pNode.childNodes` si no me cree), que es un nodo `Text` para el carácter `'0'`.

5. El atributo `.data` de un nodo `Text` le da la cadena que representa el texto del nodo. Pero, ¿qué es esa `'u'` delante de la cadena? La respuesta para eso merece su propia sección.

9.4. Unicode

Unicode es un sistema para representar caracteres de todos los diferentes idiomas en el mundo. Cuando Python analiza un documento XML, todos los datos se almacenan en memoria como unicode.

Llegaremos a eso en un momento, pero antes, unos antecedentes.

Nota histórica. Antes de unicode, había diferentes sistemas de codificación de caracteres para cada idioma, cada uno usando los mismos números (0-255) para representar los caracteres de ese lenguaje. Algunos (como el ruso) tienen varios estándares incompatibles que dicen cómo representar los mismos caracteres; otros idiomas (como el japonés) tienen tantos caracteres que precisan más de un byte. Intercambiar documentos entre estos sistemas era difícil porque no había manera de que un computador supiera con certeza qué esquema de codificación de caracteres había usado el autor del documento; el computador sólo veía números, y los números pueden significar muchas cosas. Piense entonces en almacenar estos documentos en el mismo sitio (como en una tabla de una base de datos); necesitaría almacenar el tipo de codificación junto con cada texto, y asegurarse de adjuntarlo con el texto cada vez que accediese a él. Ahora imagine documentos multilingües, con caracteres de varios idiomas en el mismo document. (Habitualmente utilizaban códigos de escape para cambiar de modos; ¡puf!, está en modo ruso koi8-r así que el carácter 241 significa esto; ¡puf!, ahora está en modo Mac Greek, así que el carácter 241 significa otra cosa. Y así con todo). Para resolver estos problemas se diseñó unicode.

Para resolver estos problemas, unicode representa cada carácter como un número de 2 bytes, de 0 a 65535.[10] Cada número de 2 bytes representa un único carácter utilizado en al menos un idioma del mundo (los caracteres que se usan

en más de un idioma tienen el mismo código numérico). Hay exactamente 1 número por carácter, y exactamente 1 carácter por número. Los datos de unicode nunca son ambiguos.

Por supuesto, sigue estando el problema de todos esos sistemas de codificación anticuados. Por ejemplo, el ASCII de 7 bits que almacena los caracteres ingleses como números del 0 al 127 (65 es la "`A`", mayúscula, 97 es la "`a`" minúscula, *etc.*). El inglés tiene un alfabeto sencillo, así que se puede expresar en ASCII de 7 bits. Los idiomas europeos occidentales como el francés, español y alemán usan todos un sistema llamado ISO-8859-1 (también conocido como "latin-1"), que usa los caracteres del ASCII de 7 bits del 0 al 127, pero lo extiende en el rango 128-255 para tener caracteres como n-con-una-tilde-sobre-ella (241) y u-con-dos-puntitos-sobre-ella (252). Y unicode usa los mismos caracteres que el ASCII de 7 bits para los números del 0 al 127, y los mismos caracteres que ISO-8859-1 del 128 al 255, y de ahí en adelante se extiende para otros lenguajes que usan el resto de los números, del 256 al 65535.

Puede que en algún momento al tratar con datos unicode tengamos la necesidad de convertirlos en alguno de estos otros sistemas anticuados. Por ejemplo, por necesidad de integración con algún sistema computador que espera que sus datos estén en un esquema específico de 1 byte, o para imprimirlo en alguna terminal o impresora que desconozca unicode. O para almacenarlo en un documento XML que especifique explícitamente la codificación de los caracteres.

Y dicho esto, volvamos a Python.

Python trabaja con unicode desde la versión 2.0 del lenguaje. El paquete XML utiliza unicode para almacenar todos los datos XML, pero puede usar unicode en cualquier parte.

Ejemplo 9.13. Presentación de unicode

```
>>> s = u'Dive in'          ❶
>>> s
```

```
u'Dive in'
>>> print s                    ❷
Dive in
```

❶ Para crear una cadena unicode en lugar de una ASCII normal, añada la letra "u" antes de la cadena. Observe que esta cadena en particular no tiene ningún carácter que no sea ASCII. Esto no es problema; unicode es un superconjunto de ASCII (un superconjunto muy grande, por cierto), así que también se puede almacenar una cadena ASCII normal como unicode.

❷ Cuando Python imprime una cadena intentará convertirla a la codificación por omisión, que suele ser ASCII (más sobre esto en un momento). Como la cadena unicode está hecha de caracteres que a la vez son ASCII, imprimirlos tiene el mismo resultado que imprimir una cadena ASCII normal; la conversión es consistente, y si no supiera que `s` era una cadena unicode nunca llegaría a notar la diferencia.

Ejemplo 9.14. Almacenamiento de caracteres no ASCII

```
>>> s = u'La Pe\xf1a'          ❶
>>> print s                    ❷
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
UnicodeError: ASCII encoding error: ordinal not in range(128)
>>> print s.encode('latin-1') ❸
La Peña
```

❶ La verdadera ventaja de unicode, por supuesto, es su capacidad de almacenar caracteres que no son ASCII, como la "ñ" española. El carácter unicode para la ñ es `0xf1` en hexadecimal (241 en decimal), que se puede escribir así: `\xf1`[11]

❷ ¿Recuerda que dije que la función `print` intenta convertir una cadena unicode en ASCII para poder imprimirla? Bien, eso no funcionará aquí, porque la cadena unicode contiene caracteres que no son de ASCII, así que Python produce un error `UnicodeError`.

❸ Aquí es donde entra la conversión-de-unicode-a-otros-esquemas-de-codificación. `s` es una cadena unicode, pero `print` sólo puede imprimir

cadenas normales. Para resolver este problema, llamamos al método `encode`, disponible en cada cadena unicode, para convertir la cadena unicode en una cadena normal en el esquema dado, que le pasamos como parámetro. En este caso usamos `latin-1` (también conocido como `iso-8859-1`), que incluye la ñ (mientras que el código ASCII no, ya que sólo incluye caracteres numerados del 0 al 127).

¿Recuerda cuando le dije que Python normalmente convierte el unicode a ASCII cuando necesita hacer una cadena normal partiendo de una unicode? Bien, este esquema por omisión es una opción que puede modificar.

Ejemplo 9.15. `sitecustomize.py`

```
# sitecustomize.py                          ❶
# this file can be anywhere in your Python path,
# but it usually goes in ${pythondir}/lib/site-packages/
import sys
sys.setdefaultencoding('iso-8859-1') ❷
```

❶ `sitecustomize.py` es un *script* especial; Python lo intentará importar cada vez que arranca, así que se ejecutará automáticamente cualquier código que incluya. Como menciona el comentario, puede estar en cualquier parte (siempre que `import` pueda encontrarlo), pero normalmente se incluye en el directorio `site-packages` dentro del directorio `lib` de Python.

❷ La función `setdefaultencoding` establece, pues eso, la codificación por omisión. Éste es el esquema de codificación que Python intentará usar cada vez que necesite convertir automáticamente una cadena unicode a una normal.

Ejemplo 9.16. Efectos de cambiar la codificación por omisión

```
>>> import sys
>>> sys.getdefaultencoding() ❶
'iso-8859-1'
>>> s = u'La Pe\xf1a'
>>> print s                  ❷
```

```
La Peña
```

❶ Este ejemplo asume que ha hecho en el fichero `sitecustomize.py` los cambios mencionados en el ejemplo anterior, y reiniciado Python. Si la codificación por omisión sigue siendo `'ascii'`, significa que no configuró adecuadamente el `sitecustomize.py`, o que no ha reiniciado Python. La codificación por omisión sólo se puede cambiar durante el inicio de Python; no puede hacerlo más adelante. (Debido a ciertos trucos de programación en los que no voy a entrar ahora, ni siquiera puede invocar a `sys.setdefaultencoding` tras que Python haya arrancado. Busque "`setdefaultencoding`" en el `site.py` para averiguar la razón).

❷ Ahora que el esquepa de codificación por omisión incluye todos los caracteres que usa en la cadena, Python no tiene problemas en autoconvertir la cadena e imprimirla.

Ejemplo 9.17. Especificación de la codificación en ficheros `.py`

Si va a almacenar caracteres que no sean ASCII dentro de código de Python, necesitará especificar la codificación en cada fichero `.py` poniendo una declaración de codificación al inicio de cada uno. Esta declaración indica que el fichero `.py` contiene UTF-8:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
```

Ahora, ¿qué pasa con XML? Bueno, cada documento XML está en una codificación específica. ISO-8859-1 es una codificación popular para los datos en idiomas europeos occidentales. KOI8-R es habitual en textos rusos. Si se especifica, la codificación estará en la cabecera del documento XML.

Ejemplo 9.18. `russiansample.xml`

```
<?xml version="1.0" encoding="koi8-r"?>       ❶
<preface>
<title>Предисловие</title>                    ❷
```

```
</preface>
```

❶ Este ejemplo está extraído de un documento XML ruso real; es parte de una traducción al ruso de este mismo libro. Observe la codificación especificada en la cabecera, `koi8-r`.

❷ Éstos son caracteres cirílicos que, hasta donde sé, constituyen la palabra rusa para “Prefacio” . Si abre este fichero en un editor de textos normal los caracteres probablemente parezcan basura, puesto que están codificados usando el esquema `koi8-r`, pero se mostrarán en `iso-8859-1`.

Ejemplo 9.19. Análisis de `russiansample.xml`

```
>>> from xml.dom import minidom
>>> xmldoc = minidom.parse('russiansample.xml') ❶
>>> title = xmldoc.getElementsByTagName('title')[0].firstChild.data
>>> title                                                          ❷
u'\u041f\u0440\u0435\u0434\u0438\u0441\u043b\u043e\u0432\u0438\u0435'
>>> print title                                                    ❸
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
UnicodeError: ASCII encoding error: ordinal not in range(128)
>>> convertedtitle = title.encode('koi8-r')                        ❹
>>> convertedtitle
'\xf0\xd2\xc5\xc4\xc9\xd3\xcc\xcf\xd7\xc9\xc5'
>>> print convertedtitle                                           ❺
Предисловие
```

❶ Estoy asumiendo que ha guardado el ejemplo anterior en el directorio actual con el nombre `russiansample.xml`. También estoy asumiendo que ha cambiado la codificación por omisión de vuelta a `'ascii'` eliminando el fichero `sitecustomize.py`, o al menos que ha comentado la línea con `setdefaultencoding`.

❷ Observe que el texto de la etiqueta `title` (ahora en la variable `title`, gracias a esa larga concatenación de funciones de Python que me he saltado por las prisas y, para su disgusto, no explicaré hasta la siguiente sección), el texto, decía, del elemento `title` del documento XML está almacenado en unicode.

❸ No es posible imprimir el título porque esta cadena unicode contiene

caracteres extraños a ASCII, de manera que Python no la convertirá a ASCII ya que eso no tendría sentido.

❹ Sin embargo, usted puede convertirlo de forma explícita a `koi8-r`, en cuyo caso obtendrá una cadena (normal, no unicode) de caracteres de un solo byte (`f0`, `d2`, `c5`, *etc.*) que son la versión codificada en `koi8-r` de los caracteres de la cadena unicode original.

❺ Imprimir la cadena codificada `koi8-r` probablemente muestre basura en su pantalla, porque el IDE de Python los interpretará como caracteres `iso-8859-1`, no `koi8-r`. Pero al menos los imprime. (Y si lo mira atentamente, verá que es la misma basura que vio cuando abrió el documento original en XML en un editor de texto que no sepa unicode. Python lo ha convertido de `koi8-r` a unicode cuando analizó el documento XML, y usted lo ha convertido de vuelta).

Para resumir, unicode en sí es un poco intimidante si nunca lo ha visto antes, pero los datos en unicode son muy fáciles de manejar con Python. Si sus documentos XML están todos en ASCII de 7 bits (como los ejemplos de este capítulo), nunca tendrá que pensar sobre unicode, literalmente. Python convertirá los datos ASCII de los documentos XML en unicode mientras lo analiza, y los autoconvertirá a ASCII cuando sea necesario, y ni siquiera lo notará. Pero si necesita tratar con otros idiomas, Python está preparado.

Lecturas complementarias

- Unicode.org es la página del estándar unicode, e incluye una breve introducción técnica.
- El Unicode Tutorial tiene muchos más ejemplos sobre el uso de funciones unicode de Python, incluyendo la manera de forzar a Python a convertir unicode en ASCII incluso cuando él probablemente no querría.
- El PEP 263 entra en detalles sobre cómo y cuándo definir una codificación de caracteres en sus ficheros `.py`.

Footnotes

[10] Tristemente, esto no es más que una simplificación *extrema*. Unicode ha sido extendido para poder tratar textos clásicos chinos, coreanos y japoneses, que tienen tantos caracteres diferentes que el sistema unicode de 2 bytes no podía representarlos todos. Pero Python no soporta eso de serie, y no sé si hay algún proyecto en marcha para añadirlo. Hemos llegado a los límites de mi conocimiento, lo siento.

[11] N. del T.: Dado que está leyendo este texto en español, es bastante probable que también cuente con una ñ en el teclado y pueda escribirla sin recurrir al hexadecimal, pero aún así he decidido mantener el ejemplo tal cual está en el original, ya que ilustra el concepto.

9.5. Búsqueda de elementos

Recorrer un documento XML saltando por cada nodo puede ser tedioso. Si está buscando algo en particular, escondido dentro del documento XML, puede usar un atajo para encontrarlo rápidamente: `getElementsByTagName`.

En esta sección usaremos el fichero de gramática `binary.xml`, que es así:

Ejemplo 9.20. `binary.xml`

```
<?xml version="1.0"?>
<!DOCTYPE grammar PUBLIC "-//diveintopython.org//DTD Kant Generator
Pro v1.0//EN" "kgp.dtd">
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
</grammar>
```

Tiene dos `ref`, `'bit'` y `'byte'`. Un `bit` puede ser `'0'` o `'1'`, y un `byte` es 8 `bits`.

Ejemplo 9.21. Presentación de `getElementsByTagName`

```
>>> from xml.dom import minidom
>>> xmldoc = minidom.parse('binary.xml')
>>> reflist = xmldoc.getElementsByTagName('ref') ❶
>>> reflist
[<DOM Element: ref at 136138108>, <DOM Element: ref at 136144292>]
>>> print reflist[0].toxml()
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
>>> print reflist[1].toxml()
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
```

❶ `getElementsByTagName` toma un argumento, el nombre del elemento que desea encontrar. Devuelve una lista de elementos `Element` que corresponden a los elementos XML que llevan ese nombre. En este caso, encontró dos elementos `ref`.

Ejemplo 9.22. Puede buscar cualquier elemento

```
>>> firstref = reflist[0]                    ❶
>>> print firstref.toxml()
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
>>> plist = firstref.getElementsByTagName("p") ❷
>>> plist
[<DOM Element: p at 136140116>, <DOM Element: p at 136142172>]
>>> print plist[0].toxml()                   ❸
<p>0</p>
>>> print plist[1].toxml()
<p>1</p>
```

❶ Siguiendo con el ejemplo anterior, el primer objeto de `reflist` es el elemento `ref 'bit'`.

❷ Puede usar el mismo método `getElementsByTagName` sobre este `Element` para encontrar todos los elementos `<p>` dentro del elemento `ref 'bit'`.

❸ Igual que antes, el método `getElementsByTagName` devuelve una lista de todos los elementos que encuentra. En este caso hay dos, uno por cada bit.

Ejemplo 9.23. La búsqueda es recursiva

```
>>> plist = xmldoc.getElementsByTagName("p") ❶
>>> plist
[<DOM Element: p at 136140116>, <DOM Element: p at 136142172>, <DOM
Element: p at 136146124>]
>>> plist[0].toxml()                          ❷
'<p>0</p>'
>>> plist[1].toxml()
'<p>1</p>'
>>> plist[2].toxml()                          ❸
'<p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>'
```

❶ Observe atentamente la diferencia entre este ejemplo y el anterior. Antes estábamos buscando elementos `p` dentro de `firstref`, pero ahora estamos buscando elementos `p` dentro de `xmldoc`, el objeto raíz que representa todo el documento XML. Esto *encuentra* los elementos `p` anidados dentro de los elementos `ref` dentro del elemento `grammar`.

❷ Los dos primeros elementos `p` están dentro del primer `ref` (el `ref 'bit'`).

❸ El último elemento `p` es el que está dentro del segundo `ref` (el `ref 'byte'`).

9.6. Acceso a atributos de elementos

Los elementos de XML pueden tener uno o más atributos, y es increíblemente sencillo acceder a ellos una vez analizado el documento XML.

En esta sección usaremos el fichero de gramática `binary.xml` que vio en la sección anterior.

Esta sección puede ser un poco confusa, debido a que la terminología se solapa. Los elementos de documentos XML tienen atributos, y los objetos de Python también. Cuando analizamos un documento XML obtenemos un montón de objetos de Python que representan todas las partes del documento XML, y algunos de estos objetos de Python representan atributos de los elementos de XML. Pero los objetos (de Python) que representan los atributos (de XML) también tienen atributos (de Python), que se usan para acceder a varias partes del atributo (de XML) que representa el objeto. Le dije que era confuso. Estoy abierto a sugerencias para distinguirlos con más claridad.

Ejemplo 9.24. Acceso a los atributos de un elemento

```
>>> xmldoc = minidom.parse('binary.xml')
>>> reflist = xmldoc.getElementsByTagName('ref')
>>> bitref = reflist[0]
>>> print bitref.toxml()
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
>>> bitref.attributes                    ❶
<xml.dom.minidom.NamedNodeMap instance at 0x81e0c9c>
>>> bitref.attributes.keys()             ❷ ❸
[u'id']
>>> bitref.attributes.values()           ❹
[<xml.dom.minidom.Attr instance at 0x81d5044>]
>>> bitref.attributes["id"]              ❺
<xml.dom.minidom.Attr instance at 0x81d5044>
```

❶ Cada objeto `Element` tiene un atributo llamado `attributes`, que es un objeto `NamedNodeMap`. Suena feo pero no lo es, porque `NamedNodeMap` es un objeto que actúa como un diccionario, así que ya sabemos cómo usarlo.

❷ Si tratamos el `NamedNodeMap` como un diccionario, podemos obtener una lista de nombres de los atributos de este elemento usando `attributes.keys()`. Este elemento sólo tiene un atributo, `'id'`.

❸ Los nombres de los atributos se almacenan en unicode, como cualquier otro texto dentro de un documento XML.

❹ De nuevo tratamos el `NamedNodeMap` como un diccionario para obtener una lista de valores de los atributos usando `attributes.values()`. Los valores en sí son objetos, de tipo `Attr`. Verá cómo obtener información útil de este objeto en el siguiente ejemplo.

❺ Seguimos tratando el `NamedNodeMap` como un diccionario y ahora accedemos a un atributo individual, usando la sintaxis normal de un diccionario. (Los lectores que hayan prestado muchísima atención ya sabrán cómo hace este magnífico truco la clase `NamedNodeMap`: definiendo un método especial `__getitem__`. Otros lectores pueden confortarse con el hecho de que no necesitan entender su funcionamiento para poder usarlo de forma efectiva.

Ejemplo 9.25. Acceso a atributos individuales

```
>>> a = bitref.attributes["id"]
>>> a
<xml.dom.minidom.Attr instance at 0x81d5044>
>>> a.name  ❶
u'id'
>>> a.value ❷
u'bit'
```

❶ El objeto `Attr` representa un único atributo de XML de un único elemento XML. El nombre del atributo (el mismo nombre que usamos para encontrar el objeto en el pseudodiccionario `NamedNodeMap` `bitref.attributes`) se almacena en `a.name`.

❷ El valor del texto de este atributo XML se almacena en `a.value`.

Al igual que un diccionario, los atributos de un elemento XML no tienen orden. *Puede que* los atributos estén listados en un cierto orden en el documento XML original, y *Puede que* los objetos `Attr` estén listados en un cierto orden cuando se convierta el documento XML en objetos de Python, pero estos órdenes son arbitrarios y no deberían tener un significado

especial. Siempre debería acceder a los atributos por su nombre, como claves de un diccionario.

9.7. Transición

Bien, esto era el material más duro sobre XML. El siguiente capítulo continuará usando estos mismos programas de ejemplo, pero centrándose en otros aspectos que hacen al programa más flexible: uso de flujos[12] para proceso de entrada, uso de `getattr` para despachar métodos, y uso de opciones en la línea de órdenes para permitir a los usuarios reconfigurar el programa sin cambiar el código.

Antes de pasar al siguiente capítulo, debería sentirse a gusto haciendo las siguientes cosas:

- Analizar documentos XML usando minidom_modulename;, hacer búsquedas dentro del documento analizado, y acceder a atributos e hijos de elementos arbitrarios
- Organizar bibliotecas completas en paquetes
- Convertir cadenas unicode a diferentes codificaciones de caracteres

Footnotes

[12] *streams*

Capítulo 10. *Scripts* y flujos

10.1. Abstracción de fuentes de datos

Uno de los puntos más fuertes de Python es su enlace dinámico, y un uso muy potente del enlace dinámico es el *objeto tipo fichero*.

Muchas funciones que precisan una entrada de datos podrían simplemente tomar un nombre fichero, abrirlo para lectura, leerlo y cerrarlo cuando hubieran acabado. Pero no lo hacen. En su lugar toman un *objeto de tipo fichero*.

En su caso más sencillo un *objeto tipo fichero* es cualquier objeto con un método `read` con un parámetro `size` opcional, que devuelve una cadena. Cuando se le invoca sin el parámetro `size`, lee cualquier cosa que quede dentro de la fuente de datos y devuelve todos esos datos en una sola cadena. Cuando se la invoca con el parámetro `size`, lee y de vuelve sólo esa cantidad de datos desde la fuente; cuando la invocan de nuevo continúa donde lo dejó y devuelve el siguiente paquete de datos.

Así es como funciona la lectura desde ficheros reales; la diferencia es que no nos estamos limitando a ficheros. La fuente de entrada puede ser cualquier cosa: un fichero en el disco, una página web e incluso una cadena de contenido fijo. Mientras pase a la función un objeto que parezca un fichero, y la función sólo

llame al método `read` del objeto, la función puede manejar cualquier tipo de fuente de datos sin necesidad de código específico para cada tipo.

En caso de que se esté preguntando qué tiene esto que ver con el procesamiento de XML, `minidom.parse` es una de esas funciones que puede tomar objetos de fichero.

Ejemplo 10.1. Análisis de XML desde un fichero

```
>>> from xml.dom import minidom
>>> fsock = open('binary.xml')     ❶
>>> xmldoc = minidom.parse(fsock) ❷
>>> fsock.close()                  ❸
>>> print xmldoc.toxml()           ❹
<?xml version="1.0" ?>
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
</grammar>
```

❶ Primero abrimos el fichero del disco. Esto nos da un objeto de fichero.

❷ Pasamos el objeto de fichero a `minidom.parse`, que llama al método `read` de `fsock` y lee el documento XML del fichero.

❸ Asegúrese de llamar al método `close` del objeto de fichero tras haber terminado con él. `minidom.parse` no lo hará automáticamente.

❹ Llamar al método `toxml()` del documento XML devuelto lo imprime entero.

Bien, todo eso parece una colosal pérdida de tiempo. Después de todo, ya hemos visto que `minidom.parse` puede tomar el nombre del fichero y hacer automáticamente esas tonterías de abrirlo y cerrarlo. Y es cierto que si sabes que sólo vas a analizar un fichero local, se le puede pasar el nombre y

`minidom.parse` es lo suficientemente inteligente para Hacer Lo Correcto™. Pero observe lo parecido (y sencillo) que es analizar un documento XML sacado directamente de Internet.

Ejemplo 10.2. Análisis de XML desde una URL

```
>>> import urllib
>>> usock = urllib.urlopen('http://slashdot.org/slashdot.rdf') ❶
>>> xmldoc = minidom.parse(usock)                              ❷
>>> usock.close()                                              ❸
>>> print xmldoc.toxml()                                       ❹
<?xml version="1.0" ?>
<rdf:RDF xmlns="http://my.netscape.com/rdf/simple/0.9/"
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

<channel>
<title>Slashdot</title>
<link>http://slashdot.org/</link>
<description>News for nerds, stuff that matters</description>
</channel>

<image>
<title>Slashdot</title>
<url>http://images.slashdot.org/topics/topicslashdot.gif</url>
<link>http://slashdot.org/</link>
</image>

<item>
<title>To HDTV or Not to HDTV?</title>
<link>http://slashdot.org/article.pl?sid=01/12/28/0421241</link>
</item>

[...corte...]
```

❶ Como vio en un capítulo anterior, `urlopen` toma la URL de una página web y devuelve un objeto de tipo fichero. Aún más importante, este objeto tiene un método `read` que devuelve el HTML fuente de la página web.

❷ Ahora pasamos el objeto de fichero a `minidom.parse` que invoca obedientemente el método `read` del objeto y analiza los datos XML que

devuelve `read`. El hecho de que los datos XML estén viniendo directamente de la página web es completamente irrelevante. `minidom.parse` no sabe qué son las páginas web, y no le importan las páginas web; sólo sabe tratar con ficheros de tipo objeto.

❸ Tan pronto como haya acabado, asegúrese de cerrar el objeto de fichero que le da `urlopen`.

❹ Por cierto, esta URL es real y verdaderamente es XML. Es una representación XML de los últimos titulares de Slashdot, un sitio de noticias y chismorreo tecnológico.

Ejemplo 10.3. Análisis de XML desde una cadena (manera sencilla pero inflexible)

```
>>> contents = "<grammar><ref
id='bit'><p>0</p><p>1</p></ref></grammar>"
>>> xmldoc = minidom.parseString(contents) ❶
>>> print xmldoc.toxml()
<?xml version="1.0" ?>
<grammar><ref id="bit"><p>0</p><p>1</p></ref></grammar>
```

❶ `minidom` tiene un método llamado `parseString` que toma un documento XML completo como cadena y lo analiza. Puede usar esto en lugar de `minidom.parse` si sabe que tiene el documento XML completo en una cadena.

Bien, así que podemos usar la función `minidom.parse` para analizar tanto ficheros locales como URLs remotas, pero para analizar cadenas usamos... una función diferente. Eso significa que si quiere poder tomar la entrada desde un fichero, una URL o una cadena, necsita una lógica especial para comprobar si es una cadena, y llamar a `parseString` en su lugar. Qué insatisfactorio.

Si hubiera una manera de convertir una cadena en un objeto de fichero, entonces podríamos pasársela simplemente a `minidom.parse`. De hecho, hay un módulo diseñado específicamente para hacer eso: `StringIO`.

Ejemplo 10.4. Presentación de `StringIO`

```
>>> contents = "<grammar><ref
id='bit'><p>0</p><p>1</p></ref></grammar>"
>>> import StringIO
>>> ssock = StringIO.StringIO(contents)   ❶
>>> ssock.read()                          ❷
"<grammar><ref id='bit'><p>0</p><p>1</p></ref></grammar>"
>>> ssock.read()                          ❸
''
>>> ssock.seek(0)                         ❹
>>> ssock.read(15)                        ❺
'<grammar><ref i'
>>> ssock.read(15)
"d='bit'><p>0</p"
>>> ssock.read()
'><p>1</p></ref></grammar>'
>>> ssock.close()                         ❻
```

❶ El módulo `StringIO` contiene una única clase que también se llama `StringIO`, que le permite convertir una cadena en un objeto de tipo fichero. La clase `StringIO` toma la cadena como parámetro al crear una instancia.

❷ Ahora tenemos un objeto de fichero y podemos hacer con él todo tipo de cosas que haríamos con un fichero. Como usar `read`, que devuelve la cadena original.

❸ Invocar de nuevo a `read` devuelve una cadena vacía. Así es como funciona también un fichero real; una vez leído el fichero entero, no se puede leer más allá sin volver explícitamente al principio del fichero. El objeto `StringIO` funciona de la misma manera.

❹ Puede volver al comienzo de la cadena de forma explícita usando el método `seek` del objeto `StringIO`, igual que lo haría con un fichero.

❺ También puede leer la cadena en trozos pasando un parámetro `size` al método `read`.

❻ En cualquier momento, `read` puede devolver el resto de la cadena que aún no se ha leído. Todo esto es idéntico a como funciona un objeto de fichero; de ahí el término *objeto de tipo fichero*.

Ejemplo 10.5. Análisis de XML desde una cadena (al estilo fichero)

```
>>> contents = "<grammar><ref
id='bit'><p>0</p><p>1</p></ref></grammar>"
>>> ssock = StringIO.StringIO(contents)
>>> xmldoc = minidom.parse(ssock) ❶
>>> ssock.close()
>>> print xmldoc.toxml()
<?xml version="1.0" ?>
<grammar><ref id="bit"><p>0</p><p>1</p></ref></grammar>
```

❶ Ahora podemos pasar el objeto de fichero (realmente una `StringIO`) a `minidom.parse`, que llamará al método `read` del objeto y lo analizará felizmente, sin llegar nunca a saber que la entrada provino de una cadena de contenido fijo.

Así que ya sabemos cómo usar una única función, `minidom.parse`, para analizar un documento XML almacenado en una página web, en un fichero local o en una cadena fija. Para una página web usaremos `urlopen` para obtener un objeto de tipo fichero; para un fichero local usaremos `open`; y para una cadena, usaremos `StringIO`. Ahora llevémoslo un paso más adelante y generalicemos también *éstas* diferencias.

Ejemplo 10.6. `openAnything`

```
def openAnything(source):                          ❶
    # try to open with urllib (if source is http, ftp, or file URL)
    import urllib
    try:
        return urllib.urlopen(source)              ❷
    except (IOError, OSError):
        pass

    # try to open with native open function (if source is pathname)
    try:
        return open(source)                        ❸
    except (IOError, OSError):
```

```
        pass

    # treat source as string
    import StringIO
    return StringIO.StringIO(str(source))  ❹
```

❶ La función `openAnything` toma un solo parámetro, `source`, y devuelve un fichero de tipo objeto. `source` es una cadena de algún tipo; que puede ser una URL (como `'http://slashdot.org/slashdot.rdf'`), una ruta absoluta o parcial a un fichero local (como `'binary.xml'`) o una cadena que contenga los datos del XML a ser analizado, propiamente dicho.

❷ Primero vemos si `source` es una URL. Lo hacemos mediante fuerza bruta: intentamos abrirlo como una URL e ignoramos los errores causados por abrir algo que no es una URL. En realidad esto es elegante en el sentido de que si `urllib` admite alguna vez nuevos tipos de URL, usted les estará dando soporte sin tener que reprogramarlo. Si `urllib` es capaz de abrir `source`, entonces `return` le saca de la función inmediatamente y no se llegan a ejecutar las siguientes sentencias `try`.

❸ Por otro lado, si `urllib` le grita diciéndole que `source` no es una URL válida, asumirá que es la ruta a un fichero en el disco e intentará abrirlo. De nuevo, no usaremos ninguna sofisticación para comprobar si `source` es un nombre de fichero válido o no (de todas maneras, las reglas para la validez de los nombres de fichero varían mucho entre plataformas, así que probablemente lo haríamos mal). En su lugar, intentará abrir el fichero a ciegas y capturaremos cualquier error en silencio.

❹ A estas alturas debemos asumir que `source` es una cadena que contiene los datos (ya que lo demás no ha funcionado), así que usaremos `StringIO` para crear un objeto de tipo fichero partiendo de ella y devolveremos eso. (En realidad, dado que usamos la función `str`, `source` ni siquiera tiene por qué ser una cadena; puede ser cualquier objeto y usaremos su representación como cadena, tal como define su método especial `__str__`).

Ahora podemos usar lafunción `openAnything` junto con `minidom.parse` para construir una función que tome una `source` que se refiera de alguna manera a

un documento XML (una URL, un fichero local o un documento XML fijado en una cadena) y lo analice.

Ejemplo 10.7. Uso de `openAnything` en `kgp.py`

```
class KantGenerator:
    def _load(self, source):
        sock = toolbox.openAnything(source)
        xmldoc = minidom.parse(sock).documentElement
        sock.close()
        return xmldoc
```

10.2. Entrada, salida y error estándar

Los usuarios de UNIX ya estarán familiarizados con el concepto de entrada estándar, salida estándar y salida estándar de error. Esta sección es para los demás.

La salida estándar y la salida estándar de error (se suelen abreviar `stdout` y `stderr`) son tuberías[13] incorporadas en todo sistema UNIX. Cuando se imprime algo, sale por la tubería `stdout`; cuando el programa aborta e imprime información de depuración (como el *traceback* de Python), sale por la tubería `stderr`. Ambas se suelen conectar a la ventana terminal donde está trabajando usted para que cuando el programa imprima, usted vea la salida, y cuando un programa aborte, pueda ver la información de depuración. (Si está trabajando en un sistema con un IDE de Python basado en ventanas, `stdout` y `stderr` descargan por omisión en la "Ventana interactiva").

Ejemplo 10.8. Presentación de `stdout` y `stderr`

```
>>> for i in range(3):
...     print 'Dive in'            ❶
Dive in
Dive in
Dive in
>>> import sys
```

```
>>> for i in range(3):
...     sys.stdout.write('Dive in') ❷
Dive inDive inDive in
>>> for i in range(3):
...     sys.stderr.write('Dive in') ❸
Dive inDive inDive in
```

❶ Como pudo ver en Ejemplo 6.9, "Contadores simples", se puede utilizar la función incorporada `range` de Python para construir bucles contadores simples que repitan algo un número determinado de veces.

❷ `stdout` es un objeto de tipo fichero; llamar a su función `write` imprimirá la cadena que se le pase. De hecho, esto es lo que hace realmente la función `print`; añade un retorno de carro al final de la cadena que esté imprimiendo e invoca a `sys.stdout.write`.

❸ En el caso más simple `stdout` y `stderr` envían su salida al mismo lugar: al IDE de Python (si está en uno), o a la terminal (si ejecuta Python desde la línea de órdenes). Igual que `stdout`, `stderr` no añade un retorno de carro por sí sola; si lo desea, añádalo usted mismo.

`stdout` y `stderr` son ambos objetos de tipo fichero, como aquellos de los que hablamos en Sección 10.1, "Abstracción de fuentes de datos", pero ambos son sólo de escritura. No tienen método `read`, sólo `write`. Aún así son objetos de tipo fichero, y puede asignarles cualquier otro objeto de tipo fichero para redirigir su salida.

Ejemplo 10.9. Redirección de la saliad

```
[usted@localhost kgp]$ python stdout.py
Dive in
[usted@localhost kgp]$ cat out.log
This message will be logged instead of displayed
```

(On Windows, you can use `type` instead of `cat` to display the contents of a file.)

Si aún no lo ha hecho, puede descargar éste ejemplo y otros usados en este libro.

```
#stdout.py
import sys

print 'Dive in'                                                    ❶
saveout = sys.stdout                                              ❷
fsock = open('out.log', 'w')                                      ❸
sys.stdout = fsock                                                ❹
print 'This message will be logged instead of displayed'          ❺
sys.stdout = saveout                                              ❻
fsock.close()                                                     ❼
```

❶ Esto imprimirá en la "Ventana interactiva" del IDE (o la terminal, si ejecuta el *script* desde la línea de órdenes).

❷ Garde siempre `stdout` antes de redireccionarlo, para que pueda devolverlo luego a la normalidad.

❸ Abrimos un fichero para escribir. El fichero será creado en caso de no existir. Si existe quedará sobrescrito.

❹ Redirige la saliad futura al nuevo fichero acabado de crear.

❺ Esto será "impreso" sólo en el fichero de registro; no será visible en la ventana del IDE o en la pantalla.

❻ Deja `stdout` tal como estaba antes de que jugase con él.

❼ Cierre el fichero de registro.

La redirección de `stderr` funciona exactamente igual, usando `sys.stderr` en lugar de `sys.stdout`.

Ejemplo 10.10. Redirección de información de error

```
[usted@localhost kgp]$ python stderr.py
[usted@localhost kgp]$ cat error.log
Traceback (most recent line last):
  File "stderr.py", line 5, in ?
    raise Exception, 'this error will be logged'
Exception: this error will be logged
```

Si aún no lo ha hecho, puede descargar éste ejemplo y otros usados en este libro.

```
#stderr.py
import sys

fsock = open('error.log', 'w')              ❶
sys.stderr = fsock                          ❷
raise Exception, 'this error will be logged' ❸ ❹
```

❶ Abra el fichero de registro donde quiera almacenar la información de depuración.

❷ Redirija la salida de error asignándole el objeto del fichero acabado de crear a `stderr`.

❸ Lance una excepción. Observe por la salida de pantalla que esto *no* imprime nada. Toda la información normal del volcado de pila se ha escrito en `error.log`.

❹ Advierta también que no está cerrando de forma explícita el fichero de registro, ni poniendo en `stderr` su valor original. No pasa nada, ya que una vez que el programa aborte (debido a la excepción), Python hará limpieza y cerrará el fichero por nosotros, y no afecta para nada que no restauremos el valor de `stderr` ya que, como mencioné, el programa aborta y Python se detiene. Restaurar el original es importante con `stdout` si espera hacer cosas más adelante dentro del mismo *script*.

Dado que es tan común escribir los mensajes de error a la salida de errores, hay una sintaxis abreviada que puede usar en lugar de tomarse las molestias de hacer la redirección explícita.

Ejemplo 10.11. Imprimir en `stderr`

```
>>> print 'entering function'
entering function
>>> import sys
>>> print >> sys.stderr, 'entering function' ❶
entering function
```

❶ Esta sintaxis abreviada de la sentencia `print` se puede usar para escribir en cualquier objeto de fichero o de tipo fichero que haya abierto. En este caso

puede redirigir una sola sentencia `print` a `stderr` sin afectar las siguientes sentencias `print`.

La entrada estándar, por otro lado, es un objeto de fichero de sólo lectura, y representa los datos que fluyen dentro del programa desde alguno previo. Esto no tendrá mucho sentido para los usuarios del Mac OS clásico, o incluso de Windows que no se manejen de forma fluida en la línea de órdenes de MS-DOS. La manera en que trabaja es que se pueden construir cadenas de órdenes en una única línea, de manera que la salida de un programa se convierte en la entrada del siguiente en la cadena. El primer programa se limita a imprimir en la salida estándar (sin hacer ningún tipo de redirección explícita, sólo ejecuta sentencias `print` normales, o lo que sea), y el siguiente programa lee de la entrada estándar, y es el sistema operativo el que se encarga de conectar la salida de un programa con la entrada del siguiente.

Ejemplo 10.12. Encadenamiento de órdenes

```
[usted@localhost kgp]$ python kgp.py -g binary.xml          ❶
01100111
[usted@localhost kgp]$ cat binary.xml                       ❷
<?xml version="1.0"?>
<!DOCTYPE grammar PUBLIC "-//diveintopython.org//DTD Kant Generator
Pro v1.0//EN" "kgp.dtd">
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
</grammar>
[usted@localhost kgp]$ cat binary.xml | python kgp.py -g -  ❸ ❹
10110001
```

❶ Como vio en Sección 9.1, “Inmersión”, esto imprimirá unacadenade ocho bits al azar, `0` o `1`.

❷ Esto imprime simplemente el contenido entero de `binary.xml`. (Los usuarios de Windows deberían usar `type` en lugar de `cat`).

❸ Esto imprime el contenido de `binary.xml`, pero el carácter "`|`", denominado carácter de "tubería", implica que el contenido no se imprimirá en la pantalla. En su lugar se convertirá en la entrada estándar de la siguiente orden, que en este caso llama al *script* de Python.

❹ En lugar de especificar un módulo (como `binary.xml`), especificamos "`-`", que hace que el *script* carge la gramática desde la entrada estándar en lugar de un fichero específico en el disco (más sobre esto en el siguiente ejemplo). Así que el efecto es el mismo que en la primera sintaxis, donde especificábamos directamente el nombre de fichero de la gramática, pero piense el aumento de posibilidades aquí. En lugar de hacer simplemente on possibilities here. Instead of simply doing `cat binary.xml`, podría ejecutar un *script* que genere la gramática de forma dinámica, y entonces pasarlo a otro *script* mediante una tubería. Puede venir de cualquier parte: una base de datos o algún tipo de meta-*script* generador de gramáticas, o lo que sea. La idea es que no hace falta cambiar el *script* `kgp.py` para incorporar esta funcionalidad. Todo lo que necesita es la capacidad de obtener todos los ficheros de gramática por la entrada estándar, y así podrá desplazar toda la lógica a otro programa.

Entonces, ¿cómo "sabe" el *script* que ha de leer de la entrada estándar cuando el fichero de la gramática es "`-`"? No es magia; es simple código.

So how does the script "know" to read from standard input when the grammar file is "`-`"? It's not magic; it's just code.

Ejemplo 10.13. Lectura de la entrada estándar con `kgp.py`

```
def openAnything(source):
    if source == "-":    ❶
        import sys
        return sys.stdin
```

```
    # try to open with urllib (if source is http, ftp, or file URL)
    import urllib
    try:

[... corte ...]
```

❶ Ésta es la función `openAnything` de `toolbox.py`, que examinamos antes en Sección 10.1, "Abstracción de fuentes de datos". Todo lo que hemos hecho es añadir tres líneas de código al principio de la función para comprobar si la fuente es "`-`"; y en ese caso devolvemos `sys.stdin`. ¡Sí, eso es todo! Recuerde, `stdin` es un objeto de tipo fichero con método `read`, así que el resto del código (en `kgp.py`, donde llama a `openAnything`) no cambia un ápice.

Footnotes

[13]

10.3. Caché de búsqueda de nodos

`kgp.py` emplea varios trucos que pueden o no serle útiles en el procesamiento de XML. El primero aprovecha la estructura consistente de los documentos de entrada para construir una caché de nodos.

Un fichero de gramática define una serie de elementos `ref`. Cada `ref` contiene uno o más elementos `p`, que pueden contener un montón de cosas diferentes, incluyendo `xref`s. Cada vez que encuentras un `xref`, se busca el elemento `ref` correspondiente con el mismo atributo `id`, se escoge uno de los hijos del elemento `ref` y se analiza (veremos cómo se hace esta elección al azar en la siguiente sección).

Así es como se construye la gramática: definimos elementos `ref` para las partes más pequeñas, luego elementos `ref` que "incluyen" el primer elemento `ref` usando `xref`, *etc.*. Entonces se analiza la referencia "más grande" y se sigue cada `xref` hasta sacar texto real. El texto de la salida depende de las decisiones (al

azar) que se tomen cada vez que seguimos un `xref`, así que la salida es diferente cada vez.

Todo esto es muy flexible, pero hay un inconveniente: el rendimiento. Cuando se encuentra un `xref` y necesitamos encontrar el elemento `ref` correspondiente, tenemos un problema. El `xref` tiene un atributo `id`, y queremos encontrar el elemento `xref` que tiene el mismo atributo `id` pero no hay manera sencilla de acceder a él. La manera lenta sería obtener la lista completa de elementos `ref` cada vez, y luego iterar manualmente mirando el atributo `id` de cada uno. La manera rápida es hacer eso una vez y construir una caché en un diccionario.

Ejemplo 10.14. `loadGrammar`

```
    def loadGrammar(self, grammar):
        self.grammar = self._load(grammar)
        self.refs = {}                                              ❶
        for ref in self.grammar.getElementsByTagName("ref"): ❷
            self.refs[ref.attributes["id"].value] = ref      ❸ ❹
```

❶ Empiece creando un diccionario vacío, `self.refs`.

❷ Como vio en Sección 9.5, "Búsqueda de elementos", `getElementsByTagName` devuelve una lista de todos los elementos con un nombre en particular. Podemos obtener fácilmente una lista de todos los elementos `ref` y luego iterar sobre esa lista.

❸ Como vio en Sección 9.6, "Acceso a atributos de elementos", podemos acceder a atributos individuales de un elemento por su nombre, usando la sintaxis estándar de un diccionario. Así que las claves del diccionario `self.refs` serán los valores del atributo `id` de cada elemento `ref`.

❹ Los valores del diccionario `self.refs` serán los elemento `ref` en sí. Como vio en Sección 9.3, "Análisis de XML", cada elemento, cada nodo, cada comentario, cada porción de texto de un documento XML analizado es un objeto.

Una vez construya esta caché, cada vez que encuentre un `xref` y necesite buscar el elemento `xref` con el mismo atributo `id`, simplemente puede buscarlo en `self.refs`.

Ejemplo 10.15. Uso de la caché de elementos `ref`

```
    def do_xref(self, node):
        id = node.attributes["id"].value
        self.parse(self.randomChildElement(self.refs[id]))
```

Exploraremos la función `randomChildElement` en la siguiente sección.

10.4. Encontrar hijos directos de un nodo

Otra técnica útil cuando se analizan documentos XML es encontrar todos los elementos que sean hijos directos de un elemento en particular. Por ejemplo, en los ficheros de gramáticas un elemento `ref` puede tener varios elementos `p`, cada uno de los cuales puede contener muchas cosas, incluyendo otros elementos `p`. Queremos encontrar sólo los elementos `p` que son hijos de `ref`, no elementos `p` que son hijos de otros elementos `p`.

Quizá piense que podría simplemente usar `getElementsByTagName` para esto, pero no puede. `getElementsByTagName` busca de forma recursiva y devuelve una lista con todos los elementos que encuentra. Como los elementos `p` pueden contener otros `p`, no podemos usar `getElementsByTagName` porque podría devolver elementos `p` anidados que no queremos. Para encontrar sólo elementos hijos directos tendremos que hacerlo por nosotros mismos.

Ejemplo 10.16. Búsqueda de elementos hijos directos

```
    def randomChildElement(self, node):
        choices = [e for e in node.childNodes
                   if e.nodeType == e.ELEMENT_NODE] ❶ ❷ ❸
        chosen = random.choice(choices)             ❹
        return chosen
```

❶ Como vio en Ejemplo 9.9, “Obtener nodos hijos”, el atributo `childNodes`

devuelve una lista de todos los nodos hijos de un elemento.

❷ Sin embargo, como vio en Ejemplo 9.11, "Los nodos hijos pueden ser un texto", la lista que devuelve `childNodes` contiene todos los tipos diferentes de nodos, incluidos nodos de texto. Esto no es lo que queremos. Sólo queremos los hijos que sean elementos.

❸ Cada nodo tiene un atributo `nodeType` que puede ser `ELEMENT_NODE`, `TEXT_NODE`, `COMMENT_NODE` y otros valores. La lista completa de posibles valores está en el fichero `__init__.py` del paquete `xml.dom` (en Sección 9.2, "Paquetes" hablamos de los paquetes). Pero sólo nos interesan los nodos que son elementos, así que podemos filtrar la lista para incluir sólo los nodos cuyo `nodeType` es `ELEMENT_NODE`.

❹ Una vez tenemos la lista de los elementos, escoger uno al azar es sencillo. Python viene con un módulo llamado `random` que incluye varias funciones útiles. La función `random.choice` toma una lista de cualquier número de elementos y devuelve uno al azar. Por ejemplo, si los elementos `ref` contienen varios elementos `p` entonces `choices` podría ser una lista de elementos `p` y `chosen` terminaría siendo uno de ellos, escogido al azar.

10.5. Creación de manejadores diferentes por tipo de nodo

El tercer consejo útil para procesamiento de XML implica separar el código en funciones lógicas, basándose en tipos de nodo y nombres de elemento. Los documentos XML analizados se componen de varios tipos de nodos que representa cada uno un objeto de Python. El nivel raíz del documento en sí lo representa un objeto `Document`. El `Document` contiene uno o más objetos `Element` (por las etiquetas XML), cada uno de los cuales contiene otros objetos `Element`, `Text` (para zonas de texto) o `Comment` (para los comentarios). Python hace sencillo escribir algo que separe la lógica por cada tipo de nodo.

Ejemplo 10.17. Nombres de clases de objetos XML analizados

```
>>> from xml.dom import minidom
>>> xmldoc = minidom.parse('kant.xml') ❶
>>> xmldoc
<xml.dom.minidom.Document instance at 0x01359DE8>
>>> xmldoc.__class__                    ❷
<class xml.dom.minidom.Document at 0x01105D40>
>>> xmldoc.__class__.__name__           ❸
'Document'
```

❶ Asuma por un momento que `kant.xml` está en el directorio actual.

❷ Como vio en Sección 9.2, "Paquetes", el objeto devuelto al analizar un documento XML es un `Document`, tal como se define en `minidom.py` en el paquete `xml.dom`. Como vio en Sección 5.4, "Instanciación de clases", `__class__` es un atributo que incorpora todo objeto de Python.

❸ Es más, `__name__` es un atributo que incorpora toda clase de Python, y ésta es su cadena. Esta cadena no tiene nada de misterioso; es el mismo nombre de la clase que escribe cuando define una clase usted mismo (vea Sección 5.3, "Definición de clases").

Bien, ahora podemos obtener el nombre de la clase de cualquier nodo XML (ya que cada nodo se representa con un objeto de Python). ¿Cómo puede aprovechar esto para separar la lógica al analizar cada tipo de nodo? La respuesta es `getattr`, que ya vio en Sección 4.4, "Obtención de referencias a objetos con getattr".

Ejemplo 10.18. `parse`, un despachador XML genérico

```
    def parse(self, node):
        parseMethod = getattr(self, "parse_%s" %
node.__class__.__name__) ❶ ❷
        parseMethod(node) ❸
```

❶ Antes de nada, advierta que está creando una cadena más grande basándose en el nombre de la clase del nodo que le pasaron (en el argumento `node`). Así que si le pasan un nodo `Document` estará creando la cadena `'parse_Document'`, *etc.*.

❷ Ahora puede tratar la cadena como el nombre de una función y obtener una referencia a la función en sí usando `getattr`

❸ Por último, podemos invocar la función y pasarle el nodo como argumento. El siguiente ejemplo muestra las definiciones de cada una de estas funciones.

Ejemplo 10.19. Funciones invocadas por `parse`

```
def parse_Document(self, node): ❶
    self.parse(node.documentElement)

def parse_Text(self, node):    ❷
    text = node.data
    if self.capitalizeNextWord:
        self.pieces.append(text[0].upper())
        self.pieces.append(text[1:])
        self.capitalizeNextWord = 0
    else:
        self.pieces.append(text)

def parse_Comment(self, node): ❸
    pass

def parse_Element(self, node): ❹
    handlerMethod = getattr(self, "do_%s" % node.tagName)
    handlerMethod(node)
```

❶ `parse_Document` sólo se invoca una vez, ya que sólo hay un nodo `Document` en un documento XML, y sólo un objeto `Document` en la representación analizada del XML. Simplemente analiza el elemento raíz del fichero de gramática.

❷ `parse_Text` se invoca sobre los nodos que representan texto. La función en sí hace algo de procesamiento especial para poner en mayúsculas de forma automática la primera palabra de una frase, pero aparte de eso sólo añade el texto representado a la lista.

❸ `parse_Comment` es un simple `pass`, ya que no nos interesan los comentarios que hay en los ficheros de gramáticas. Observe sin embargo que aún debemos definir la función e indicar explícitamente que no haga nada. Si la

función no existiese la `parse` genérica fallaría tan pronto como se topase con un comentario, porque intentaría encontrar la inexistente función `parse_Comment`. Definir una función por cada tipo de nodo, incluso los que no usamos, permite que la función genérica `parse` sea sencilla y tonta.

4. El método `parse_Element` es en sí mismo un despachador, que se basa en el nombre de la etiqueta del elemento. La idea básica es la misma: tomar lo que distingue a los elementos unos de otros (el nombre de sus etiquetas) y despacharlos a funciones diferentes. Construimos una cadena como `'do_xref'` (para una etiqueta `<xref>`), encontramos una función con ese nombre, y la invocamos. Y así con cada uno de los otros nombres de etiquetas que podrían encontrarse durante el análisis de un fichero de gramática (etiquetas `<p>`, `<choice>`).

En este ejemplo, las funciones despachadoras `parse` y `parse_Element` encuentras métodos en su misma clase. Si el procesamiento se vuelve complejo (o tenemos muchos tipos diferentes de etiquetas), podríamos dividir el código en módulos separados y usar importación dinámica para llamar a las funciones necesarias dentro de sus módulos. La importación dinámica la expondremos en Capítulo 16, *Programación Funcional*.

10.6. Tratamiento de los argumentos en línea de órdenes

Python admite lacreación de programas que se pueden ejecutar desde la línea de órdenes, junto con argumentos y opciones tanto de estilo corto como largo para especificar varias opciones. Nada de esto es específico al XML, pero este *script* hace buen uso del tratamiento de la línea de órdenes, así que parece buena idea mencionarlo.

Es complicado hablar del procesamiento de la línea de órdenes sin entender cómo es expuesta a un programa en Python, así que escribiremos un programa sencillo para verlo.

Ejemplo 10.20. Presentación de `sys.argv`

Si aún no lo ha hecho, puede descargar éste ejemplo y otros usados en este libro.

```
#argecho.py
import sys

for arg in sys.argv: ❶
    print arg
```

❶ Cada argumento de la línea de órdenes que se pase al programa estará en `sys.argv`, que es una lista. Aquí estamos imprimiendo cada argumento en una línea aparte.

Ejemplo 10.21. El cotenido de `sys.argv`

```
[usted@localhost py]$ python argecho.py             ❶
argecho.py
[usted@localhost py]$ python argecho.py abc def     ❷
argecho.py
abc
def
[usted@localhost py]$ python argecho.py --help      ❸
argecho.py
--help
[usted@localhost py]$ python argecho.py -m kant.xml ❹
argecho.py
-m
kant.xml
```

❶ La primera cosa que hay de saber sobre `sys.argv` es que contiene el nombre del *script* que se ha ejecutado. Aprovecharemos este conocimiento más adelante, en Capítulo 16, *Programación Funcional*. No se preocupe de eso por ahora.

❷ Los argumentos en la línea de órdenes se separan con espacios, y cada uno aparece como un elemento separado en la lista `sys.argv`.

❸ Las opciones como `--help` también aparecen como un elemento aparte en la

lista `sys.argv`.

❹ Para hacer las cosas incluso más interesantes, algunas opciones pueden tomar argumentos propios. Por ejemplo, aquí hay una opción (`-m`) que toma un argumento (`kant.xml`). Tanto la opción como su argumento son simples elementos secuenciales en la lista `sys.argv`. No se hace ningún intento de asociar uno con otro; todo lo que tenemos es una lista.

Así que como puede ver, ciertamente tenemos toda la información que se nos pasó en la línea de órdenes, pero sin embargo no parece que vaya a ser tan fácil hacer uso de ella. Para programas sencillos que sólo tomen un argumento y no tengan opciones, podemos usar simplemente `sys.argv[1]` para acceder a él. No hay que avergonzarse de esto; yo mismo lo hago a menudo. Para programas más complejos necesitará el módulo `getopt`.

Ejemplo 10.22. Presentación de `getopt`

```
def main(argv):
    grammar = "kant.xml"                    ❶
    try:
        opts, args = getopt.getopt(argv, "hg:d", ["help", "grammar="])
❷
    except getopt.GetoptError:              ❸
        usage()                             ❹
        sys.exit(2)

...

if __name__ == "__main__":
    main(sys.argv[1:])
```

❶ Antes de nada, mire al final del ejemplo y advierta que está llamando a la función `main` con `sys.argv[1:]`. Recuerde, `sys.argv[0]` es el nombre del *script* que está ejecutándose; no nos interesa eso a la hora de procesar la línea de órdenes, así que lo eliminamos y pasamos el resto de la lista.

❷ Aquí es donde sucede todo el procesamiento interesante. La función `getopt`

del módulo `getopt` toma tres parámetros: la lista de argumentos (que conseguimos de `sys.argv[1:]`), una cadena que contiene todas las posibles opciones de un solo carácter que acepta este programa, y una lista de opciones más largas que son equivalentes a las versiones de un solo carácter. Esto es bastante confuso a primera vista, y se explicará con más detalle enseguida.

❸ Si algo va mal al intentar analizar estas opciones de la línea de órdenes `getopt` emitirá una excepción, que capturamos. Le dijimos a `getopt` todas las opciones que entedemos, así que probablemente esto significa que el usuario pasó alguna opción que no entendemos.

❹ Como es práctica estándar en el mundo de UNIX, cuando se le pasan opciones que no entiende, el *script* muestra un resumen de su uso adecuado y sale de forma controlada. Fíjese en que no he enseñado aquí la función `usage`. Tendrá que programarla en algún lado y hacer que imprima el resumen apropiado; no es automático.

Así que, ¿qué son todos esos parámetros que le pasamos a la función `getopt`? Bien, el primero es simplemente la lista sin procesar de opciones y argumentos de la línea de órdenes (sin incluir el primer elemento, el nombre del *script*, que ya eliminamos antes de llamar a la función `main`). El segundo es la lista de opciones cortas que acepta el *script*.

```
"hg:d"

-h
        print usage summary
-g ...
        use specified grammar file or URL
-d
        show debugging information while parsing
```

La primera opción y la tercera son autosuficientes; las especificamos o no y hacen cosas (imprimir ayuda) o cambian estados (activar la depuración). Sin embargo, a la segunda opción (`-g`) *debe* seguirle un argumento, que es el

nombre del fichero de gramática del que hay que leer. De hecho puede ser un nombre de fichero o una dirección web, y aún no sabemos qué (lo averiguaremos luego), pero sabemos que debe ser *algo*. Se lo decimos a `getopt` poniendo dos puntos tras `g` en el segundo parámetro de la función `getopt`.

Para complicar más las cosas, el *script* acepta tanto opciones cortas (igual que `-h`) como opciones largas (igual que `--help`), y queremos que hagan lo mismo. Para esto es el tercer parámetro de `getopt`, para especificar una lista de opciones largas que corresponden a las cortas que especificamos en el segundo parámetro.

```
["help", "grammar="]
```

```
--help
        print usage summary
--grammar ...
        use specified grammar file or URL
```

Three things of note here:

1. Todas las opciones largas van precedidas de dos guiones en la línea de órdenes, pero no incluimos esos guiones al llamar a `getopt`. Se sobreentienden.
2. La opción `--grammar` siempre debe ir seguida por un argumento adicional, igual que la `-g`. Esto se indica con un signo igual, `"grammar="`.
3. La lista de opciones largas es más corta que la de las cortas, porque la opción `-d`no tiene una versión larga correspondiente. Esto es correcto; sólo `-d` activará la depuración. Pero el orden de las opciones cortas y largas debe ser el mismo, así que necesitará espeficar todas las opciones cortas que *tienen* su versión larga correspondiente primero, y luego el resto de opciones cortas.

¿Sigue confundido? Miremos el código real y veamos si tiene sentido en ese contexto.

Ejemplo 10.23. Tratamiento de los argumentos de la línea de órdenes en `kgp.py`

```
def main(argv):                                    ❶
    grammar = "kant.xml"
    try:
        opts, args = getopt.getopt(argv, "hg:d", ["help", "grammar="])
    except getopt.GetoptError:
        usage()
        sys.exit(2)
    for opt, arg in opts:                          ❷
        if opt in ("-h", "--help"):                ❸
            usage()
            sys.exit()
        elif opt == '-d':                          ❹
            global _debug
            _debug = 1
        elif opt in ("-g", "--grammar"):           ❺
            grammar = arg

    source = "".join(args)                         ❻

    k = KantGenerator(grammar, source)
    print k.output()
```

❶ La variable `grammar` contendrá el nombre del fichero de gramática que estemos usando. La inicializamos aquí en cado de que no se especifique en la línea de órdenes (usando la opción `-g` o `--grammar`).

❷ La variable `opts` que obtenemos de `getopt` contiene una lista de tuplas `opción` y `argumento`. Si la opción no toma argumentos, entonces `arg` será `None`. Esto hace más sencillo iterar sobre las opciones.

❸ `getopt` valida que las opciones de la línea de órdenes sean aceptables, pero no hace ningún tipo de conversión entre cortas y largas. Si especificamos la opción `-h`, `opt` contendrá `"-h"`; si especificamos `--help`, `opt` contendrá `"--help"`. Así que necesitamos comprobar ambas.

❹ Recuerde que la opción `-d` no tiene la opción larga correspondiente, así que

sólo nos hace falta comprobar la corta. Si la encontramos, damos valor a una variable global a la que nos referiremos luego para imprimir información de depuración. (Usé esto durante el desarrollo del *script*. ¿Qué, pensaba que todos estos ejemplos funcionaron a la primera?)

5. Si encuentra un fichero de gramática, con la opción `-g` o con `--grammar`, guardamos el argumento que le sigue (almacenado en `arg`) dentro de la variable `grammar` sustituyendo el valor por omisión que inicializamos al principio de la función `main`.

6. Esto es todo. Hemos iterado y tenido en cuenta todas las opciones de línea de órdenes. Eso significa que si queda algo deben ser argumentos. Ese resto lo devuelve la función `getopt` en la variable `args`. En este caso, estamos tratándolo como material fuente para el analizador. Si no se especifican argumentos en la línea de órdenes `args` será una lista vacía, y `source` una cadena vacía.

10.7. Todo junto

A cubierto mucho terreno. Volvamos atrás y comprobemos cómo se unen todas las piezas.

Para empezar, este es un *script* que toma argumentos desde la línea de órdenes usando el módulo `getopt`.

```
def main(argv):
...
    try:
        opts, args = getopt.getopt(argv, "hg:d", ["help", "grammar="])
    except getopt.GetoptError:
...
    for opt, arg in opts:
...
```

Creamos una nueva instancia de la clase `KantGenerator` y le pasamos el fichero de gramática y una fuente que pueden haber sido especificados desde la línea de órdenes, o no.

```
    k = KantGenerator(grammar, source)
```

La instancia de `KantGenerator` carga automáticamente la gramática, que está en un fichero XML. Usamos la función `openAnything` modificada para abrir el fichero (que puede estar almacenado en un fichero local o en un servidor web remoto), luego usa las funciones de análisis que incorpora `minidom` para traducir el XML a un árbol de objetos de Python.

```
    def _load(self, source):
        sock = toolbox.openAnything(source)
        xmldoc = minidom.parse(sock).documentElement
        sock.close()
```

Oh, y ya que estamos, aprovechamos nuestro conocimiento de la estructura del documentoXML para construir una pequeña caché de referencias que no son sino elementos del documento XML.

```
    def loadGrammar(self, grammar):
        for ref in self.grammar.getElementsByTagName("ref"):
            self.refs[ref.attributes["id"].value] = ref
```

Si especificamos algún material fuente en la línea de órdenes, lo usaremos; en caso contrario buscamos en la gramática la referencia de "nivel más alto" (que no está referenciada por ninguna otra) y la usamos como punto de partida.

```
    def getDefaultSource(self):
        xrefs = {}
        for xref in self.grammar.getElementsByTagName("xref"):
            xrefs[xref.attributes["id"].value] = 1
        xrefs = xrefs.keys()
        standaloneXrefs = [e for e in self.refs.keys() if e not in
xrefs]
        return '<xref id="%s"/>' % random.choice(standaloneXrefs)
```

Ahora exploramos el material fuente. Este material también es XML y lo analizamos nodo por nodo. Para que el código sea desacoplado y de fácil mantenimiento, usamos manejadores diferentes por cada tipo de nodo.

```
    def parse_Element(self, node):
        handlerMethod = getattr(self, "do_%s" % node.tagName)
        handlerMethod(node)
```

Nos movemos por la gramática analizando todos los hijos de cada elemento `p`,

```
    def do_p(self, node):
...
        if doit:
            for child in node.childNodes: self.parse(child)
```

sustituyendo elementos `choice` con un hijo al azar,

```
    def do_choice(self, node):
        self.parse(self.randomChildElement(node))
```

y los elementos `xref` con un hijo al azar del elemento `ref` correspondiente, que pusimos anteriormente en la caché.

```
    def do_xref(self, node):
        id = node.attributes["id"].value
        self.parse(self.randomChildElement(self.refs[id]))
```

Con el tiempo el análisis llegará al texto plano,

```
    def parse_Text(self, node):
        text = node.data
...
            self.pieces.append(text)
```

que imprimiremos.

```
def main(argv):
...
    k = KantGenerator(grammar, source)
    print k.output()
```

10.8. Resumen

Python incluye bibliotecas potentes para el análisis y la manipulación de documentos XML. `minidom` toma un fichero XML y lo convierte en objetos de Python, proporcionando acceso aleatorio a elementos arbitrarios. Aún más, este capítulo muestra cómo se puede usar Python para crear un *script* de línea de órdenes, completo con sus opciones, argumentos, gestión de errores, e incluso la capacidad de tomar como entrada el resultado de un programa anterior mediante una tubería.

Antes de pasar al siguiente capítulo debería sentirse cómodo haciendo las siguientes cosas:

- Encadenar programas con entrada y salida estándar
- Definir directores dinámicos usando `getattr`.
- Usar opciones de línea de órdenes y validarlos usando `getopt`

Capítulo 11. Servicios Web HTTP

11.1. Inmersión

Hemos aprendido cosas sobre procesamiento de HTML y de XML, y por el camino también vio cómo descargar una página web y analizar XML de una URL, pero profundicemos algo más en el tema general de los servicios web HTTP.

Dicho sencillamente, los servicios web HTTP son formas programáticas de enviar y recibir datos a/desde servidores remotos usando directamente operaciones HTTP. Si queremos obtener datos del servidor usaremos HTTP GET; si queremos enviar datos nuevos usaremos HTTP POST. (Algunos API más avanzados de servicios web HTTP también definen maneras para modificar y eliminar datos existentes, usando HTTP PUT y HTTP DELETE). En otras palabras, los “verbos” que incorpora el protocolo HTTP (GET, POST, PUT

y DELETE) se relacionan directamente con las operaciones de las aplicaciones para recibir, enviar, modificar y borrar datos.

La ventaja principal de este enfoque es la simplicidad, y esta simplicidad ha resultado ser popular en muchos sitios diferentes. Se pueden crear y almacenar de forma estática los datos (normalmente XML), o generarlos de forma dinámica mediante un *script* en el servidor, y todos los lenguajes más usados incluyen bibliotecas HTTP para descargarlos. La depuració también es sencilla porque se puede cargar el servicio web en cualquier servidor web y ver los datos sin tratar. Los navegadores modernos incluso dan un formato de aspecto agradable a los datos XML, para permitirle navegar por ellos rápidamente.

Ejemplos de servicios web puros XML-sobre-HTTP:

- La Amazon API le permite descargar información de productos de la tienda web de Amazon.com.
- El National Weather Service (Estados Unidos) y el Hong Kong Observatory (Hong Kong) ofrecen alertas meteorológicas como servicio web.
- La Atom API para gestión de contenido basado en web.
- Los *feeds* sindicados de los weblog y sitios de noticias le traen noticias de última hora desde gran variedad de sitios.

En próximos capítulos explorará APIs que usan HTTP como transporte para enviar y recibir datos, pero no relacionará la semántica de la aplicación a la propia de HTTP (hacen todo usando HTTP POST). Pero este capítulo se concentrará en usar HTTP GET para obtener datos de un servidor remoto, y exploraremos varias características de HTTP que podemos usar para obtener el máximo beneficio de los servicios web puramente HTTP.

Aquí tiene una versión más avanzada del módulo `openanything` que vio en el capítulo anterior:

Ejemplo 11.1. `openanything.py`

Si aún no lo ha hecho, puede descargar éste ejemplo y otros usados en este libro.

```
import urllib2, urlparse, gzip
from StringIO import StringIO

USER_AGENT = 'OpenAnything/1.0
+http://diveintopython.org/http_web_services/'

class SmartRedirectHandler(urllib2.HTTPRedirectHandler):
    def http_error_301(self, req, fp, code, msg, headers):
        result = urllib2.HTTPRedirectHandler.http_error_301(
            self, req, fp, code, msg, headers)
        result.status = code
        return result

    def http_error_302(self, req, fp, code, msg, headers):
        result = urllib2.HTTPRedirectHandler.http_error_302(
            self, req, fp, code, msg, headers)
        result.status = code
        return result

class DefaultErrorHandler(urllib2.HTTPDefaultErrorHandler):
    def http_error_default(self, req, fp, code, msg, headers):
        result = urllib2.HTTPError(
            req.get_full_url(), code, msg, headers, fp)
        result.status = code
        return result

def openAnything(source, etag=None, lastmodified=None,
agent=USER_AGENT):
    '''URL, filename, or string --> stream

    This function lets you define parsers that take any input source
    (URL, pathname to local or network file, or actual data as a
string)
    and deal with it in a uniform manner.  Returned object is
guaranteed
    to have all the basic stdio read methods (read, readline,
readlines).
    Just .close() the object when you're done with it.
```

```
    If the etag argument is supplied, it will be used as the value of
an
    If-None-Match request header.

    If the lastmodified argument is supplied, it must be a formatted
    date/time string in GMT (as returned in the Last-Modified header
of
    a previous request).  The formatted date/time will be used
    as the value of an If-Modified-Since request header.

    If the agent argument is supplied, it will be used as the value of
a
    User-Agent request header.
    '''

    if hasattr(source, 'read'):
        return source

    if source == '-':
        return sys.stdin

    if urlparse.urlparse(source)[0] == 'http':
        # open URL with urllib2
        request = urllib2.Request(source)
        request.add_header('User-Agent', agent)
        if etag:
            request.add_header('If-None-Match', etag)
        if lastmodified:
            request.add_header('If-Modified-Since', lastmodified)
        request.add_header('Accept-encoding', 'gzip')
        opener = urllib2.build_opener(SmartRedirectHandler(),
DefaultErrorHandler())
        return opener.open(request)

    # try to open with native open function (if source is a filename)
    try:
        return open(source)
    except (IOError, OSError):
        pass

    # treat source as string
```

```
    return StringIO(str(source))

def fetch(source, etag=None, last_modified=None, agent=USER_AGENT):
    '''Fetch data and metadata from a URL, file, stream, or string'''
    result = {}
    f = openAnything(source, etag, last_modified, agent)
    result['data'] = f.read()
    if hasattr(f, 'headers'):
        # save ETag, if the server sent one
        result['etag'] = f.headers.get('ETag')
        # save Last-Modified header, if the server sent one
        result['lastmodified'] = f.headers.get('Last-Modified')
        if f.headers.get('content-encoding', '') == 'gzip':
            # data came back gzip-compressed, decompress it
            result['data'] =
gzip.GzipFile(fileobj=StringIO(result['data']])).read()
    if hasattr(f, 'url'):
        result['url'] = f.url
        result['status'] = 200
    if hasattr(f, 'status'):
        result['status'] = f.status
    f.close()
    return result
```

Lecturas complementarias

- Paul Prescod cree que los servichos web HTTP puros son el futuro de Internet.

11.2. Cómo no obtener datos mediante HTTP

Digamos que quiere descargar un recurso mediante HTTP, tal como un *feed* sindicado Atom. Pero no sólo queremos descargarlo una vez; queremos descargarlo una y otra vez, cada hora, para obtener las últimas noticias de un sitio que nos ofrece noticias sindicadas. Hagámoslo primero a la manera sucia y rápida, y luego veremos cómo hacerlo mejor.

Ejemplo 11.2. Descarga de una sindicación a la manera rápida y fea

```
>>> import urllib
>>> data =
urllib.urlopen('http://diveintomark.org/xml/atom.xml').read()     ❶
>>> print data
<?xml version="1.0" encoding="iso-8859-1"?>
<feed version="0.3"
  xmlns="http://purl.org/atom/ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xml:lang="en">
  <title mode="escaped">dive into mark</title>
  <link rel="alternate" type="text/html"
href="http://diveintomark.org/"/>
  <-- resto del feed omitido por brevedad -->
```

❶ En Python es increíblemente sencillo descargar cualquier cosa mediante HTTP; de hecho, se hace en una línea. El módulo `urllib` tiene la útil función `urlopen` que toma la dirección de la página que queremos, y devuelve un objeto de tipo fichero del que podemos simplemente leer con `read()` para obtener todo el contenido de la página. No puede ser más fácil.

¿Qué hay de malo en esto? Bien, para una prueba rápida o durante el desarrollo no hay nada de malo. Lo hago a menudo. Quería el contenido de del *feed* y obtuve el contenido del *feed*. La misma técnica funciona con cualquier página web. Pero una vez empezamos a pensar en términos de servicios web a los que queremos acceder con regularidad (y recuerde, dijimos que planeamos hacer esto cada hora) estamos siendo ineficientes, y rudos.

Hablemos un poco sobre las características básicas de HTTP.

11.3. Características de HTTP

Hay cinco características importantes de HTTP que debería tener en cuenta.

11.3.1. `User-Agent`

El `User-Agent` es simplemente una manera para que el cliente indique al servidor quién es cuando solicita una página web, un *feed* sindicado o cualquier tipo de servicio web sobre HTTP. Cuando el cliente solicite un recurso, siempre debería anunciar quién es, lo más específicamente que sea posible. Esto permite al administrador del servidor estar en contacto con el desarrollador del lado del usuario en caso de que algo vaya espectacularmente mal.

Python envía un `User-Agent` genérico por omisión: `Python-urllib/1.15`. En la siguiente sección veremos cómo cambiar esto a algo más específico.

11.3.2. Redirecciones

A veces se cambia de sitio un recurso. Los sitios web se reorganizan, las páginas pasan a tener direcciones nuevas. Incluso se puede reorganizar un servicio web. Una sindicación en `http://example.com/index.xml` puede convertirse en `http://example.com/xml/atom.xml`. O puede que cambie un dominio completo, al expandirse y reorganizarse una organización; por ejemplo, `http://www.example.com/index.xml` podría redirigirse a `http://server-farm-1.example.com/index.xml`.

Cada vez que solicita cualquier tipo de recurso a un servidor HTTP, el servidor incluye un código de estado en su respuesta. El código `200` significa "todo normal, aquí está la página que solicitó". El código `404` significa "no encontré la página" (probablemente haya visto errores 404 al navegar por la web).

HTTP tiene dos maneras diferentes de indicar que se ha cambiado de sitio un recurso. El código de estado `302` es una *redirección temporal*; significa "ups, lo hemos movido temporalmente aquí" (y entonces da la dirección temporal en una cabecera `Location:`). El código de estado `301` es una *redirección permanente*; significa "ups, lo movimos permanentemente" (y entonces da la dirección

temporal en una cabecera `Location:`). Si recibe un código de estado `302` y una dirección nueva, la especificación de HTTP dice que debería usar la nueva dirección que le indicaron, pero la siguiente vez que acceda al mismo recurso, puede intentar de nuevo con la dirección antigua. Pero si obtiene un código `301` y una nueva dirección, se espera que la utilice siempre de ahí en adelante.

`urllib.urlopen` "seguirá" automáticamente las redirecciones cuando reciba del servidor HTTP el código apropiado, pero desafortunadamente no nos lo indica cuando lo hace. Obtendremos los datos que solicitamos, pero nunca sabremos que la biblioteca tuvo la "atención" de seguir la redirección por nosotros. Así que seguiremos usando la dirección antigua y nos redirigirán cada vez a la nueva. Así damos dos rodeos en lugar de uno: ¡no es muy eficiente! Más adelante en este capítulo verá cómo evitar esto para poder tratar adecuada y eficientemente las redirecciones permanentes.

11.3.3. `Last-Modified`/`If-Modified-Since`

Algunos datos cambian con el tiempo. La página principal de CNN.com cambia constantemente cada pocos minutos. Por otor lado, la página principal de Google.com sólo cambia una vez cada pocas semanas (cuando ponen un logotipo especial de fiesta, o anuncian un nuevo servicio). Los servicios web no son diferentes; normalmente el servidor sabe cuándo fue la última vez que cambiaron los datos solicitados, y HTTP proporciona una manera para que el servidor incluya esta fecha de última modificación junto con los datos pedidos.

Si pedimos los mismos datos una segunda vez (o tercera, o cuarta), podemos decirle al servidor la fecha de última modificación que obtuvimos la vez anterior: enviamos junto a la petición la cabecera `If-Modified-Since`, con la fecha que nos dieron la última vez. Si los datos no han cambiado desde entonces, el servidor envía un código de estado HTTP especial, `304`, que significa "estos datos no han cambiado desde la última vez que los pediste". ¿Por qué supone esto una mejora? Porque cuando el servidor envía un `304`, *no envía los datos*. Todo lo que obtenemos es el código de estado. Así que no hace

falta descargar los mismos datos una y otra vez si no han cambiado; el servidor asume que los tenemos en caché local.

Todos los navegadores web modernos admiten la comprobación de fecha de última modificación. Si entró en una página, la visitó de nuevo un día después encontrando que no había cambiado, y se preguntó por qué había cargado tan rápidamente la segunda vez, ésta podría ser la respuesta. El navegador hizo dejó el contenido de la página en una caché local la primera vez, y cuando la visitamos de nuevo el servidor envió automáticamente la última fecha de modificación que obtuvo del servidor. El servidor se limita a decir `304: Not Modified`, así que el navegador sabe que debe cargar la página de la cache. Los servicios web también pueden ser así de inteligentes.

La biblioteca de URL de Python no incorpora la verificación de última modificación pero, dado que podemos incluir cabeceras arbitrarias a cada consulta y leer las cabeceras de cada respuesta, podemos añadir esta funcionalidad nosotros mismos.

11.3.4. `ETag/If-None-Match`

Las ETags son una manera alternativa de conseguir lo mismo que la comprobación de última modificación: no descargar de nuevo datos que no han variado. Funciona así, el servidor envía algún tipo de suma de comprobación de los datos (en una cabecera `ETag`) junto con los datos solicitados. La manera en que se determina esta suma es asunto del servidor. La segunda vez que pedimos esos mismos datos, incluimos la suma ETag en una cabecera `If-None-Match:` y, si los datos no han cambiado, el servidor nos devolverá un código `304`. Igual que con la comprobación de última modificación el servidor *simplemente* envía el `304`; no envía los mismos datos una segunda vez. Al incluir la suma de comprobación ETag en la segunda consulta, le estamos diciendo al servidor que no necesita enviarlos los datos si aún coinciden con esta suma, ya que aún conservamos los de la última consulta.

La biblioteca de URL de Python no incorpora la funcionalidad de las ETag, pero veremos cómo añadirla más adelante en este capítulo.

11.3.5. Compresión

La última característica importante de HTTP es la compresión gzip. Cuando hablamos de servicios web HTTP, casi siempre hablamos de mover datos XML por los cables. XML es texto, la verdad es que bastante prolijo, y el texto se comprime bien por lo general. Cuando solicitamos un recurso mediante HTTP podemos pedirle al servidor que, si tiene datos nuevos que enviarnos, haga el favor de enviarlos en un formato comprimido. Incluimos la cabecera `Accept-encoding: gzip` en la consulta y, si el servidor admite compresión, enviará los datos comprimidos con gzip y lo indicará con una cabecera `Content-encoding: gzip`.

La biblioteca de URL de Python no admite per se la compresión `gzip`, pero podemos añadir cabeceras arbitrarias a la consulta. Y Python incluye un módulo `gzip` con funciones que podemos usar para descomprimir los datos nosotros mismos.

Advierta que nuestro pequeño *script* de una línea para descargar *feeds* sindicados no admite ninguna de estas características de HTTP. Veamos cómo mejorarlo.

11.4. Depuración de servicios web HTTP

Primero vamos a activar las características de depuración de la biblioteca de HTTP de Python y veamos qué está viajando por los cables. Esto será útil durante el capítulo según añadamos características.

Ejemplo 11.3. Depuración de HTTP

```
>>> import httplib
>>> httplib.HTTPConnection.debuglevel = 1          ❶
>>> import urllib
```

```
>>> feeddata =
urllib.urlopen('http://diveintomark.org/xml/atom.xml').read()
connect: (diveintomark.org, 80)                          ❷
send: '
GET /xml/atom.xml HTTP/1.0                               ❸
Host: diveintomark.org                                   ❹
User-agent: Python-urllib/1.15                           ❺
'
reply: 'HTTP/1.1 200 OK\r\n'                             ❻
header: Date: Wed, 14 Apr 2004 22:27:30 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Content-Type: application/atom+xml
header: Last-Modified: Wed, 14 Apr 2004 22:14:38 GMT     ❼
header: ETag: "e8284-68e0-4de30f80"                      ❽
header: Accept-Ranges: bytes
header: Content-Length: 26848
header: Connection: close
```

❶ `urllib` se apoya en otra biblioteca estándar de Python, `httplib`. Normalmente no necesitará importar directamente `httplib` (`urllib` lo hace automáticamente), pero lo haremos aquí para que pueda activar el indicador de la clase `HTTPConnection` que usa `urllib` internamente para conectar con el servidor HTTP. Esta técnica es increíblemente útil. Algunas otras bibliotecas de Python disponen de indicadores de depuración similares, pero no hay una norma particular para darles nombre o activarlas; necesitará leer la documentación de cada una para saber si está disponible una característica como ésta.

❷ Ahora que está activo el indicador de depuración se imprimirá información sobre la consulta HTTP y su respuesta en tiempo real. La primera cosa que nos dice es que estamos conectando al servidor `diveintomark.org` en el puerto 80, que es el estándar de HTTP.

❸ Cuando solicitamos el *feed* Atmp `urllib` envía tres líneas al servidor. La primera especifica el verbo HTTP que estamos usando, y la ruta del recurso (sin el nombre de dominio). Todas las consultas de este capítulo usarán `GET`, pero en el próximo capítulo sobre SOAP verá que usa `POST` para todo. La sintaxis básica es la misma, independientemente del verbo.

❹ La segunda línea es la cabecera `Host`, que especifica el nombre de dominio del servicio al que accedemos. Esto es importante, porque un único servidor HTTP puede albergar varios dominios diferentes. Mi servidor alberga actualmente 12 dominios; otros pueden albergar cientos e incluso miles.

❺ La tercera línea es la cabecera `User-Agent`. Lo que ve aquí es la `User-Agent` genérica que añade por omisión la biblioteca `urllib`. En la próxima sección verá cómo personalizar esto para ser más específico.

❻ El servidor responde con un código de estado y un puñado de cabeceras (y posiblemente datos, que quedarán almacenados en la variable `feeddata`). El código de estado aquí es `200`, que quiere decir "todo normal, aquí están los datos que pidió". El servidor también le dice la fecha en que respondió a la petición, algo de información sobre el servidor en sí, y el tipo de contenido de los datos que le está dando. Dependiendo de la aplicación, esto podría ser útil o no. Ciertamente es tranquilizador pensar que hemos pedido un *feed* Atom y, sorpresa, estamos obteniendo un *feed* Atom (`application/atom+xml`, que es el tipo de contenido registrado para *feeds* Atom).

❼ El servidor nos dice cuándo ha sido modificado por última vez este *feed* Atom (en este caso, hace unos 13 minutos). Puede enviar esta fecha de vuelta al servidor la siguiente vez que pida el mismo feed, y el servidor puede hacer una comprobación de última modificación.

❽ El servidor también nos dice que este *feed* Atom tiene una suma de comprobación ETag de valor `"e8284-68e0-4de30f80"`. La suma no significa nada en sí; no hay nada que podamos hacer con ella excepto enviársela al servidor la siguiente vez que pidamos el mismo *feed*. Entonces el servidor puede usarlo para decirle si los datos han cambiado o no.

11.5. Establecer el `User-Agent`

El primer paso para mejorar nuestro cliente de servicios web HTTP es identificarnos adecuadamente con `User-Agent`. Para hacerlo nos hace falta pasar de la `urllib` básica y zambullirnos en `urllib2`.

Ejemplo 11.4. Presentación de `urllib2`

```
>>> import httplib
>>> httplib.HTTPConnection.debuglevel = 1
❶
>>> import urllib2
>>> request = urllib2.Request('http://diveintomark.org/xml/atom.xml')
❷
>>> opener = urllib2.build_opener()
❸
>>> feeddata = opener.open(request).read()
❹
connect: (diveintomark.org, 80)
send: '
GET /xml/atom.xml HTTP/1.0
Host: diveintomark.org
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Wed, 14 Apr 2004 23:23:12 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Content-Type: application/atom+xml
header: Last-Modified: Wed, 14 Apr 2004 22:14:38 GMT
header: ETag: "e8284-68e0-4de30f80"
header: Accept-Ranges: bytes
header: Content-Length: 26848
header: Connection: close
```

❶ Si aún mantiene abierto el IDE de Python tras el ejemplo de la sección anterior, puede saltarse esto, ya que es para activar la depuración de HTTP de manera que pueda ver lo que se está enviando realmente, y qué viene de vuelta.

❷ Descargar un recurso HTTP con `urllib2` es un proceso de tres pasos, por buenas razones que aclararé en breve. El primer paso es crear un objeto `Request`, que toma la URL del recurso que pretendemos descargar. Tenga en cuenta que este paso no descarga nada realmente.

❸ El segundo paso es construir algo que abra la URL. Puede tomar cualquier tipo de manejador, que controlará cómo se manipulan las respuestas. Pero

también podríamos crear uno de estos "abridores" sin manejadores a medida, que es lo que estamos haciendo aquí. Veremos cómo definir y usar estos manejadores más adelante en este capítulo, cuando exploremos las redirecciones.

❹ El último paso es decirle al abridor que abra la URL usando el objeto `Request` que hemos creado. Como puede ver por toda la información de depuración que se imprime, este paso sí que descarga el recurso, y almacena los datos devueltos en `feeddata`.

Ejemplo 11.5. Añadir cabeceras con la `Request`

```
>>> request                                              ❶
<urllib2.Request instance at 0x00250AA8>
>>> request.get_full_url()
http://diveintomark.org/xml/atom.xml
>>> request.add_header('User-Agent',
...     'OpenAnything/1.0 +http://diveintopython.org/')  ❷
>>> feeddata = opener.open(request).read()               ❸
connect: (diveintomark.org, 80)
send: '
GET /xml/atom.xml HTTP/1.0
Host: diveintomark.org
User-agent: OpenAnything/1.0 +http://diveintopython.org/  ❹
'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Wed, 14 Apr 2004 23:45:17 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Content-Type: application/atom+xml
header: Last-Modified: Wed, 14 Apr 2004 22:14:38 GMT
header: ETag: "e8284-68e0-4de30f80"
header: Accept-Ranges: bytes
header: Content-Length: 26848
header: Connection: close
```

❶ Continuamos el ejemplo anterior; hemos creado un objeto `Request` con la URL a la que queremos acceder.

❷ Podemos añadir cabeceras HTTP arbitrarias a la consulta usando el método `add_header` del objeto `Request`. El primer argumento es la cabecera, el

segundo es el valor de la cabecera. La convención dicta que un `User-Agent` debería estar en este formato específico: un nombre de aplicación, seguido por una barra, seguido por un número de versión. El resto tiene forma libre, y verá muchas variaciones en el mundo real, pero en algún lado debería incluir una URL de su aplicación. El `User-Agent` se registra normalmente en el servidor junto a otros detalles de la consulta, e incluir una URL de la aplicación le permite a los administradores del servidor buscar en sus registros de acceso para contactar con usted si algo fue mal.

❸ También podemos reutilizar el objeto `opener` que creó antes, y descargará de nuevo el mismo *feed*, pero con nuestra cabecera `User-Agent` modificada.

❹ Y aquí estamos enviando nuestra propia `User-Agent`, en lugar de la genérica que Python envía por omisión. Si observa atentamente notará que hemos definido una cabecera `User-Agent`, pero que en realidad se envió una `User-agent`. ¿Ve la diferencia? `urllib2` cambió las mayúsculas para que sólo la primera letra lo sea. En realidad no importa; HTTP especifica que los nombres de los campos de la cabecera son totalmente independientes de las mayúsculas.

11.6. Tratamiento de `Last-Modified` y `ETag`

Ahora que sabe cómo añadir cabeceras HTTP personalizadas a la consulta al servicio web, veamos cómo añadir la funcionalidad de las cabeceras `Last-Modified` y `ETag`.

Estos ejemplos muestran la salida con la depuración inactiva. Si sigue teniéndola activa tras la sección anterior puede desactivarla haciendo `httplib.HTTPConnection.debuglevel = 0`. O simplemente puede dejarla activa, si le apetece.

Ejemplo 11.6. Pruebas con `Last-Modified`

```
>>> import urllib2
>>> request = urllib2.Request('http://diveintomark.org/xml/atom.xml')
```

```
>>> opener = urllib2.build_opener()
>>> firstdatastream = opener.open(request)
>>> firstdatastream.headers.dict                          ❶
{'date': 'Thu, 15 Apr 2004 20:42:41 GMT',
 'server': 'Apache/2.0.49 (Debian GNU/Linux)',
 'content-type': 'application/atom+xml',
 'last-modified': 'Thu, 15 Apr 2004 19:45:21 GMT',
 'etag': '"e842a-3e53-55d97640"',
 'content-length': '15955',
 'accept-ranges': 'bytes',
 'connection': 'close'}
>>> request.add_header('If-Modified-Since',
...     firstdatastream.headers.get('Last-Modified'))  ❷
>>> seconddatastream = opener.open(request)            ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "c:\python23\lib\urllib2.py", line 326, in open
    '_open', req)
  File "c:\python23\lib\urllib2.py", line 306, in _call_chain
    result = func(*args)
  File "c:\python23\lib\urllib2.py", line 901, in http_open
    return self.do_open(httplib.HTTP, req)
  File "c:\python23\lib\urllib2.py", line 895, in do_open
    return self.parent.error('http', req, fp, code, msg, hdrs)
  File "c:\python23\lib\urllib2.py", line 352, in error
    return self._call_chain(*args)
  File "c:\python23\lib\urllib2.py", line 306, in _call_chain
    result = func(*args)
  File "c:\python23\lib\urllib2.py", line 412, in http_error_default
    raise HTTPError(req.get_full_url(), code, msg, hdrs, fp)
urllib2.HTTPError: HTTP Error 304: Not Modified
```

❶ ¿Recuerda todas aquellas cabeceras HTTP que vio impresas cuando activó la depuración? Así es como puede obtener acceso de forma programática a ellas: `firstdatastream.headers` es un objeto que actúa como un diccionario y le permite obtener cualquiera de las cabeceras individuales devueltas por el servidor HTTP.

❷ En la segunda consulta añadimos la cabecera `If-Modified-Since` con la última fecha de modificación que nos dio la primera petición. Si los datos no han cambiado, el servidor debería devolver un código de estado `304`.

❸ Pues bien, los datos no han cambiado. Puede ver por la traza de depuración que `urllib2` lanza una excepción especial, `HTTPError`, en respuesta del código de estado `304`. Esto es un poco inusual y no ayuda demasiado. Después de todo no es un error; le indicamos específicamente al servidor que no nos enviase datos si no habían cambiado, y los datos no cambiaron así que el servidor nos dijo que no nos iba a enviar datos. Eso no es un error; es exactamente lo que esperábamos.

`urllib2` también lanza una excepción `HTTPError` por condiciones en las que sí pensaríamos como errores, tales como `404` (página no encontrada). De hecho, lanzará `HTTPError` por *cualquier* código de estado diferente a `200` (OK), `301` (redirección permanente), o `302` (redirección temporal). Sería más útil a nuestros propósitos capturar el código de estado y devolverlo, simplemente, sin lanzar excepción. Para hcerlo necesitaremos definir un manipulador de URL personalizado.

Ejemplo 11.7. Definición de manipuladores de URL

Este manipulador de URL es parte de `openanything.py`.

```
class DefaultErrorHandler(urllib2.HTTPDefaultErrorHandler):    ❶
    def http_error_default(self, req, fp, code, msg, headers): ❷
        result = urllib2.HTTPError(
            req.get_full_url(), code, msg, headers, fp)
        result.status = code                                   ❸
        return result
```

❶ `urllib2` está diseñada sobre manipuladores de URL. Cada manipulador es una clase que puede definir cualquier cantidad de métodos. Cuando sucede algo (como un error de HTTP o incluso un código `304`) `urllib2` hace introspección en la lista de manipuladores definidos a la busca de un método que lo maneje. Usamos una introspección similar en Capítulo 9, *Procesamiento de XML* para definir manipuladores para diferentes tipos de nodo, pero `urllib2` es más flexible e inspecciona todos los manipuladores que se hayan

definido para la consulta actual.

❷ `urllib2` busca entre los manipuladores definidos y llama a `http_error_default` cuando encuentra que el servidor le envía un código de estado `304`. Definiendo un manipulador personalizado podemos evitar que `urllib2` lance una excepción. En su lugar, creamos el objeto `HTTPError` pero lo devolvemos en lugar de lanzar una excepción con él.

❸ Ésta es la parte clave: antes de volver se ha de guardar el código de estado que devolvió el servidor HTTP. Esto nos permite acceder fácilmente a él desde el programa que llama.

Ejemplo 11.8. Uso de manipuladores URL personalizados

```
>>> request.headers                              ❶
{'If-modified-since': 'Thu, 15 Apr 2004 19:45:21 GMT'}
>>> import openanything
>>> opener = urllib2.build_opener(
...     openanything.DefaultErrorHandler())      ❷
>>> seconddatastream = opener.open(request)
>>> seconddatastream.status                      ❸
304
>>> seconddatastream.read()                      ❹
''
```

❶ Continuamos con el ejemplo anterior, así que aún existe el objeto `Request` y hemos añadido la cabecera `If-Modified-Since`.

❷ Ésta es la clave: ahora que hemos definido nuestro manipulador URL personal, debemos decirle a `urllib2` que lo use. ¿Recuerda cuando dije que `urllib2` dividía el proceso de acceder a un recurso HTTP en tres pasos, y por buenas razones? Aquí tiene por qué la construcción del abridor de URL tiene su propio paso: porque puede hacerlo con su propio manipulador de URL personalizado que sustituya el comportamiento por omisión de `urllib2`.

❸ Ahora podemos abrir el recurso tranquilamente, y lo que obtenemos junto a las cabeceras normales (use `seconddatastream.headers.dict` para acceder a ellas) es un objeto que contiene el código de estado de HTTP. En este caso,

como esperábamos, el estado `304`, lo que significa que estos datos no han cambiado desde la última vez que los solicitó.

❹ Observe que cuando el servidor envía un código `304` no envía los datos. Ésa es la idea: ahorrar ancho de banda al no descargar de nuevo los datos que no hayan cambiado. Así que si realmente quiere los datos, necesitará guardarlos en una caché local la primera vez que los obtiene.

El tratamiento de `ETag` funciona de la misma manera, pero en lugar de buscar `Last-Modified` y enviar `If-Modified-Since`, buscamos `ETag` y enviamos `If-None-Match`. Empecemos una sesión nueva del IDE.

Ejemplo 11.9. Soporte de `ETag`/`If-None-Match`

```
>>> import urllib2, openanything
>>> request = urllib2.Request('http://diveintomark.org/xml/atom.xml')
>>> opener = urllib2.build_opener(
...     openanything.DefaultErrorHandler())
>>> firstdatastream = opener.open(request)
>>> firstdatastream.headers.get('ETag')        ❶
'"e842a-3e53-55d97640"'
>>> firstdata = firstdatastream.read()
>>> print firstdata                            ❷
<?xml version="1.0" encoding="iso-8859-1"?>
<feed version="0.3"
  xmlns="http://purl.org/atom/ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xml:lang="en">
  <title mode="escaped">dive into mark</title>
  <link rel="alternate" type="text/html"
href="http://diveintomark.org/"/>
  <-- rest of feed omitted for brevity -->
>>> request.add_header('If-None-Match',
...     firstdatastream.headers.get('ETag'))   ❸
>>> seconddatastream = opener.open(request)
>>> seconddatastream.status                    ❹
304
>>> seconddatastream.read()                    ❺
''
```

❶ Podemos obtener la `ETag` devuelta por el servidor usando el pseudo-diccionario `firstdatastream.headers`. (¿Qué sucede si el servidor no envió una `ETag`? Que esta línea devolvería `None`.)

❷ Bien, tenemos los datos.

❸ Ahora haga la segunda llamada indicando en la cabecera `If-None-Match` la `ETag` que obtuvo de la primera llamada.

❹ La segunda llamada tiene un éxito sin aspavientos (no lanza una excepción), y de nuevo podemos ver que el servidor envió un código de estado `304`. Sabe que los datos no han cambiado basándose en la `ETag` que enviamos la segunda vez.

❺ Independientemente de si el `304` sucede debido a una comprobación de fecha `Last-Modified` o por comparación de una suma `ETag`, no obtendremos los datos junto con el `304`. Y eso es lo que se pretendía.

En estos ejemplos el servidor HTTP ha respondido tanto a la cabecera `Last-Modified` como a `ETag`, pero no todos los servidores lo hacen. Como cliente de un servicio web debería estar preparado para usar ambos, pero debe programar de forma defensiva por si se da el caso de que el servidor sólo trabaje con uno de ellos, o con ninguno.

11.7. Manejo de redirecciones

Puede admitir redirecciones permanentes y temporales usando un tipo diferente de manipulador de URL.

Primero veamos por qué es necesario registrar los manipuladores.

Ejemplo 11.10. Acceso a servicios web sin admitir redirecciones

```
>>> import urllib2, httplib
>>> httplib.HTTPConnection.debuglevel = 1          ❶
>>> request = urllib2.Request(
```

```
...     'http://diveintomark.org/redir/example301.xml') ❷
>>> opener = urllib2.build_opener()
>>> f = opener.open(request)
connect: (diveintomark.org, 80)
send: '
GET /redir/example301.xml HTTP/1.0
Host: diveintomark.org
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 301 Moved Permanently\r\n'                ❸
header: Date: Thu, 15 Apr 2004 22:06:25 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Location: http://diveintomark.org/xml/atom.xml     ❹
header: Content-Length: 338
header: Connection: close
header: Content-Type: text/html; charset=iso-8859-1
connect: (diveintomark.org, 80)
send: '
GET /xml/atom.xml HTTP/1.0                                 ❺
Host: diveintomark.org
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Thu, 15 Apr 2004 22:06:25 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Last-Modified: Thu, 15 Apr 2004 19:45:21 GMT
header: ETag: "e842a-3e53-55d97640"
header: Accept-Ranges: bytes
header: Content-Length: 15955
header: Connection: close
header: Content-Type: application/atom+xml
>>> f.url                                                  ❻
'http://diveintomark.org/xml/atom.xml'
>>> f.headers.dict
{'content-length': '15955',
'accept-ranges': 'bytes',
'server': 'Apache/2.0.49 (Debian GNU/Linux)',
'last-modified': 'Thu, 15 Apr 2004 19:45:21 GMT',
'connection': 'close',
'etag': '"e842a-3e53-55d97640"',
'date': 'Thu, 15 Apr 2004 22:06:25 GMT',
'content-type': 'application/atom+xml'}
```

```
>>> f.status
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: addinfourl instance has no attribute 'status'
```

❶ Será capaz de ver mejor lo que sucede si activa la depuración.

❷ Esta URL la he dispuesto con una redirección permanente a mi *feed* Atom en `http://diveintomark.org/xml/atom.xml`.

❸ Exacto, cuando intenta descargar los datos de esa dirección, el servidor envía un código de estado `301` diciéndonos que se movió el recurso de forma permanente.

❹ El servidor también responde con una cabecera `Location:` que indica el lugar de la nueva dirección de estos datos.

❺ `urllib2` lee el código de estado e intenta automáticamente descargar los datos de la nueva dirección especificada en la cabecera `Location:`.

❻ El objeto que nos devuelve el `opener` contiene la nueva dirección permanente y todas la cabeceras de la segunda respuesta (obtenidas de la nueva dirección permanente). Pero no está el código de estado, así que no tenemos manera programática de saber si esta redirección era temporal o permanente. Y eso importa mucho: si es temporal entonces podemos continuar pidiendo los datos en el sitio antiguo. Pero si fuera una redirección permanente (como así es), deberíamos buscar los datos en el nuevo sitio desde ahora.

Esto no es óptimo, pero sencillo de arreglar. `urllib2` no se comporta exactamente como querríamos cuando se encuentra un `301` o un `302`, así que tendremos que cambiar su comportamiento. ¿Cómo? Con un manipulador URL personalizado, exactamente igual que hicimos para trabajar con los códigos `304`.

Ejemplo 11.11. Definición del manipulador de redirección

Esta clase está definida en `openanything.py`.

```
class SmartRedirectHandler(urllib2.HTTPRedirectHandler):     ❶
    def http_error_301(self, req, fp, code, msg, headers):
```

```
        result = urllib2.HTTPRedirectHandler.http_error_301( ❷
            self, req, fp, code, msg, headers)
        result.status = code                                 ❸
        return result

    def http_error_302(self, req, fp, code, msg, headers):   ❹
        result = urllib2.HTTPRedirectHandler.http_error_302(
            self, req, fp, code, msg, headers)
        result.status = code
        return result
```

❶ El comportamiento de redirección lo define en `urllib2` una clase llamada `HTTPRedirectHandler`. No queremos sustituirlo completamente, sólo extenderlo un poco, así que derivaremos `HTTPRedirectHandler` de manera que podamos llamar a la clase ancestro para que haga todo el trabajo duro.

❷ Cuando encuentra un código de estado `301` del servidor, `urllib2` buscará entre sus manipuladores y llamará al método `http_error_301`. Lo primero que hace el nuestro es llamar al método `http_error_301` del ancestro, que hace todo el trabajo duro de buscar la cabecera `Location:` y seguir la redirección a la nueva dirección.

❸ Aquí está lo importante: antes de volver almacenamos el código de estado (`301`), de manera que pueda acceder a él después el programa que hizo la llamada.

❹ Las redirecciones temporales (código `302`) funcionan de la misma manera: sustituimos el método `http_error_302`, llamamos al ancestro y guardamos el código de estado antes de volver.

¿Qué hemos conseguido con esto? Ahora podemos crear un abridor de URL con el manipulador personalizado de redirecciones, y seguirá accediendo automáticamente a la nueva dirección, pero ahora también podemos averiguar el código de estado de la redirección.

Ejemplo 11.12. Uso del manejador de redirección para detectar redirecciones permanentes

```
>>> request =
urllib2.Request('http://diveintomark.org/redir/example301.xml')
>>> import openanything, httplib
>>> httplib.HTTPConnection.debuglevel = 1
>>> opener = urllib2.build_opener(
...     openanything.SmartRedirectHandler())          ❶
>>> f = opener.open(request)
connect: (diveintomark.org, 80)
send: 'GET /redir/example301.xml HTTP/1.0
Host: diveintomark.org
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 301 Moved Permanently\r\n'           ❷
header: Date: Thu, 15 Apr 2004 22:13:21 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Location: http://diveintomark.org/xml/atom.xml
header: Content-Length: 338
header: Connection: close
header: Content-Type: text/html; charset=iso-8859-1
connect: (diveintomark.org, 80)
send: '
GET /xml/atom.xml HTTP/1.0
Host: diveintomark.org
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Thu, 15 Apr 2004 22:13:21 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Last-Modified: Thu, 15 Apr 2004 19:45:21 GMT
header: ETag: "e842a-3e53-55d97640"
header: Accept-Ranges: bytes
header: Content-Length: 15955
header: Connection: close
header: Content-Type: application/atom+xml

>>> f.status                                          ❸
301
>>> f.url
'http://diveintomark.org/xml/atom.xml'
```

❶ Primero creamos un abridor de URL con el manejador de redirecciones que acabamos de definir.

❷ Enviamos una petición y obtuvimos un código de estado `301` por respuesta. Llegados aquí, se invoca el método `http_error_301`. Llamamos al método ancestro, que sigue la redirección y envía una petición al nuevo sitio (`http://diveintomark.org/xml/atom.xml`).

❸ Aquí está nuestra recompensa: ahora no sólo podemos acceder a la nueva URL, sino que también accedemos al código de estado de la redirección, de manera que podemos saber que fue una permanente. La próxima vez que pidamos estos datos deberíamos hacerlo a la dirección nueva (`http://diveintomark.org/xml/atom.xml`, como se especifica en `f.url`). Si hemos almacenado la dirección en un fichero de configuración o base de datos, deberemoos actualizarla de manera que no sigamos molestando al servidor con peticiones en la dirección antigua. Es hora de actualizar la libreta de direcciones.

El mismo manejador de redirecciones también puede decirle si *no debe* actualizar la libreta de direcciones.

Ejemplo 11.13. Uso del manejador de redirección para detectar redirecciones temporales

```
>>> request = urllib2.Request(
...     'http://diveintomark.org/redir/example302.xml')    ❶
>>> f = opener.open(request)
connect: (diveintomark.org, 80)
send: '
GET /redir/example302.xml HTTP/1.0
Host: diveintomark.org
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 302 Found\r\n'                             ❷
header: Date: Thu, 15 Apr 2004 22:18:21 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Location: http://diveintomark.org/xml/atom.xml
header: Content-Length: 314
header: Connection: close
header: Content-Type: text/html; charset=iso-8859-1
connect: (diveintomark.org, 80)
```

```
send: '
GET /xml/atom.xml HTTP/1.0                                    ❸
Host: diveintomark.org
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Thu, 15 Apr 2004 22:18:21 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Last-Modified: Thu, 15 Apr 2004 19:45:21 GMT
header: ETag: "e842a-3e53-55d97640"
header: Accept-Ranges: bytes
header: Content-Length: 15955
header: Connection: close
header: Content-Type: application/atom+xml
>>> f.status                                                   ❹
302
>>> f.url
http://diveintomark.org/xml/atom.xml
```

❶ Ésta es una URL de ejemplo que he preparado configurada para decirle a los clientes que se redirijan *temporalmente* a `http://diveintomark.org/xml/atom.xml`.

❷ El servidor devuelve un código de estado `302` que indica una redirección temporal. La nueva situación temporal de los datos la da la cabecera `Location:`.

❸ `urllib2` invoca nuestro método `http_error_302`, quien a su vez llama al método ancestro del mismo nombre en `urllib2.HTTPRedirectHandler`, que sigue la redirección al nuevo sitio. Entonces nuestro método `http_error_302` almacena el código de estado (`302`) para que la aplicación que lo invocó pueda obtenerlo después.

❹ Y aquí estamos, tras seguir con éxito la redirección a `http://diveintomark.org/xml/atom.xml`. `f.status` nos dice que fue una redirección temporal, lo que significa que deberíamos seguir pidiendo los datos a la dirección original (`http://diveintomark.org/redir/example302.xml`). Quizá nos redirija también la siguiente vez, o quizá no. Puede que nos redirija a un sitio

diferente. No es decisión nuestra. El servidor dijo que esta redirección sólo es temporal, así que deberíamos respetarlo. Y ahora estamos exponiendo a la aplicación que invoca información suficiente para que respete eso.

11.8. Tratamiento de datos comprimidos

La última característica importante de HTTP que queremos tratar es la compresión. Muchos servicios web tienen la capacidad de enviar los datos comprimidos, lo que puede rebajar en un 60% o más la cantidad de datos a enviar. Esto se aplica especialmente a los servicios web XML, ya que los datos XML se comprimen bastante bien.

Los servidores no nos van a dar comprimidos los datos a menos que le digamos que lo aceptamos así.

Ejemplo 11.14. Le decimos al servidor que queremos datos comprimidos

```
>>> import urllib2, httplib
>>> httplib.HTTPConnection.debuglevel = 1
>>> request = urllib2.Request('http://diveintomark.org/xml/atom.xml')
>>> request.add_header('Accept-encoding', 'gzip')        ❶
>>> opener = urllib2.build_opener()
>>> f = opener.open(request)
connect: (diveintomark.org, 80)
send: '
GET /xml/atom.xml HTTP/1.0
Host: diveintomark.org
User-agent: Python-urllib/2.1
Accept-encoding: gzip                                    ❷
'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Thu, 15 Apr 2004 22:24:39 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Last-Modified: Thu, 15 Apr 2004 19:45:21 GMT
header: ETag: "e842a-3e53-55d97640"
header: Accept-Ranges: bytes
header: Vary: Accept-Encoding
```

```
header: Content-Encoding: gzip                    ❸
header: Content-Length: 6289                      ❹
header: Connection: close
header: Content-Type: application/atom+xml
```

❶ Ésta es la clave: una vez creado el objeto `Request` object, añada una cabecera `Accept-encoding` para decirle al servidor que podemos aceptar datos gzip-encoded. `gzip` es el nombre del algoritmo de compresión que usamos. En teoría podría haber otros algoritmos de compresión, pero el que usa el 99% de los servidores es `gzip`.

❷ Ahí va nuestra cabecera camino al cable.

❸ Y aquí está lo que devuelve el servidor: la cabecera `Content-Encoding: gzip` significa que los datos que estamos a punto de recibir están comprimidos con gzip.

❹ La cabecera `Content-Length` es la longitud de los datos comprimidos, no de la versión sin comprimir. Como veremos en un momento, la longitud real de los datos sin comprimir era 15955, ¡así que la compresión gzip minimizó la descarga en más del 60%!

Ejemplo 11.15. Descompresión de los datos

```
>>> compresseddata = f.read()                     ❶
>>> len(compresseddata)
6289
>>> import StringIO
>>> compressedstream = StringIO.StringIO(compresseddata)   ❷
>>> import gzip
>>> gzipper = gzip.GzipFile(fileobj=compressedstream)      ❸
>>> data = gzipper.read()                         ❹
>>> print data                                    ❺
<?xml version="1.0" encoding="iso-8859-1"?>
<feed version="0.3"
  xmlns="http://purl.org/atom/ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xml:lang="en">
  <title mode="escaped">dive into mark</title>
  <link rel="alternate" type="text/html"
href="http://diveintomark.org/"/>
```

```
  <-- rest of feed omitted for brevity -->
>>> len(data)
15955
```

❶ Continuamos el ejemplo anterior, y `f` es el objeto de tipo fichero devuelto por el abridor de URL. Normalmente obtendríamos datos sin comprimir usando su método `read()`, pero dado que estos datos han sido comprimidos, éste sólo es el primer paso para obtener los datos que realmente queremos.

❷ Bien, este paso es un rodeo un poco sucio. Python tiene un módulo `gzip` que lee (y escribe) ficheros comprimidos en el disco. Pero no tenemos un fichero sino un búfer en memoria comprimido con gzip, y no queremos escribirlo en un fichero temporal sólo para poder descomprimirlo. Así que lo que haremos es crear un objeto de tipo fichero partiendo de los datos en memoria (`compresseddata`), haciendo uso del módulo `StringIO`. Conocimos este módulo en el capítulo anterior, pero ahora le hemos encontrado un nuevo uso.

❸ Ahora podemos crear una instancia de `GzipFile` y decirle que su "fichero" es el objeto tipo fichero `compressedstream`.

❹ Ésta es la línea que hace todo el trabajo: "leer" de `GzipFile` descomprimirá los datos. ¿Extraño? Sí, pero tiene sentido de una manera retorcida. `gzipper` es un objetoo de tipo fichero que representa un fichero comprimido con gzip. Ese "fichero" no es real, sin embargo; en realidad `gzipper` sólo está "leyendo" del objeto tipo fichero que creamos con `StringIO` para obtener los datos comprimidos, que están en memoria en la variable `compresseddata`. ¿Y de dónde vinieron esos datos comprimidos? Originalmente los descargamos de un servidor HTTP remoto "leyendo" el objeto tipo fichero que construimos con `urllib2.build_opener`. Y sorprendentemente, todo funciona. Cada paso en la cadena ignora que el paso anterior está engañándole.

❺ ¡Mira mamá!, datos de verdad (15955 bytes, para ser exactos).

"¡Pero espere!" puedo oírle gritar. "¡Esto podría ser incluso más sencillo!" Sé lo que está pensando. Está pensando que `opener.open` devuelve un fichero de tipo

objeto así que, ¿por qué no eliminar el paso intermedio por `StringIO` y limitarnos a pasar `f` directamente a `GzipFile`? Bueno vale, quizá no estuviera pensando eso, pero no se preocupe que tampoco hubiera funcionado.

Ejemplo 11.16. Descompresión de datos directamente del servidor

```
>>> f = opener.open(request)                    ❶
>>> f.headers.get('Content-Encoding')           ❷
'gzip'
>>> data = gzip.GzipFile(fileobj=f).read()      ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "c:\python23\lib\gzip.py", line 217, in read
    self._read(readsize)
  File "c:\python23\lib\gzip.py", line 252, in _read
    pos = self.fileobj.tell()   # Save current position
AttributeError: addinfourl instance has no attribute 'tell'
```

❶ Siguiendo con el ejemplo anterior, ya tenemos un objeto `Request` preparado con la cabecera `Accept-encoding: gzip`.

❷ Abrir la petición nos dará las cabeceras (aunque aún no descargará datos). Como puede ver de la cabecera `Content-Encoding` devuelta, estos datos se han enviado comprimidos con gzip.

❸ Como `opener.open` devuelve un objeto tipo fichero y sabemos por las cabeceras que cuando lo leamos vamos a obtener datos comprimidos con gzip, ¿por qué no pasarle directamente ese objeto tipo fichero a `GzipFile`? A medida que lea de la instancia de `GzipFile`, leerá datos comprimidos del servidor remoto HTTP y los descomprimirá al vuelo. Es una buena idea, pero desafortunadamente no funciona. Debido a la manera en que funciona la compresión gzip, `GzipFile` necesita guardar su posición y retroceder y avanzar por el fichero comprimido. Esto no funciona cuando el "fichero" es un flujo de bytes que viene de un servidor remoto; todo lo que podemos hacer es recoger bytes uno tras otro, y no movernos adelante y atrás por el flujo de datos. Por ello la mejor solución es el poco elegante uso de `StringIO`:

descargar los datos comprimidos, crear un objeto tipo fichero partiendo de ellos con `StringIO`, y luego descomprimir los datos con eso.

11.9. Todo junto

Ya hemos visto todas las partes necesarias para construir un cliente inteligente de servicios web HTTP. Ahora veamos cómo encaja todo.

Ejemplo 11.17. La función `openanything`

Esta función está definida en `openanything.py`.

```
def openAnything(source, etag=None, lastmodified=None,
agent=USER_AGENT):
    # se omite el código no relativo a HTTP por brevedad
    if urlparse.urlparse(source)[0] == 'http':
❶
        # open URL with urllib2
        request = urllib2.Request(source)
        request.add_header('User-Agent', agent)
❷
        if etag:
            request.add_header('If-None-Match', etag)
❸
        if lastmodified:
            request.add_header('If-Modified-Since', lastmodified)
❹
        request.add_header('Accept-encoding', 'gzip')
❺
        opener = urllib2.build_opener(SmartRedirectHandler(),
DefaultErrorHandler()) ❻
        return opener.open(request)
❼
```

❶ `urlparse` es un módulo de utilidad muy a mano para, adivínelo, analizar URLs. Su función primaria, llamada también `urlparse`, toma una URL y la divide en tuplas de (esquema, dominio, ruta, parámetros, parámetros de la cadena de consulta, e identificador de fragmento). De estos, lo único que nos

interesa es el esquema, para asegurarnos de que tratamos con una URL HTTP (que pueda manejar `urllib2`).

❷ Nos identificamos ante el servidor HTTP con la `User-Agent` que pasamos a la función de llamada. Si no se especifica una `User-Agent` trabajaremos con la definida por omisión anteriormente en el módulo `openanything.py`. Nunca usamos la que se define por omisión en `urllib2`.

❸ Si se nos dio una suma `ETag`, la enviamos en la cabecera `If-None-Match`.

❹ Si se nos dio una fecha de última modificación, la enviamos en la cabecera `If-Modified-Since`.

❺ Le decimos al servidor que querríamos datos comprimidos si fuera posible.

❻ Creamos un abridor de URL que usa *ambos* manejadores de URL personalizados: `SmartRedirectHandler` para controlar las redirecciones `301` y `302`, y `DefaultErrorHandler` para controlar grácilmente `304`, `404` y otras condiciones de error.

❼ ¡Eso es todo! Abre la URL y devuelve a quien invocó un objeto tipo fichero.

Ejemplo 11.18. La función `fetch`

Esta función está definida en `openanything.py`.

```
def fetch(source, etag=None, last_modified=None, agent=USER_AGENT):
    '''Fetch data and metadata from a URL, file, stream, or string'''
    result = {}
    f = openAnything(source, etag, last_modified, agent)
❶
    result['data'] = f.read()
❷
    if hasattr(f, 'headers'):
        # save ETag, if the server sent one
        result['etag'] = f.headers.get('ETag')
❸
        # save Last-Modified header, if the server sent one
        result['lastmodified'] = f.headers.get('Last-Modified')
❹
```

```
        if f.headers.get('content-encoding', '') == 'gzip':
❺
            # data came back gzip-compressed, decompress it
            result['data'] =
gzip.GzipFile(fileobj=StringIO(result['data']])).read()
    if hasattr(f, 'url'):
❻
        result['url'] = f.url
        result['status'] = 200
    if hasattr(f, 'status'):
❼
        result['status'] = f.status
    f.close()
    return result
```

❶ Primero llamamos a la función `openAnything` con una URL, una suma `ETag`, una fecha `Last-Modified` y una `User-Agent`.

❷ Leemos los datos devueltos por el servidor. Pueden estar comprimidos; si lo están los descomprimiremos luego.

❸ Guarda la suma `ETag` devuelta por el servidor para que la aplicación que llama pueda pasárnosla de nuevo la siguiente vez, y a su vez nosotros a `openAnything`, que puede adjuntarla a la cabecera `If-None-Match` y enviarla al servidor remoto.

❹ Guardamos también la fecha `Last-Modified`.

❺ Si el servidor dice que envió datos comprimidos, los descomprimimos.

❻ Si obtuvimos una URL del servidor, la guardamos y asumismo que el código de estado es `200` a menos que encontremos otra cosa.

❼ Si uno de los manejadores de URL personalizados capturó un código de estado, también lo guardamos.

Ejemplo 11.19. Uso de `openanything.py`

```
>>> import openanything
>>> useragent = 'MyHTTPWebServicesApp/1.0'
>>> url = 'http://diveintopython.org/redir/example301.xml'
>>> params = openanything.fetch(url, agent=useragent)          ❶
>>> params                                                     ❷
```

```
{'url': 'http://diveintomark.org/xml/atom.xml',
'lastmodified': 'Thu, 15 Apr 2004 19:45:21 GMT',
'etag': '"e842a-3e53-55d97640"',
'status': 301,
'data': '<?xml version="1.0" encoding="iso-8859-1"?>
<feed version="0.3"
<-- rest of data omitted for brevity -->'}
>>> if params['status'] == 301:                                    ❸
...     url = params['url']
>>> newparams = openanything.fetch(
...     url, params['etag'], params['lastmodified'], useragent)    ❹
>>> newparams
{'url': 'http://diveintomark.org/xml/atom.xml',
'lastmodified': None,
'etag': '"e842a-3e53-55d97640"',
'status': 304,
'data': ''}                                                        ❺
```

❶ La primera vez que pedimos un recurso no tenemos suma `ETag` o fecha `Last-Modified`, así que no las incluimos. (Son parámetros opcionales.)

❷ Lo que obtenemos es un diccionario de varias cabeceras útiles, el código de estado HTTP y los datos en sí devueltos por el servidor. `openanything` trata la compresión gzip internamente; no tenemos que preocuparnos de eso a este nivel.

❸ Si alguna vez obtenemos un código de estado `301`, eso es una redirección permanente y necesitamos actualizar la URL a la nueva dirección.

❹ La segunda vez que pedimos el mismo recurso, tenemos todo tipo de información para acompañar: Una URL (posiblemente actualizada), la `ETag` y la fecha `Last-Modified` de la última vez, y por supuesto nuestra `User-Agent`.

❺ De nuevo lo que obtenemos es un diccionario, pero los datos no han cambiado, así que todo lo que obtenemos es un código de estado `304` sin datos.

11.10. Resumen

`openanything.py` y sus funciones deberían tener sentido ahora.

Hay 5 características importantes de los servicios web HTTP que cada cliente debería admitir:

- Identificar nuestra aplicación estableciendo una `User-Agent` apropiada.
- Tratamiento adecuado de redirecciones permanentes.
- Soporte de comprobación de fecha `Last-Modified` para evitar descargar de nuevo datos que no han cambiado.
- Soporte de sumas de comprobación `ETag` para evitar descargar de nuevo datos que no han cambiado.
- Soporte de compresión gzip para reducir el consumo de ancho de banda incluso cuando los datos *hayan* cambiado.

Capítulo 12. Servicios web SOAP

Capítulo 11 se centró en los servicios web sobre HTTP orientados a documentos. El "parámetro de entrada" era la URL y el "valor devuelto" era un documento XML que debíamos responsabilizarnos de interpretar.

Este capítulo se centrará en los servicios web SOAP, que tienen un enfoque más estructurad. En lugar de trabajar directamente con consultas HTTP y documentos XML SOAP nos permite simular llamadas a función que devuelven tipos de datos nativos. Como veremos, la ilusión es casi perfecta; podemos "invocat" una función mediante una biblioteca de SOAP, con la sintaxis estándar de Python, y resulta que la función devuelve valores y objetos de Python. Sin embargo, bajo esa apariencia la biblioteca de SOAP ha ejecutado en realidad una transacción compleja que implica varios documentos XML y un servidor remoto.

SOAP es una especificación compleja y es un poco inexacto decir que SOAP se trata de llamadas a funciones remotas. Algunas personas añadirían que SOAP permite realizar paso asíncrono de mensajes en un solo sentido y servicios web orientados al documento. Y estarían en lo correcto; SOAP se puede usar de esa

manera y de muchas maneras diferentes. Pero este capítulo se centrará en el también llamado SOAP "al estilo RPC" (llamar a una función remota y obtener sus resultados).

12.1. Inmersión

Usted usa Google, ¿cierto? Es un motor de búsquedas popular. ¿Ha deseado alguna vez poder acceder a los resultados de búsquedas de Google de forma programática? Ahora puede. Aquí tiene un programa que busca en Google desde Python.

Ejemplo 12.1. `search.py`

```
from SOAPpy import WSDL

# you'll need to configure these two values;
# see http://www.google.com/apis/
WSDLFILE = '/path/to/copy/of/GoogleSearch.wsdl'
APIKEY = 'YOUR_GOOGLE_API_KEY'

_server = WSDL.Proxy(WSDLFILE)
def search(q):
    """Search Google and return list of {title, link, description}"""
    results = _server.doGoogleSearch(
        APIKEY, q, 0, 10, False, "", False, "", "utf-8", "utf-8")
    return [{"title": r.title.encode("utf-8"),
             "link": r.URL.encode("utf-8"),
             "description": r.snippet.encode("utf-8")}
            for r in results.resultElements]

if __name__ == '__main__':
    import sys
    for r in search(sys.argv[1])[:5]:
        print r['title']
        print r['link']
        print r['description']
        print
```

Puede importar esto como módulo y usarlo desde otro programa mayor, o puede ejecutar el script desde la línea de órdenes. Desde la línea de órdenes puede indicar la consulta de búsqueda como argumento, y como resultado obtendrá la URL, título y descripción de los cinco primeros resultados de Google.

Aquí tiene la salida de ejemplo para una búsqueda de la palabra "python"

Ejemplo 12.2. Ejemplo de uso de `search.py`

```
C:\diveintopython\common\py> python search.py "python"
<b>Python</b> Programming Language
http://www.python.org/
Home page for <b>Python</b>, an interpreted, interactive, object-
oriented,
extensible<br> programming language. <b>...</b> <b>Python</b>
is OSI Certified Open Source: OSI Certified.

<b>Python</b> Documentation Index
http://www.python.org/doc/
 <b>...</b> New-style classes (aka descrintro). Regular expressions.
Database
API. Email Us.<br> docs@<b>python</b>.org. (c) 2004. <b>Python</b>
Software Foundation. <b>Python</b> Documentation. <b>...</b>

Download <b>Python</b> Software
http://www.python.org/download/
Download Standard <b>Python</b> Software. <b>Python</b> 2.3.3 is the
current production<br> version of <b>Python</b>. <b>...</b>
<b>Python</b> is OSI Certified Open Source:

Pythonline
http://www.pythonline.com/


Dive Into <b>Python</b>
http://diveintopython.org/
Dive Into <b>Python</b>. <b>Python</b> from novice to pro. Find:
<b>...</b> It is also available in multiple<br> languages. Read
Dive Into <b>Python</b>. This book is still being written. <b>...</b>
```

Lecturas complementarias sobre SOAP

- http://www.xmethods.net/ es un repositorio de servicios web SOAP de acceso público.
- La especificación de SOAP es sorprendentemente legible, si es que le gustan ese tipo de cosas.

12.2. Instalación de las bibliotecas de SOAP

Al contrario que el resto de código de este libro, este capítulo se apoya en bibliotecas que no se incluyen preinstaladas den Python.

Antes de que pueda zambullirse en los servicios web SOAP necestará instalar tres bibliotecas: PyXML, fpconst y SOAPpy.

12.2.1. Instalación de PyXML

La primera biblioteca que necesitará es PyXML, un conjunto avanzado de bibliotecas de XML que proporcionan más funcionalidad que las bibliotecas de XML incoradas por el lenguaje que estudiamos en Capítulo 9.

Procedimiento 12.1.

Éste es el procedimiento para instalar PyXML:

1. Acceda a http://pyxml.sourceforge.net/, pulse en Downloads y descargue la última versión para su sistema operativo.
2. Si está usando Windows tiene varias opcones. Asegúrese de descargar la versión de PyXML que corresponda a la de Python que esté usando.

3. Haga clic dos veces sobre el instlador. Si descargó PyXML 0.8.3 para Windows y Python 2.3, el programa instalador será `PyXML-0.8.3.win32-py2.3.exe`.
4. Avance por las opciones del programa instalador.
5. Tras completar la instalación, cierre el instalador. No habrá indicación visible de éxito (no se instalan programas en el Menú de Inicio ni atajos en el escritorio). PyXML es simplemente una colección de librerías XML usadas por otros programas.

Para verificar que ha instalado PyXML correctamente, ejecute su IDE de Python y compruebe la versión de las bibliotecas XML que tiene instaladas, como se muestra aquí.

Ejemplo 12.3. Verificación de la instalación de PyXML

```
>>> import xml
>>> xml.__version__
'0.8.3'
```

Éste número de versión debería corresponder con el del programa instalador de PyXML que descargó y ha ejecutado.

12.2.2. Instalación de fpconst

La segunda biblioteca que necesita es fpconst, un conjunto de constantes y funciones para trabajar con valores especiales de doble precisión IEEE754. Esto da soporte para tratar con los valores especiales Not-a-Number (NaN), Infinito positivo (Inf) e Infinito negativo (-Inf), que son parte de la especificación de tipos de datos de SOAP.

Procedimiento 12.2.

Aquí tiene el procedimiento para instalar fpconst:

1. Descargue la última versión de fpconst que encontrará en http://www.analytics.washington.edu/statcomp/projects/rzope/fpconst/.
2. Hay dos descargas disponibles, una en formato `.tar.gz`, la otra en `.zip`. Si está usando Windows, descargue el fichero `.zip`; si no, descargue el fichero `.tar.gz`
3. Descomprima el fichero descargado. En Windows XP puede hacer clic derecho sobre el fichero y escoger "Extraer todo"; en versiones anteriores de Windows necesitará un programa de terceros como WinZip. En Mac OS X puede hacer clic derecho sobre el fichero comprimido para descomprimirlo con Stuffit Expander.
4. Abra una consola y sitúese en el directorio donde descomprimió los ficheros fpconst.
5. Escriba **`python setup.py install`** para ejecutar el programa de instalación.

Ejecute su IDE Python y compruebe el número de versión para verificar que ha instalado fpconst correctamente.

Ejemplo 12.4. Verificación de la instalación de fpconst

```
>>> import fpconst
>>> fpconst.__version__
'0.6.0'
```

Este número de versión debería corresponder al del archivo fpconst que ha descargado e instalado.

12.2.3. Instalación de SOAPpy

El tercer y último requisito es la propia biblioteca de SOAP: SOAPpy.

Procedimiento 12.3.

Éste es el procedimiento para instalar SOAPpy:

1. Seleccione *Latest Official Release* en http://pywebsvcs.sourceforge.net/, bajo la sección SOAPpy.
2. Hay dos descargas disponibles. Si usa Windows descargue el fichero `.zip`; si no, descargue el fichero `.tar.gz`.
3. Descomprima el fichero descargado, igual que hizo con fpconst.
4. Abra una consola y sitúese en el directorio donde descomprimió los ficheros de SOAPpy.
5. Escriba **`python setup.py install`** para ejecutar el programa de instalación.

Ejecute su IDE Python y compruebe el número de versión para verificar que ha instalado SOAPpy correctamente.

Ejemplo 12.5. Verificación de la instalación de SOAPpy

```
>>> import SOAPpy
>>> SOAPpy.__version__
'0.11.4'
```

Este número de versión debería corresponder al del archivo SOAPpy que ha descargado e instalado.

12.3. Primeros pasos con SOAP

El corazón de SOAP es la capacidad de invocar funciones remotas. Hay varios servidores SOAP de acceso público que proporcionan funciones simples con propósito de demostración.

El más popular de los servidores SOAP de acceso público es http://www.xmethods.net/. Este ejemplo usa una función de demostración que toma un código postal de los Estados Unidos de América y devuelve la temperatura actual en esa región.

Ejemplo 12.6. Obtención de la temperatura actual

```
>>> from SOAPpy import SOAPProxy                    ❶
```

```
>>> url = 'http://services.xmethods.net:80/soap/servlet/rpcrouter'
>>> namespace = 'urn:xmethods-Temperature'  ❷
>>> server = SOAPProxy(url, namespace)      ❸
>>> server.getTemp('27502')                 ❹
80.0
```

❶ Accedemos al servidor SOAP remoto mediante una clase proxy, `SOAPProxy`. El proxy gestiona por usted todas las operaciones internas de SOAP, incluyendo la creación del documento XML de consulta partiendo del nombre de la función y su lista de argumentos, enviándola sobre HTTP al servidor SOAP remoto, analizando el documento XML de respuesta, y creando valores nativos de Python que devolver. Verá el aspecto de estos documentos XML en la próxima sección.

❷ Cada servicio SOAP tiene una URL que gestiona todas las consultas. Se usa la mismoa URL en todas las llamadas. Este servicio particular sólo tiene una función, pero más adelante en el capítulo verá ejemplos de la API de Google, que tiene varias funciones. La URL de servicio la comparten todas las funciones.Cada servicio de SOAP tiene también un espacio de nombres que está definido por el servidor y es completamente arbitrario. Es parte de la configuración que se precisa para llamar a los métodos de SOAP. Permite al servidor compartir una única URL de servicio y entregar consultas a diferentes servicios sin relación entre ellos. Es como dividir Python en paquetes.

❸ Estamos creando el `SOAPProxy` con la URL y el espacio de nombres del servicio. Esto no hace ninguna conexión con el servidor SOAP; simplemente crea un objeto local de Python.

❹ Ahora que todo está configurado correctamente puede invocar métodos de SOAP como si fueran funciones locales. Se le pasan argumentos de la misma manera que a una función normal, y se obtienen valores de retorno igual que con una función normal. Pero tras el telón están sucediendo muchas cosas.

Echemos una mirada tras el telón.

12.4. Depuración de servicios web SOAP

Las bibliotecas de SOAP proporcionan una manera sencilla de comprobar lo que está sucediendo tras la escena.

Activar la depuración es simplemente cuestión de cambiar dos indicadores en la configuraciónl `SOAPProxy`.

Ejemplo 12.7. Depuración de servicios web SOAP

```
>>> from SOAPpy import SOAPProxy
>>> url = 'http://services.xmethods.net:80/soap/servlet/rpcrouter'
>>> n = 'urn:xmethods-Temperature'
>>> server = SOAPProxy(url, namespace=n)      ❶
>>> server.config.dumpSOAPOut = 1             ❷
>>> server.config.dumpSOAPIn = 1
>>> temperature = server.getTemp('27502')     ❸
*** Outgoing SOAP
*************************************************
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<ns1:getTemp xmlns:ns1="urn:xmethods-Temperature" SOAP-ENC:root="1">
<v1 xsi:type="xsd:string">27502</v1>
</ns1:getTemp>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
************************************************************************
**
*** Incoming SOAP
*************************************************
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
<ns1:getTempResponse xmlns:ns1="urn:xmethods-Temperature"
```

```
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<return xsi:type="xsd:float">80.0</return>
</ns1:getTempResponse>

</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
************************************************************************
**

>>> temperature
80.0
```

❶ Primero creamos `SOAPProxy` igual que siempre, con la URL y el espacio de nombres del servicio.

❷ Segundo, activamos la depuración cambiando `server.config.dumpSOAPIn` y `server.config.dumpSOAPOut`.

❸ Tercero, invoque el método remoto SOAP de la manera normal. La biblioteca de SOAP imprimirá tanto el documento XML de la petición saliente como el de la respuesta entrante. Éste es todo el trabajo duro que está haciendo `SOAPProxy` por nosotros. Intimidante, ¿verdad? Analicémoslo.

La mayoría del documento de petición XML que se envía al servidor es una plantilla. Ignore todas las declaraciones de espacio de nombres; van a ser lo mismo (o parecido) en todas las invocaciones SOAP. El centro de la "llamada a función" es este fragmento dentro del elemento `<Body>`:

```
<ns1:getTemp                                    ❶
  xmlns:ns1="urn:xmethods-Temperature"          ❷
  SOAP-ENC:root="1">
<v1 xsi:type="xsd:string">27502</v1>            ❸
</ns1:getTemp>
```

❶ El nombre del elemento es a su vez el nombre de la función, `getTemp`. `SOAPProxy` utiliza [getattr para despachar]. En lugar de llamar a varios métodos locales diferentes basándose en el nombre del método, utiliza este último para construir el documento XML de petición.

❷ El elemento XML de la función está contenido en un espacio de nombres

específico, que es el que especificamos cuando creamos el objeto `SOAPProxy`. No se preocupe por el `SOAP-ENC:root`; también sale de una plantilla.

❸ Los argumentos de la función se traducen también a XML. `SOAPProxy` hace uso de introspección sobre cada argumento para determinar el tipo de datos (en este caso una cadena). El tipo del argumento va en el atributo `xsi:type`, seguido por una cadena con el valor en sí.

El documento XML devuelto es igualmente sencillo de entender, una vez que sabe lo que ha de ignorar. Concéntrese en este fragmento dentro del `<Body>`:

```
<ns1:getTempResponse                                    ❶
  xmlns:ns1="urn:xmethods-Temperature"                  ❷
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<return xsi:type="xsd:float">80.0</return>              ❸
</ns1:getTempResponse>
```

❶ El servidor encapsula el valor de vuelta de la función dentro de un elemento `<getTempResponse>`. Por convención, este elemento encapsulador es el nombre de la funci`ón, más `Response`. Pero en realidad podría ser casi cualquier cosa; la cosa importante en que se fija `SOAPProxy` no es el nombre del elemento, sino el espacio de nombres.

❷ El servidor devuelve la respuesta en el mismo espacio de nombresq ue usamos en la consulta, el mismo que especificamos cuando creamos el `SOAPProxy`. Más adelante en este capítulo veremos lo que sucede si olvidamos especificar el espacio de nombres al crear el `SOAPProxy`.

❸ Se especifica el valor devuelto junto con su tipo de datos (coma flotante). `SOAPProxy` utiliza este tipo de dato explícito para crear un objeto de Python del tipo nativo correcto y lo devuelve.

12.5. Presentación de WSDL

La clase `SOAPProxy` hace proxy de llamadas locales a métodos y las convierte de forma transparente en invocaciones a métodos SOAP remotos. Como ya ha visto esto lleva mucho trabajo, y el `SOAPProxy` lo hace rápida y

transparentemente. Lo que no hace es proporcionarnos ningún tipo de método para introspección.

Consideremos esto: las dos secciones anteriores mostraron un ejemplo de llamada a un método remoto SOAP sencillo con un argumento y un valor de vuelta, ambos de tipos de datos simples. Esto precisó saber y hacer un seguimiento de: la URL del servicio, el espacio de nombres, el nombre de la función, el número de argumentos, y el tipo de datos de cada uno. Si alguno de estos falta o es incorrecto, todo se cae en pedazos.

Eso no debería sorprendernos. Si yo quisiera invocar una función local necesitaría saber en qué paquete o módulo se encuentra (el equivalente a la URL y espacio de nombres del servicio). Necesitaría saber el nombre y número de argumentos correctos de la función. Python maneja con destreza los tipos de datos sin necesidad de explicitarlos, pero aún así necesita saber cuántos ha de pasar, y cuántos se esperan de vuelta.

La gran diferencia es la introspección. Como pudo ver en Capítulo 4, Python es excelente en lo que se refiere a permitirle descubrir cosas sobre módulos y funciones en tiempo de ejecución. Puede listar las funciones que dispone un módulo y, con un poco de trabajo, ahondar hasta la declaración y argumentos de funciones individuales.

WSDL le permite hacer esto con los servicios web SOAP. WSDL significa "Web Services Description Language[14]". Aunque se diseñó para ser lo suficientemente flexible para describir muchos tipos de servicios web, se usa con más frecuencia para describir servicios SOAP.

Un fichero WSDL es eso: un fichero. Más específicamente, es un fichero XML. Suele residir en el mismo servidor al que accede en busca del servicio web SOAP que describe, aunque no hay nada especial en él. Más adelante en este capítulo descargaremos el fichero WSDL de la API de Google y lo usaremos de forma local. Eso no significa que vayamos invocar localmente a Google; el

fichero WSDL describe las funciones remotas que se encuentran en el servidor de Google.

Un fichero WSDL contiene una descripción de todo lo que implica una llamada a un servicio web SOAP:

- La URL y el espacio de nombres del servicio
- El tipo de servicio web (probablemente llamadas a función usando SOAP aunque, como mencioné, WSDL es suficientemente flexible para describir un amplio abanico de servicios web)
- La lista de funciones disponibles
- Los argumentos de cada función
- El tipo de dato de cada argumento
- Los valores de retorno de cada función, y el tipo de dato de cada uno

En otras palabras, un fichero WSDL le dice todo lo que necesita saber para poder llamar a un servicio web SOAP.

Footnotes

[14] Lenguaje de descripción de servicios web

12.6. Introspección de servicios web SOAP con WSDL

Como muchas otras cosas en el campo de los servicios web, WSDL tiene una historia larga y veleidosa, llena de conflictos e intrigas políticas. Me saltaré toda esta historia ya que me aburre hasta lo indecible. Hay otros estándares que intentaron hacer cosas similares, pero acabó ganando WSDL así que aprendamos cómo usarlo.

Lo más fundamental que nos permite hacer WSDL es descubrir los métodos que nos ofrece un servidor SOAP.

Ejemplo 12.8. Descubrimiento de los métodos disponibles

```
>>> from SOAPpy import WSDL                    ❶
>>> wsdlFile =
'http://www.xmethods.net/sd/2001/TemperatureService.wsdl'
>>> server = WSDL.Proxy(wsdlFile)              ❷
>>> server.methods.keys()                      ❸
[u'getTemp']
```

❶ SOAPpy incluye un analizador de WSDL. En el momento en que escribo esto se indicaba que estaba en las fases tempranas de desarrollo, pero no he tenido problemas con ninguno de los ficheros WSDL que he probado.

❷ Para usar un fichero WSDL debemos de nuevo utilizar una clase proxy, `WSDL.Proxy`, que toma un único argumento: el fichero WSDL. Observe que en este caso estamos pasando la URL de un fichero WSDL almacenado en un servidor remoto, pero las clases proxy trabajan igual de bien con una copia local del fichero WSDL. El acto de crear el proxy WSDL descargará el fichero y lo analizará, así que si hubiera algún error en el fichero WSDL (o si no pudiera acceder a él debido a problemas de red) lo sabría de inmediato.

❸ La clase proxy WSDL expone las funciones disponibles como un diccionario de Python, `server.methods`. Así que obtener la lista de métodos disponibles es tan simple como invocar el método `keys()` del diccionario.

Bien, así que sabemos que este servidor SOAP ofrece un solo método: `getTemp`. Pero, ¿cómo lo invocamos? El objeto proxy WSDL también nos puede decir eso.

Ejemplo 12.9. Descubrimiento de los argumentos de un método

```
>>> callInfo = server.methods['getTemp']  ❶
>>> callInfo.inparams                     ❷
[<SOAPpy.wstools.WSDLTools.ParameterInfo instance at 0x00CF3AD0>]
>>> callInfo.inparams[0].name             ❸
u'zipcode'
>>> callInfo.inparams[0].type             ❹
(u'http://www.w3.org/2001/XMLSchema', u'string')
```

❶ El diccionario `server.methods` contiene una estructura específica de SOAPpy llamada `CallInfo`. Un objeto `CallInfo` lleva información sobre una función específica, incluyendo sus argumentos.

❷ Los argumentos de la función están almacenados en `callInfo.inparams`, que es una lista de objetos `ParameterInfo` que contienen información sobre cada parámetro.

❸ Cada objeto `ParameterInfo` contiene un atributo `name` que es el nombre del argumento. No hace falta saber el nombre del argumento para llamar a la función mediante SOAP, pero SOAP admite invocación de funciones con argumentos por nombre (igual que Python), y `WSDL.Proxy` hará relacionará correctamente los argumentos nombrados con la función remota si escoge usarlos.

❹ También se da el tipo explícito de cada parámetro, usando tipos definidos en XML Schema. Ya vio esto en la traza de datos de la sección anterior; el espacio de nombres de XML Schema era parte de la "plantilla" que le pedí que ignorase. Puede continuar ignorándolos para nuestros propósitos actuales. El parámetro `zipcode` es una cadena de texto y si pasa una cadena de Python al objeto `WSDL.Proxy`, lo convertirá correctamente para enviarlo al servidor.

WSDL también le permite hacer introspección sobre los valores de vuelta de una función.

Ejemplo 12.10. Descubrimiento de los valores de retorno de un método

```
>>> callInfo.outparams                    ❶
[<SOAPpy.wstools.WSDLTools.ParameterInfo instance at 0x00CF3AF8>]
>>> callInfo.outparams[0].name            ❷
u'return'
>>> callInfo.outparams[0].type
(u'http://www.w3.org/2001/XMLSchema', u'float')
```

❶ El adjunto de `callInfo.inparams` para los argumentos de las funciones es

`callInfo.outparams` para sus valores de retorno. También es una lista, porque las funciones invocadas mediante SOAP pueden devolver varios valores, igual que las funcionesde Python.

❷ Cada objeto `ParameterInfo` contiene nombre (`name`) y tipo (`type`). Esta función devuelve un solo valor, de nombre `return`, que es de coma flotante.

Juntémoslo todo y llamemos a un servicio web SOAP mediante un proxy WSDL.

Ejemplo 12.11. Invocación de un servicio web mediante un proxy WSDL

```
>>> from SOAPpy import WSDL
>>> wsdlFile =
'http://www.xmethods.net/sd/2001/TemperatureService.wsdl')
>>> server = WSDL.Proxy(wsdlFile)              ❶
>>> server.getTemp('90210')                    ❷
66.0
>>> server.soapproxy.config.dumpSOAPOut = 1    ❸
>>> server.soapproxy.config.dumpSOAPIn = 1
>>> temperature = server.getTemp('90210')
*** Outgoing SOAP
*************************************************
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<ns1:getTemp xmlns:ns1="urn:xmethods-Temperature" SOAP-ENC:root="1">
<v1 xsi:type="xsd:string">90210</v1>
</ns1:getTemp>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
************************************************************************
**
```

```
*** Incoming SOAP
*****************************************************
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
<ns1:getTempResponse xmlns:ns1="urn:xmethods-Temperature"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<return xsi:type="xsd:float">66.0</return>
</ns1:getTempResponse>

</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
*****************************************************************
**

>>> temperature
66.0
```

❶ La configuración es más sencilla que al llamar directamente al servicio SOAP, ya que el fichero WSDL contiene tanto la URL como el espacio de nombres que necesitamos para llamar al servicio. La creación del objeto `WSDL.Proxy` descarga el fichero WSDL, lo analiza y configura un objeto `SOAPProxy` que usa para invocar al servicio web SOAP.

❷ Una vez se crea el objeto `WSDL.Proxy` invocamos a la función tan fácilmente como lo haríamos con el objeto `SOAPProxy`. Esto no es sorprendente; el `WSDL.Proxy` es un simple envoltorio alrededor de `SOAPProxy` con algunos métodos de introspección añadidos, así que la sintaxis de llamada de funciones es la misma.

❸ Podemos acceder al `SOAPProxy` de `WSDL.Proxy` con `server.soapproxy`. Esto es útil para activar la depuración, para que cuando invoquemos funciones mediante el proxy WSDL su `SOAPProxy` vuelque los documentos XML de salida y entrada.

12.7. Búsqueda en Google

Volvamos finalmente al ejemplo de código que vio al comienzo de este capítulo, que hace algo más útil y excitante que obtener la temperatura.

Google proporciona un API de SOAP para acceder de forma programática a buscar resultados. Para usarla deberá suscribirse a los Google Web Services.

Procedimiento 12.4. Suscripción a los Google Web Services

1. Vaya ahttp://www.google.com/apis/ y cree una cuenta en Google. Esto sólo precisa de una dirección de correo electrónico. Tras registrrase recibirá por correo la clave de licencia de la Google API. Necesitará esta clave para pasarla como parámetro cada vez que invoque las funciones de búsqueda de Google.
2. Descargue también desde http://www.google.com/apis/ el kit de desarrollo de Google Web API. Incluye algo de código de ejemplo en varios lenguajes de programación (pero no Python), y lo más importante, incluye el fichero WSDL.
3. Descomprima el fichero del kit de desarrolo y busque `GoogleSearch.wsdl`. Copie este fichero a algún sitio permanente en su disco duro. Lo necesitará más adelante en este capítulo.

Una vez tenga la clave de desarrollador y el fichero WSDL de Google en un sitio conocido, puede empezar a jugar con los Google Web Services.

Ejemplo 12.12. Introspección de los Google Web Services

```
>>> from SOAPpy import WSDL
>>> server = WSDL.Proxy('/path/to/your/GoogleSearch.wsdl') ❶
>>> server.methods.keys()                                  ❷
[u'doGoogleSearch', u'doGetCachedPage', u'doSpellingSuggestion']
>>> callInfo = server.methods['doGoogleSearch']
>>> for arg in callInfo.inparams:                          ❸
...     print arg.name.ljust(15), arg.type
key             (u'http://www.w3.org/2001/XMLSchema', u'string')
q               (u'http://www.w3.org/2001/XMLSchema', u'string')
start           (u'http://www.w3.org/2001/XMLSchema', u'int')
maxResults      (u'http://www.w3.org/2001/XMLSchema', u'int')
```

```
filter          (u'http://www.w3.org/2001/XMLSchema', u'boolean')
restrict        (u'http://www.w3.org/2001/XMLSchema', u'string')
safeSearch      (u'http://www.w3.org/2001/XMLSchema', u'boolean')
lr              (u'http://www.w3.org/2001/XMLSchema', u'string')
ie              (u'http://www.w3.org/2001/XMLSchema', u'string')
oe              (u'http://www.w3.org/2001/XMLSchema', u'string')
```

❶ Empezar con los servicios web de Google es sencillo: cree un objeto `WSDL.Proxy` y diríjalo a su copia local del fichero WSDL de Google.

❷ De acuerdo con el fichero WSDL, Google ofrece tres funciones: `doGoogleSearch`, `doGetCachedPage`, y `doSpellingSuggestion`. Hacen exactamente lo que dicen: realizar una búsqueda y recoger los resultados programáticamente, obtener acceso a la versión en caché de una página de la última vezq ue la vio Google, y ofrecer sugerencias de deletro de palabras que se suelen escribir mal.

❸ La función `doGoogleSearch` toma varios parámetros de diferentes tipos. Observe que aunque el fichero WSDL puede dcirnos cómo se llaman y de qué tipo son los argumentos, no puede decirnos lo que significan o cómo usarlos. Teóricamente podría decirnos el rango aceptable de valores de cada parámetro si se permitieran unos valores específicos, pero el fichero WSDL de Google no es tan detallado. `WSDL.Proxy` no puede hacer magia; sólo nos dará la información que proporcione el fichero WSDL.

Aquí tiene una breve sinopsis de todos los parámetros de la función `doGoogleSearch`:

- `key` - La clave de la Google API, que recibimos al suscribir los servicios web de Google.
- `q` - La palabra o frase que vamos a buscar. La sintaxis es exactamente la misma que la del formulario web de Google, así que si conoce sintaxis o trucos avanzados funcionarán aquí igualmente.
- `start` - El índice del resultado en el que empezar. Igual que en la versión interactiva de Google, esta función devuelve 10 resultados cada vez. Si quiere empezar en la segunda “página” de resultados, deberá poner `start` en 10.

- `maxResults` - El número de resultados a devolver. Actualmente limitado a 10, aunque se pueden especificar menos si sólo interesan unos pocos resultados y queremos ahorrar algo de ancho de banda.
- `filter` - Si es `True`, Google filtrará las páginas duplicadas de los resultados.
- `restrict` - Ponga aquí `country` junto con un código de país para obtener resultados particulares de un solo país. Ejemplo: `countryUK` para buscar páginas en el Reino Unido. Puede especificar también `linux`, `mac`, o `bsd` para buscar en conjuntos de sitios técnicos definidos por Google, o `unclesam` para buscar en sitios sobre el gobierno de los Estados Unidos.
- `safeSearch` - Si es `True`, Google filtrará los sitios porno.
- `lr` ("restricción de idioma") - Ponga aquí un código de idioma si desea obtener resultados sólo en un idioma en particular.
- `ie` y `oe` ("codificación de entrada" y "codificación de salida") - Obsoletos, deben ser ambos `utf-8`.

Ejemplo 12.13. Búsqueda en Google

```
>>> from SOAPpy import WSDL
>>> server = WSDL.Proxy('/path/to/your/GoogleSearch.wsdl')
>>> key = 'YOUR_GOOGLE_API_KEY'
>>> results = server.doGoogleSearch(key, 'mark', 0, 10, False, "",
...     False, "", "utf-8", "utf-8")             ❶
>>> len(results.resultElements)                  ❷
10
>>> results.resultElements[0].URL                ❸
'http://diveintomark.org/'
>>> results.resultElements[0].title
'dive into <b>mark</b>'
```

❶ Tras crear el objeto `WSDL.Proxy`, debemos llamar a `server.doGoogleSearch` con los diez parámetros. Recuerde usar su propia clave de la Google API que recibió cuando suscribió los servicios web de Google.

❷ Se devuelve mucha información, pero miremos primero los resultados de la búsqueda. Están almacenados en `results.resultElements`, y podemos acceder a ellos como a una lista normal de Python.

❸ Cada elemento de `resultElements` es un objeto que tiene `URL`, `title`, `snippet` y otros atributos útiles. En este momento puede usar técnicas normales de introspección de Python como **`dir(results.resultElements[0])`** para ver los atributos disponibles. O puede hacer introspección mediante el objeto proxy WSDL y ver los `outparams` de la función. Cada técnica le dará la misma información.

El objeto `results` contiene más que los resultados de la búsqueda. También incluye información sobre la búsqueda en sí, como el tiempo que llevó y cuántos resultados se encontraron (aunque sólo se devolvieran 10). La interfaz web de Google muestra esta información, y también puede acceder a ella programáticamente.

Ejemplo 12.14. Acceso a información secundaria de Google

```
>>> results.searchTime                          ❶
0.224919
>>> results.estimatedTotalResultsCount          ❷
29800000
>>> results.directoryCategories                 ❸
[<SOAPpy.Types.structType item at 14367400>:
 {'fullViewableName':

'Top/Arts/Literature/World_Literature/American/19th_Century/Twain,_Mar
k',
  'specialEncoding': ''}]
>>> results.directoryCategories[0].fullViewableName
'Top/Arts/Literature/World_Literature/American/19th_Century/Twain,_Mar
k'
```

❶ Esta búsqueda llevó 0.224919 segundos. Eso no incluye el tiempo que tardó el envío y recepción de documentos XML de SOAP. Sólo el tiempo que empleó Google procesando la consulta una vez que la recibió.

❷ En total hubieron aproximadamente unos 30 millones de resultados. Puede acceder a ellos a 10 por vez cambiando el parámetro `start` e invocando `server.doGoogleSearch` de nuevo.

❸ En algunas consultas Google también devuelve una lista de categorías del Google Directory relacionadas. Puede añadir estas URL a http://directory.google.com/ para construir el enlace a la página de categoría del directorio.

12.8. Solución de problemas en servicios web SOAP

Por supuesto, el mundo de los servicios web SOAP no es todo luz y felicidad. Algunas veces las cosas van mal.

Como ha visto durante este capítulo, SOAP supone varias capas. Primero está la capa HTTP, ya que SOAP envía y recibe documentos a y desde un servidor HTTP. Así que entran en juego todas las técnicas de depuración que aprendió en Capítulo 11, *Servicios Web HTTP*. Puede hacer `import httplib` y luego `httplib.HTTPConnection.debuglevel = 1` para ver el tráfico HTTP subyacente.

Más allá de la capa HTTP hay varias cosas que pueden ir mal. SOAPpy hace un trabajo admirable ocultándonos la sintaxis de SOAP, pero eso también significa que puede ser difícil determinar dónde está el problema cuando las cosas no funcionan.

Aquí tiene unos pocos ejemplos de fallos comunes que he cometido al suar servicios web SOAP, y los errores que generaron.

Ejemplo 12.15. Invocación de un método con un proxy configurado incorrectamente

```
>>> from SOAPpy import SOAPProxy
>>> url = 'http://services.xmethods.net:80/soap/servlet/rpcrouter'
>>> server = SOAPProxy(url)                                          ❶
>>> server.getTemp('27502')                                          ❷
<Fault SOAP-ENV:Server.BadTargetObjectURI:
Unable to determine object id from call: is the method element
namespaced?>
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 453, in
__call__
    return self.__r_call(*args, **kw)
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 475, in
__r_call
    self.__hd, self.__ma)
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 389, in
__call
    raise p
SOAPpy.Types.faultType: <Fault SOAP-ENV:Server.BadTargetObjectURI:
Unable to determine object id from call: is the method element
namespaced?>
```

❶ ¿Vio la equivocación? Estamos creando un `SOAPProxy` de forma manual, y especificamos correctamente la URL del servicio pero no el espacio de nombres. Dado que se puede acceder a servicios múltiples mediante la misma URL de servicio es esencial el espacio de nombres a la hora de determinar a cual de ellos queremos hablar, y por lo tanto a cual método estamos invocando realmente.

❷ El servidor responde enviando una Fault SOAP, que SOAPpy convierte en una excepción de Python del tipo `SOAPpy.Types.faultType`. Todos los errores devueltos por cualquier servidor SOAP serán siempre Faults de SOAP, así que podemos capturar esta excepción fácilmente. En este caso, la parte legible por humanos de la Fault de SOAP nos da una pista del problema: el elemento del método no está en un espacio de nombres, porque no se configuró con un espacio de nombres el objeto `SOAPProxy` original.

Uno de los problemas que intenta resolver WSDL es la desconfiguración de elementos básicos del servicio SOAP. El fichero WSDL contiene la URL y el espacio de nombres del servicio, para que no pueda equivocarse. Por supuesto, hay otras cosas que pueden ir mal.

Ejemplo 12.16. Invocación de un método con argumentos equivocados

```
>>> wsdlFile =
'http://www.xmethods.net/sd/2001/TemperatureService.wsdl'
>>> server = WSDL.Proxy(wsdlFile)
>>> temperature = server.getTemp(27502)
❶
<Fault SOAP-ENV:Server: Exception while handling service request:
services.temperature.TempService.getTemp(int) -- no signature match>
❷
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 453, in
__call__
    return self.__r_call(*args, **kw)
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 475, in
__r_call
    self.__hd, self.__ma)
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 389, in
__call
    raise p
SOAPpy.Types.faultType: <Fault SOAP-ENV:Server: Exception while
handling service request:
services.temperature.TempService.getTemp(int) -- no signature match>
```

❶ ¿Vio el fallo? Es sutil: estámos invocando `server.getTemp` con un entero en lugar de una cadena. Como vio en la introspección del fichero WSDL, la función `getTemp()` de SOAP acepta un único argumento, `zipcode`, que debe ser unacadena. `WSDL.Proxy`*no* hará conversión de tipos de datos por usted; debe pasar los tipos exactos que espera el servidor.

❷ De nuevo, el servidor devuelve una Fault de SOAP y la parte legible del error nos da una pista del problema: estamos invocando la función `getTemp` con un valor entero, pero no hay definida una función con ese nombre que acepte un entero. En teoría, SOAP le permite *sobrecargar* funciones así que podría etner dos funciones en el mismo servicio SOAP con el mismo nombre y argumentos en mismo número, pero con tipo de diferente. Por eso es importante que los tipos de datos se ajusten de forma exacta y es la razón de que `WSDL.Proxy` no transforme los tipos. Si lo hiciera, ¡podría acabar llamando a una función completamente diferente! Buena suerte depurando eso en tal caso. Es mucho más fácil ser detallista con los tipos de datos y obtener un

fallo lo antes posible si los indicamos de forma errónea.

También es posible escribir código en Python que espere valores de retorno diferentes de los que devuelve realmente la función remota.

Ejemplo 12.17. Invocación de un método esperando una cantidad errónea de valores de retorno

```
>>> wsdlFile =
'http://www.xmethods.net/sd/2001/TemperatureService.wsdl'
>>> server = WSDL.Proxy(wsdlFile)
>>> (city, temperature) = server.getTemp(27502)  ❶
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: unpack non-sequence
```

❶ ¿Vio el fallo? `server.getTemp` sólo devuelve un valor, un decimal, pero hemos escrito código que asume que vamos a obtener dos valores e intenta asignarlos a dos variables diferentes. Esto no se registra con una falta de SOAP. En lo que concierne al servidor remoto nada ha ido mal. El error ha ocurrido *tras* completar la transacción SOAP, `WSDL.Proxy` devolvió un número decimal, y el intérprete Python intentó en local acomodar nuestra petición de dividirlo en dos variables diferentes. Como la función sólo devuelve un valor, obtenemos una excepción de Python al intentar partirlo, y no una Fault de SOAP.

¿Qué hay del servicio web de Google? El problema más común que he tenido con él es que olvido indicar adecuadamente la clave de la aplicación.

Ejemplo 12.18. Invocación de un método con un error específico a la aplicació

```
>>> from SOAPpy import WSDL
>>> server = WSDL.Proxy(r'/path/to/local/GoogleSearch.wsdl')
>>> results = server.doGoogleSearch('foo', 'mark', 0, 10, False, "", ❶
...     False, "", "utf-8", "utf-8")
<Fault SOAP-ENV:Server:                                              ❷
```

```
 Exception from service object: Invalid authorization key: foo:
 <SOAPpy.Types.structType detail at 14164616>:
 {'stackTrace':
  'com.google.soap.search.GoogleSearchFault: Invalid authorization
key: foo
   at com.google.soap.search.QueryLimits.lookUpAndLoadFromINSIfNeedBe(
     QueryLimits.java:220)
   at
com.google.soap.search.QueryLimits.validateKey(QueryLimits.java:127)
   at com.google.soap.search.GoogleSearchService.doPublicMethodChecks(
     GoogleSearchService.java:825)
   at com.google.soap.search.GoogleSearchService.doGoogleSearch(
     GoogleSearchService.java:121)
   at sun.reflect.GeneratedMethodAccessor13.invoke(Unknown Source)
   at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
   at java.lang.reflect.Method.invoke(Unknown Source)
   at org.apache.soap.server.RPCRouter.invoke(RPCRouter.java:146)
   at org.apache.soap.providers.RPCJavaProvider.invoke(
     RPCJavaProvider.java:129)
   at org.apache.soap.server.http.RPCRouterServlet.doPost(
     RPCRouterServlet.java:288)
   at javax.servlet.http.HttpServlet.service(HttpServlet.java:760)
   at javax.servlet.http.HttpServlet.service(HttpServlet.java:853)
   at
com.google.gse.HttpConnection.runServlet(HttpConnection.java:237)
   at com.google.gse.HttpConnection.run(HttpConnection.java:195)
   at
com.google.gse.DispatchQueue$WorkerThread.run(DispatchQueue.java:201)
Caused by: com.google.soap.search.UserKeyInvalidException: Key was of
wrong size.
   at com.google.soap.search.UserKey.<init>(UserKey.java:59)
   at com.google.soap.search.QueryLimits.lookUpAndLoadFromINSIfNeedBe(
     QueryLimits.java:217)
   ... 14 more
'}>
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 453, in
__call__
    return self.__r_call(*args, **kw)
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 475, in
__r_call
```

```
    self.__hd, self.__ma)
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 389, in
__call
    raise p
SOAPpy.Types.faultType: <Fault SOAP-ENV:Server: Exception from service
object:
Invalid authorization key: foo:
<SOAPpy.Types.structType detail at 14164616>:
{'stackTrace':
  'com.google.soap.search.GoogleSearchFault: Invalid authorization
key: foo
   at com.google.soap.search.QueryLimits.lookUpAndLoadFromINSIfNeedBe(
     QueryLimits.java:220)
   at
com.google.soap.search.QueryLimits.validateKey(QueryLimits.java:127)
   at com.google.soap.search.GoogleSearchService.doPublicMethodChecks(
     GoogleSearchService.java:825)
   at com.google.soap.search.GoogleSearchService.doGoogleSearch(
     GoogleSearchService.java:121)
   at sun.reflect.GeneratedMethodAccessor13.invoke(Unknown Source)
   at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
   at java.lang.reflect.Method.invoke(Unknown Source)
   at org.apache.soap.server.RPCRouter.invoke(RPCRouter.java:146)
   at org.apache.soap.providers.RPCJavaProvider.invoke(
     RPCJavaProvider.java:129)
   at org.apache.soap.server.http.RPCRouterServlet.doPost(
     RPCRouterServlet.java:288)
   at javax.servlet.http.HttpServlet.service(HttpServlet.java:760)
   at javax.servlet.http.HttpServlet.service(HttpServlet.java:853)
   at
com.google.gse.HttpConnection.runServlet(HttpConnection.java:237)
   at com.google.gse.HttpConnection.run(HttpConnection.java:195)
   at
com.google.gse.DispatchQueue$WorkerThread.run(DispatchQueue.java:201)
Caused by: com.google.soap.search.UserKeyInvalidException: Key was of
wrong size.
   at com.google.soap.search.UserKey.<init>(UserKey.java:59)
   at com.google.soap.search.QueryLimits.lookUpAndLoadFromINSIfNeedBe(
     QueryLimits.java:217)
   ... 14 more
'}>
```

❶ ¿Puede encontrar el fallo? No hay nada malo en la sintaxis de la llamada, o el número de los argumentos, o los tipos. El problema es específico de la aplicación: el primer argumento debería ser mi clave de la aplicación, pero `foo` no es una clave válida en Google.

❷ El servidor de Google responde con una Fault de SOAP y un mensaje de error increíblemente largo, que incluye una traza de pila de Java completa. Recuerde que *todos* los errores de SOAP se señalan mediante una Fault: los errores de configuración, de argumentos y los errores específicos de aplicaciones como éste. Enterrada en algún sitio está la información crucial: `Invalid authorization key: foo`.

Lecturas complementarias sobre solución de problemas con SOAP

- New developments for SOAPpy analiza algunos intentos de conectar a otro servicio SOAP que no funciona exactamente tal como se esperaba.

12.9. Resumen

Los servicios web SOAP son muy complicados. La especificación es muy ambiciosa e intenta cubrir muchos casos de uso de servicios web. Este capítulo ha tocado algunos de los casos más sencillos.

Antes de zambullirse en el siguiente capítulo, asegúrese de que se siente cómo haciendo todo esto:

- Conectar a un servidor SOAP e invocar métodos remotos
- Cargar un fichero WSDL y hacer introspección de métodos remotos
- Depurar llamadas a SOAP con trazas de comunicación
- Solucionar errores comunes relacionados con SOAP

Capítulo 13. Pruebas unitarias (*Unit Testing*)

13.1. Introducción a los números romanos

En capítulos interiores, se "sumergió" dando un vistazo rápido al código e intentándo comprenderlo lo más rápidamente posible. Ahora que tiene bastante Python a sus espaldas, vamos a retroceder y ver los pasos que se dan *antes* de escribir el código.

En los próximos capítulos vamos a escribir, depurar y optimizar una serie de funciones de utilidad para convertir números romanos a decimales, y viceversa. Vimos el mecanismo de construir y validar los números romanos en Sección 7.3, "Caso de estudio: números romanos", pero ahora retrocederemos y consideraremos lo que tomaría expandirlo en una utilidad de doble sentido.

Las reglas de los números romanos llevan a varias observaciones interesantes:

1. Sólo hay una manera correcta de representar un número en particular en romanos.
2. A la inversa también es cierto: sio una cadena de caracteres es un número romano válido, representa un número único (es decir sólo se puede leer de una manera).
3. Hay una cantidad limitada de números expresables en romanos, específicamente desde `1` a `3999`. (Los romanos tenían varias maneras de expresar números más grandes, por ejemplo poniendo una barra sobre un número representando que su valor normal debería multiplicarse por

`1000`, pero no vamos a tratar con eso. Para el propósito de este capítulo, estipularemos que los números romanos van de `1` a `3999`).
4. No hay manera de representar `0` en números romanos. (Sorprendentemente, los antiguos romanos no tenían concepto de `0` como número. Los números eran para contar cosas que tenías; ¿cómo ibas a contar lo que no tenías?)
5. No hay manera de representar números negativos en romanos.
6. No hay manera de representar fracciones o números no enteros en romanos.

Dado todo esto, ¿qué podríamos esperar de un conjunto de funciones para convertir de y a números romanos?

Requisitos de `roman.py`

1. `toRoman` debería devolver la representación en romanos de todos los enteros del `1` al `3999`.
2. `toRoman` debería fallar cuando se le dé un entero fuera del rango `1` al `3999`.
3. `toRoman` debería fallar cuando se le dé un número no entero.
4. `fromRoman` debería tomar un número romano válido y devolver el número que representa.
5. `fromRoman` debería fallar cuando se le dé un número romano no válido.
6. Si tomamos un número, lo convertimos a romanos, y luego de vuelta a un número, deberíamos acabar con el mismo número con que empezamos. Así que `fromRoman(toRoman(n)) == n` para todo `n` en `1..3999`.
7. `toRoman` debería devolver siempre un número romano usando letras mayúsculas.
8. `fromRoman` sólo debería aceptar números romanos en mayúsculas (es decir debería fallar si se le da una entrada en minúsculas).

Lecturas complementarias

- Este sitio cuenta más cosas sobre los números romanos, incluyendo una fascinante historia sobre la manera en que los romanos y otras civilizaciones los usaban (versión corta: descuidada e inconsistentemente).

13.2. Inmersión

Ahora que ya hemos definido completamente el comportamiento esperado de nuestras funciones de conversión, vamos a hacer algo un poco inesperado: vamos a escribir una batería de pruebas que ponga estas funciones contra las cuerdas y se asegure de que se comportan de la manera en que queremos. Ha leído bien: va a escribir código que pruebe código que aún no hemos escrito.

A esto se le llama pruebas unitarias (*unit testing*), ya que el conjunto de dos funciones de conversión se puede escribir y probar como una unidad, separado de cualquier programa más grande del que pueden formar parte más adelante. Python dispone de infraestructura para hacer pruebas unitarias: el módulo `unittest`, de nombre bastante apropiado.

`unittest` está incluido en Python 2.1 y posteriores. Los usuarios de Python 2.0 pueden descargarlo de `pyunit.sourceforge.net`.

La prueba unitaria es una parte importante de una estrategia de desarrollo centrada en las pruebas. Si escribe pruebas unitarias, es importante hacerlo pronto (preferiblemente antes de escribir el código que van a probar) y mantenerlas actualizadas con los cambios de código y de requisitos. Las pruebas unitarias no sustituyen a pruebas de mayor nivel funcional o de sistema, pero son importantes en todas las fases de desarrollo:

- Antes de escribir código, le fuerzan a detallar los requisitos de una manera útil.
- Mientras escribe el código, evitan un exceso de funcionalidad. Cuando pasan todos los casos de prueba, la función está completa.

- Cuando se hace refactorización de código, le aseguran que la nueva versión se comporta de la misma manera que la antigua.
- Cuando mantiene código, le ayuda a cubrirse las espaldas si alguien viene gritando que el último cambio hace fallar código ajeno ("Pero *señor*, todas las pruebas unitarias habían pasado correctamente antes de enviarlo...")
- Cuando escribe código en equipo, aumenta la confianza en que el código que está a punto de enviar no va a romper el de otras personas, porque puede ejecutar las pruebas unitarias de ellos antes. (He visto este tipo de cosas en momentos de prisas. Un equipo divide y asigna la tareas, todo el mundo toma los requisitos de su tarea, escribe pruebas unitarias y luego las comparte con el resto del equipo. De esa manera, nadie va demasiado lejos desarrollando código que no juegue correctamente con el de otros).

13.3. Presentación de `romantest.py`

Ésta es la batería de pruebas completa para las funciones de conversión de números romanos, que todavía no están escritas, pero en el futuro estarán en `roman.py`. No es inmediatamente obvio cómo encaja todo esto; ninguna de las clases o funciones hace referencia a las otras. Hay buenas razones para esto, como veremos en breve.

Ejemplo 13.1. `romantest.py`

Si aún no lo ha hecho, puede descargar éste ejemplo y otros usados en este libro.

```
"""Unit test for roman.py"""

import roman
import unittest

class KnownValues(unittest.TestCase):
    knownValues = ( (1, 'I'),
```

```
(2, 'II'),
(3, 'III'),
(4, 'IV'),
(5, 'V'),
(6, 'VI'),
(7, 'VII'),
(8, 'VIII'),
(9, 'IX'),
(10, 'X'),
(50, 'L'),
(100, 'C'),
(500, 'D'),
(1000, 'M'),
(31, 'XXXI'),
(148, 'CXLVIII'),
(294, 'CCXCIV'),
(312, 'CCCXII'),
(421, 'CDXXI'),
(528, 'DXXVIII'),
(621, 'DCXXI'),
(782, 'DCCLXXXII'),
(870, 'DCCCLXX'),
(941, 'CMXLI'),
(1043, 'MXLIII'),
(1110, 'MCX'),
(1226, 'MCCXXVI'),
(1301, 'MCCCI'),
(1485, 'MCDLXXXV'),
(1509, 'MDIX'),
(1607, 'MDCVII'),
(1754, 'MDCCLIV'),
(1832, 'MDCCCXXXII'),
(1993, 'MCMXCIII'),
(2074, 'MMLXXIV'),
(2152, 'MMCLII'),
(2212, 'MMCCXII'),
(2343, 'MMCCCXLIII'),
(2499, 'MMCDXCIX'),
(2574, 'MMDLXXIV'),
(2646, 'MMDCXLVI'),
(2723, 'MMDCCXXIII'),
(2892, 'MMDCCCXCII'),
```

```
                    (2975, 'MMCMLXXV'),
                    (3051, 'MMMLI'),
                    (3185, 'MMMCLXXXV'),
                    (3250, 'MMMCCL'),
                    (3313, 'MMMCCCXIII'),
                    (3408, 'MMMCDVIII'),
                    (3501, 'MMMDI'),
                    (3610, 'MMMDCX'),
                    (3743, 'MMMDCCXLIII'),
                    (3844, 'MMMDCCCXLIV'),
                    (3888, 'MMMDCCCLXXXVIII'),
                    (3940, 'MMMCMXL'),
                    (3999, 'MMMCMXCIX'))

    def testToRomanKnownValues(self):
        """toRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman.toRoman(integer)
            self.assertEqual(numeral, result)

    def testFromRomanKnownValues(self):
        """fromRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman.fromRoman(numeral)
            self.assertEqual(integer, result)

class ToRomanBadInput(unittest.TestCase):
    def testTooLarge(self):
        """toRoman should fail with large input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, 4000)

    def testZero(self):
        """toRoman should fail with 0 input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, 0)

    def testNegative(self):
        """toRoman should fail with negative input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, -1)

    def testNonInteger(self):
        """toRoman should fail with non-integer input"""
        self.assertRaises(roman.NotIntegerError, roman.toRoman, 0.5)
```

```
class FromRomanBadInput(unittest.TestCase):
    def testTooManyRepeatedNumerals(self):
        """fromRoman should fail with too many repeated numerals"""
        for s in ('MMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):
            self.assertRaises(roman.InvalidRomanNumeralError,
roman.fromRoman, s)

    def testRepeatedPairs(self):
        """fromRoman should fail with repeated pairs of numerals"""
        for s in ('CMCM', 'CDCD', 'XCXC', 'XLXL', 'IXIX', 'IVIV'):
            self.assertRaises(roman.InvalidRomanNumeralError,
roman.fromRoman, s)

    def testMalformedAntecedent(self):
        """fromRoman should fail with malformed antecedents"""
        for s in ('IIMXCC', 'VX', 'DCM', 'CMM', 'IXIV',
                  'MCMC', 'XCX', 'IVI', 'LM', 'LD', 'LC'):
            self.assertRaises(roman.InvalidRomanNumeralError,
roman.fromRoman, s)

class SanityCheck(unittest.TestCase):
    def testSanity(self):
        """fromRoman(toRoman(n))==n for all n"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            result = roman.fromRoman(numeral)
            self.assertEqual(integer, result)

class CaseCheck(unittest.TestCase):
    def testToRomanCase(self):
        """toRoman should always return uppercase"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            self.assertEqual(numeral, numeral.upper())

    def testFromRomanCase(self):
        """fromRoman should only accept uppercase input"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            roman.fromRoman(numeral.upper())
            self.assertRaises(roman.InvalidRomanNumeralError,
```

```
                                roman.fromRoman, numeral.lower())

if __name__ == "__main__":
    unittest.main()
```

Lecturas complementarias

- La página de PyUnit tiene una discusión a fondo del uso de la infraestructura de `unittest`, incluyendo características avanzadas que no cubrimos en este capítulo.
- Las FAQ de PyUnit explican por qué los casos de prueba se almacenan aparte del código sobre el que prueban.
- La *Referencia de bibliotecas de Python* expone el módulo `unittest`.
- ExtremeProgramming.org argumenta las razones por las que debe escribir pruebas unitarias.
- El Portland Pattern Repository tiene una discusión en marcha sobre pruebas unitarias, que incluyen una definicióne stándar, razones por las que debería escribir primero las pruebas unitarias, y varios casos de estudio en profundidad.

13.4. Prueba de éxito

La parte más fundamental de una prueba unitaria es la construcción de casos de prueba individuales. Un caso de prueba responde a una única pregunta sobre el código que está probando.

Un caso de prueba debería ser capaz de...

- ...ejecutarse de forma autónoma, sin interacción humana. Las pruebas unitarias implican automatización
- ...determinar por sí solo si la función que está probando es correcta o falla, sin que un humano deba interpretar los resultados.
- ...ejecutarse de forma aislada y separada de otros casos de prueba (incluso si prueban la misma función). Cada caso de prueba es una isla.

Dado esto, construyamos el primer caso de prueba. Tenemos el siguiente requisito:

1. `toRoman` debería devolver la representación en romanos de todos los enteros del `1` al `3999`.

Ejemplo 13.2. `testToRomanKnownValues`

```
class KnownValues(unittest.TestCase):                                    ❶
    knownValues = ( (1, 'I'),
                    (2, 'II'),
                    (3, 'III'),
                    (4, 'IV'),
                    (5, 'V'),
                    (6, 'VI'),
                    (7, 'VII'),
                    (8, 'VIII'),
                    (9, 'IX'),
                    (10, 'X'),
                    (50, 'L'),
                    (100, 'C'),
                    (500, 'D'),
                    (1000, 'M'),
                    (31, 'XXXI'),
                    (148, 'CXLVIII'),
                    (294, 'CCXCIV'),
                    (312, 'CCCXII'),
                    (421, 'CDXXI'),
                    (528, 'DXXVIII'),
                    (621, 'DCXXI'),
                    (782, 'DCCLXXXII'),
                    (870, 'DCCCLXX'),
                    (941, 'CMXLI'),
                    (1043, 'MXLIII'),
                    (1110, 'MCX'),
                    (1226, 'MCCXXVI'),
                    (1301, 'MCCCI'),
                    (1485, 'MCDLXXXV'),
                    (1509, 'MDIX'),
                    (1607, 'MDCVII'),
```

```
                (1754, 'MDCCLIV'),
                (1832, 'MDCCCXXXII'),
                (1993, 'MCMXCIII'),
                (2074, 'MMLXXIV'),
                (2152, 'MMCLII'),
                (2212, 'MMCCXII'),
                (2343, 'MMCCCXLIII'),
                (2499, 'MMCDXCIX'),
                (2574, 'MMDLXXIV'),
                (2646, 'MMDCXLVI'),
                (2723, 'MMDCCXXIII'),
                (2892, 'MMDCCCXCII'),
                (2975, 'MMCMLXXV'),
                (3051, 'MMMLI'),
                (3185, 'MMMCLXXXV'),
                (3250, 'MMMCCL'),
                (3313, 'MMMCCCXIII'),
                (3408, 'MMMCDVIII'),
                (3501, 'MMMDI'),
                (3610, 'MMMDCX'),
                (3743, 'MMMDCCXLIII'),
                (3844, 'MMMDCCCXLIV'),
                (3888, 'MMMDCCCLXXXVIII'),
                (3940, 'MMMCMXL'),
                (3999, 'MMMCMXCIX'))                          ❷

    def testToRomanKnownValues(self):                         ❸
        """toRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman.toRoman(integer)                   ❹ ❺
            self.assertEqual(numeral, result)                 ❻
```

❶ Para escribir un caso de prueba, primero derivamos la clase `TestCase` del módulo `unittest`. Esta clase proporciona muchos métodos útiles que puede invocar desde el caso de prueba para comprobar condiciones específicas.

❷ Una lista de pares entero/número que he verificado manualmente. Incluye los diez números más pequeños, los más grandes, cada número que se traduce a un único carácter numeral romano, y una muestra al azar de otros números válidos. El objetivo de una prueba unitaria no es probar cada caso posible, sino una muestra representativa.

❸ Cada prueba individual es un método separado, que no debe tomar parámetros ni devolver valores. Si el método sale de forma normal sin lanzar una excepción, se considera que ha pasado la prueba; si el método lanza una excepción, se considera que ha fallado la prueba.

❹ Aquí llamamos a la función `toRoman` (bien, no se ha escrito aún la función, pero una vez que lo esté, esta es la línea que la invocaría). Observe que hemos definido el API de la función `toRoman`: debe tomar un entero (el número a convertir) y devolver una cadena (la representación en números romanos). Si el API es diferente de eso, esta prueba se considera fallida.

❺ Observe también que no estamos atrapando las excepciones cuando llamamos a `toRoman`. Esto es intencionado. `toRoman` no debería lanzar ninguna excepción cuando se le llama con una entrada válida, y estos valores de entrada son todos válidos. Si `toRoman` lanza una excepción, se considera que esta prueba es fallida.

❻ Asumiendo que se haya definido e invocado correctamente la función `toRoman`, que haya terminado con éxito y devuelto un valor, el último paso es comprobar si devuelve el valor *correcto*. Ésta es una operación común y la clase `TestCase` proporciona un método para comprobar si dos valores son iguales, `assertEqual`. Si el resultado devuelto por `toRoman` (`result`) no coincide con el valor conocido que esperábamos (`numeral`), `assertEqual` lanzará una excepción y la prueba fallará. Si los dos valores son iguales `assertEqual` no hará nada. Si cada valor devuelto por `toRoman` coincide con el esperado, `assertEqual` no lanzará ninguna excepción, así que `testToRomanKnownValues` saldrá de forma normal eventualmente, lo que significará que `toRoman` ha pasado la prueba.

13.5. Prueba de fallo

No es suficiente probar que las funciones tienen éxito cuando se les pasa valores correctos; también debe probar que fallarán si se les da una entrada incorrecta. Y no sólo cualquier tipo de fallo; deben fallar de la manera esperada.

Recuerde los otros requisitos de `toRoman`:

2. `toRoman` debería fallar cuando se le dé un entero fuera del rango `1` al `3999`.
3. `toRoman` debería fallar cuando se le dé un número no entero.

En Python las funciones indican los fallos lanzando excepciones y el módulo `unittest` proporciona métodos para probar si una función lanza una excepción en particular cuando se le da una entrada incorrecta.

Ejemplo 13.3. Prueba de `toRoman` con entrada incorrecta

```
class ToRomanBadInput(unittest.TestCase):
    def testTooLarge(self):
        """toRoman should fail with large input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, 4000)
❶

    def testZero(self):
        """toRoman should fail with 0 input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, 0)
❷

    def testNegative(self):
        """toRoman should fail with negative input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, -1)

    def testNonInteger(self):
        """toRoman should fail with non-integer input"""
        self.assertRaises(roman.NotIntegerError, roman.toRoman, 0.5)
❸
```

❶ La clase `TestCase` de `unittest` proporciona el método `assertRaises`, que toma los siguientes argumentos: la excepción que esperamos, la función que vamos a probar, y los argumentos que le queremos pasar a esa función (si la función que vamos a probar toma más de un argumento, se los pasamos todos a `assertRaises`, en orden, y se los pasará directamente a la función que queremos probar). Preste mucha atención a lo que estamos haciendo aquí: en

lugar de llamar directamente a `toRoman` y comprobar manualmente que lanza una excepción en particular (encerrándolo en un bloque `try...except`, `assertRaises` lo encapsula por nosotros. Todo lo que hacemos es darle la excepción (`roman.OutOfRangeError`), la función (`toRoman`) y los argumentos de `toRoman` (`4000`), y `assertRaises` se encarga de invocar a `toRoman` y comprueba para asegurarse de que lanza `roman.OutOfRangeError`. (Observe también que está pasando la función `toRoman` en sí como un argumento; no la está invocando y no está pasando su nombre como una cadena. ¿Le he mencionado últimamente lo útil que es que todo en Python sea un objeto, incluyendo funciones y excepciones?)

❷ Además de las pruebas de números que son demasiado grandes, necesita probar los números que son demasiado pequeños. Recuerde, los números romanos no pueden expresar `0` o números negativos, así que tenemos un caso de prueba para cada uno (`testZero` y `testNegative`). En `testZero` estamos probando que `toRoman` lance una excepción `roman.OutOfRangeError` cuando se le pasa `0`; si *no* lanza una `roman.OutOfRangeError` (bien porque devuelva un valor, o porque lanza otra excepción), se considerará que ha fallado esta prueba.

❸ El tercer requisito especifica que `toRoman` no puede aceptar un número que no sea entero, así que aquí probamos para aseguranos de que `toRoman` lanza una excepción `roman.NotIntegerError` cuando se le llama con `0.5`. Si `toRoman` no lanza una `roman.NotIntegerError` se considerará que ha fallado esta prueba.

Los dos siguientes requisitos son similares a los tres primeros, excepto que se aplican a `fromRoman` en lugar de a `toRoman`:

4. `fromRoman` debería tomar un número romano válido y devolver el número que representa.
5. `fromRoman` debería fallar cuando se le dé un número romano no válido.

El 4º requisito se trata de la misma manera que el requisito 1, iterando sobre una muestra de valores conocidos y probándolos por turno. El 5º requisito se

trata de la misma manera que el 2º y el 3º, probando una serie de entradas incorrectas y asegurándose de que `fromRoman` lanza la excepción adecuada.

Ejemplo 13.4. Prueba de entradas incorrectas a `fromRoman`

```
class FromRomanBadInput(unittest.TestCase):
    def testTooManyRepeatedNumerals(self):
        """fromRoman should fail with too many repeated numerals"""
        for s in ('MMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):
            self.assertRaises(roman.InvalidRomanNumeralError,
roman.fromRoman, s) ❶

    def testRepeatedPairs(self):
        """fromRoman should fail with repeated pairs of numerals"""
        for s in ('CMCM', 'CDCD', 'XCXC', 'XLXL', 'IXIX', 'IVIV'):
            self.assertRaises(roman.InvalidRomanNumeralError,
roman.fromRoman, s)

    def testMalformedAntecedent(self):
        """fromRoman should fail with malformed antecedents"""
        for s in ('IIMXCC', 'VX', 'DCM', 'CMM', 'IXIV',
                  'MCMC', 'XCX', 'IVI', 'LM', 'LD', 'LC'):
            self.assertRaises(roman.InvalidRomanNumeralError,
roman.fromRoman, s)
```

❶ No hay mucho nuevo que decir aquí; el patrón es exactamente el mismo que usamos para probar la entrada incorrecta en `toRoman`. Señalaré brevemente que tenemos otra excepción: `roman.InvalidRomanNumeralError`. Esto hace un total de tras excepciones propias que necesitaremos definir en `roman.py` (junto con `roman.OutOfRangeError` y `roman.NotIntegerError`). Veremos cómo definir estas excepciones cuando empecemos a escribir `roman.py`, más adelante en este capítulo.

13.6. Pruebas de cordura

A menudo se encontrará conque un código unitario contiene un conjunto de funciones recíprocas normalmente en forma de funciones de conversión en que una convierte de A a B y la otra de B a A. En estos casos es útil crear “pruebas

de cordura" (*sanity checks* para asegurarse de que puede convertir de A a B y de vuelta a A sin perder precisión, incurrir en errores de redondeo o encontrar ningún otro tipo de fallo.

Considere este requisito:

6. Si tomamos un número, lo convertimos a romanos, y luego de vuelta a un número, deberíamos acabar con el mismo número con que empezamos. Así que `fromRoman(toRoman(n)) == n` para todo `n` en `1..3999`.

Ejemplo 13.5. Prueba de `toRoman` frente a `fromRoman`

```
class SanityCheck(unittest.TestCase):
    def testSanity(self):
        """fromRoman(toRoman(n))==n for all n"""
        for integer in range(1, 4000):        ❶ ❷
            numeral = roman.toRoman(integer)
            result = roman.fromRoman(numeral)
            self.assertEqual(integer, result) ❸
```

❶ Hemos visto antes la función `range`, pero aquí la invocamos con dos argumentos, que devuelven una lista de enteros que empiezan en el primero (`1`) y avanzan consecutivamente hasta el segundo argumento (`4000`) *sin incluirlo*. O sea, `1..3999`, que es el rango válido para convertir a números romanos.

❷ Sólo quiero mencionar de pasada que `integer` no es una palabra reservada de Python; aquí es un nombre de variable como cualquier otro.

❸ La lógica real de la prueba es muy simple: tomamos un número (`integer`), lo convertimos en número romano (`numeral`), luego lo convertimos de vuelta en número (`result`) y nos aseguramos de que es el mismo número con que empezamos. Si no, `assertEqual` lanzará una excepción y se considerará inmediatamente la prueba fallida. Si todos los números coinciden, `assertEqual` volverá siempre silenciosa, lo mismo hará eventualmente el

método `testSanity`, y se considerará pasada la prueba.

Los dos últimos requisitos son diferentes de los otros porque ambos son arbitrarios y tribiales:

7. `toRoman` debería devolver siempre un número romano usando letras mayúsculas.
8. `fromRoman` sólo debería aceptar números romanos en mayúsculas (es decir debería fallar si se le da una entrada en minúsculas).

Ciertamente, son bastante arbitrarios. Podríamos por ejemplo haber estipulado que `fromRoman` acepte entradas en minúsculas y en una mezcla de ambas. Pero no son completamente arbitrarias; si `toRoman` devuelve siempre una salida en mayúsculas, entonces `fromRoman` debe al menos aceptar entrada en mayúsculas, o fallaría la "prueba de cordura" (6º requisito). El hecho de que *sólo* acepte entradas en mayúscula es arbitrario, pero como le dirá cualquier integrador de sistemas, la caja de las letras[15] siempre importa así que merece la pena especificar el comportamiento de entrada. Y si vale la pena especificarlo, también probarlo.

Ejemplo 13.6. Pruebas de mayúsculas

```
class CaseCheck(unittest.TestCase):
    def testToRomanCase(self):
        """toRoman should always return uppercase"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            self.assertEqual(numeral, numeral.upper())          ❶

    def testFromRomanCase(self):
        """fromRoman should only accept uppercase input"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            roman.fromRoman(numeral.upper())                    ❷ ❸
            self.assertRaises(roman.InvalidRomanNumeralError,
                              roman.fromRoman, numeral.lower())  ❹
```

❶ Lo más interesante de este caso de prueba son todas las cosas que no prueba. No prueba si el valor devuelto por `toRoman` es correcto o incluso consistente; estas cuestiones las resuelven otros casos de prueba. Tenemos un caso de prueba sólo para probar las mayúsculas. Podría tentarle combinar esto con la prueba de cordura, ya que ambas pasan por todo el rango de valores y llaman a `toRoman`.[16] Pero eso violaría una de las reglas fundamentales: cada caso de prueba debería responder sólo una cuestión. Imagine que combinase este prueba de mayúsculas con la de cordura, y que falla el caso de prueba. Necesitaríamos hacer más análisis para averiguar qué parte del caso de prueba fallo para determinar el problema. Si necesita analizar los resultados de las pruebas unitarias sólo para averiguar qué significan, es un caso seguro de que ha desarrollado mal sus casos de prueba.

❷ Aquí se puede aprender una lección similar: incluso aunque "sepa" que `toRoman` siempre devuelve mayúsculas, estamos convirtiendo explícitamente su valor de retorno a mayúsculas aquí para probar que `fromRoman` acepta entrada en mayúsculas. ¿Por qué? Porque el hecho de que `toRoman` siempre devuelve mayúsculas es un requisito independiente. Si cambiamos ese requisito a, por ejemplo, que siempre devuelva minúsculas, habría que cambiar la prueba `testToRomanCase`, pero este caso debería seguir funcionando. Ésta es otra de las reglas fundamentales: cada prueba debe ser capaz de funcionar aisladamente de las otras. Cada caso de prueba es una isla.

❸ Observe que no estamos asignando el valor de retorno de `fromRoman` a nada. Esto es sintaxis válida en Python; si una función devuelve un valor pero nadie está a la escucha, Python simplemente descarta el valor devuelto. En este caso, es lo que queremos. Este caso de prueba no tiene nada que ver con el valor de retorno; simplemente comprueba que `fromRoman` acepta la entrada en mayúsculas sin lanzar ninguna excepción.

❹ Esta línea es complicada, pero es muy similar a lo que hicimos en las pruebas `ToRomanBadInput` y `FromRomanBadInput`. Estamos probando para aseguranos de que llamar a una función en particular (`roman.fromRoman`) con un valor en

particular (`numeral.lower()`, la versión en minúsculas del valor en romanos actual en el bucle) lanza una excepción en particular (`roman.InvalidRomanNumeralError`). Si lo hace (cada vez en el bucle), la prueba es válida; si al menos una vez hace cualquier otra cosa (como lanzar una excepción diferente, o devolver un valor sin lanzar una excepción), la prueba falla.

En el siguiente capítulo, veremos cómo escribir código que pasa estas pruebas.

Footnotes

[15] si están en mayúsculas o minúsculas

[16] “puedo resistirlo todo menos la tentación”--Oscar Wilde

Capítulo 14. Programación *Test-First*[17]

14.1. `roman.py`, fase 1

Ahora que están terminadas las pruebas unitarias, es hora de empezar a escribir el código que intentan probar esos casos de prueba. Vamos a hacer esto por etapas, para que pueda ver fallar todas las pruebas, y verlas luego pasar una por una según llene los huecos de `roman.py`.

Ejemplo 14.1. `roman1.py`

Este fichero está disponible en `py/roman/stage1/` en el directorio de ejemplos.

Si aún no lo ha hecho, puede descargar éste ejemplo y otros usados en este libro.

```
"""Convert to and from Roman numerals"""

#Define exceptions
class RomanError(Exception): pass                ❶
class OutOfRangeError(RomanError): pass          ❷
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass ❸

def toRoman(n):
    """convert integer to Roman numeral"""
    pass                                         ❹

def fromRoman(s):
    """convert Roman numeral to integer"""
    pass
```

❶ Así es como definimos nuestras excepciones propias en Python. Las excepciones son clases, y podemos crear las nuestras propias derivando las excepciones existentes. Se recomienda encarecidamente (pero no se precisa) que derive `Exception`, que es la clase base de la que heredan todas las excepciones que incorpora Python. Aquí defino `RomanError` (que hereda de `Exception`) para que actúe de clase base para todas las otras excepciones que seguirán. Esto es cuestión de estilo; podría igualmente haber hecho que cada excepción derivase directamente de la clase `Exception`.

❷ Las excepciones `OutOfRangeError` y `NotIntegerError` se usarán en `toRoman` para indicar varias formas de entrada inválida, como se especificó en `ToRomanBadInput`.

❸ La excepción `InvalidRomanNumeralError` se usará en `fromRoman` para indicar entrada inválida, como se especificó en `FromRomanBadInput`.

❹ En esta etapa, queremos definir la API de cada una de nuestras funciones, pero no programarlas aún, así que las dejamos vacías usando la palabra reservada de Python `pass`.

Ahora el momento que todos esperan (redoble de tambor, por favor): finalmente vamos a ejecutar la prueba unitaria sobre este pequeño módulo vacío. En este momento deberían fallar todas las pruebas. De hecho, si alguna prueba pasa en la fase 1, deberíamos volver a `romantest.py` y evaluar por qué programamos una prueba tan inútil que pasa en funciones que no hacen nada.

Ejecute `romantest1.py` con la opción `-v`, que le dará una salida más prolija para que pueda ver exactamente qué sucede mientras se ejecuta cada caso. Con algo de suerte, su salida debería ser como ésta:

Ejemplo 14.2. Salida de `romantest1.py` frente a `roman1.py`

```
fromRoman should only accept uppercase input ... ERROR
toRoman should always return uppercase ... ERROR
fromRoman should fail with malformed antecedents ... FAIL
fromRoman should fail with repeated pairs of numerals ... FAIL
fromRoman should fail with too many repeated numerals ... FAIL
```

```
fromRoman should give known result with known input ... FAIL
toRoman should give known result with known input ... FAIL
fromRoman(toRoman(n))==n for all n ... FAIL
toRoman should fail with non-integer input ... FAIL
toRoman should fail with negative input ... FAIL
toRoman should fail with large input ... FAIL
toRoman should fail with 0 input ... FAIL

======================================================================
ERROR: fromRoman should only accept uppercase input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 154, in
testFromRomanCase
    roman1.fromRoman(numeral.upper())
AttributeError: 'None' object has no attribute 'upper'
======================================================================
ERROR: toRoman should always return uppercase
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 148, in
testToRomanCase
    self.assertEqual(numeral, numeral.upper())
AttributeError: 'None' object has no attribute 'upper'
======================================================================
FAIL: fromRoman should fail with malformed antecedents
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 133, in
testMalformedAntecedent
    self.assertRaises(roman1.InvalidRomanNumeralError,
roman1.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with repeated pairs of numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 127, in
testRepeatedPairs
```

```
    self.assertRaises(roman1.InvalidRomanNumeralError,
roman1.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with too many repeated numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 122, in
testTooManyRepeatedNumerals
    self.assertRaises(roman1.InvalidRomanNumeralError,
roman1.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 99, in
testFromRomanKnownValues
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
======================================================================
FAIL: toRoman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 93, in
testToRomanKnownValues
    self.assertEqual(numeral, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: I != None
======================================================================
FAIL: fromRoman(toRoman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 141, in
testSanity
```

```
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
======================================================================
FAIL: toRoman should fail with non-integer input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 116, in
testNonInteger
    self.assertRaises(roman1.NotIntegerError, roman1.toRoman, 0.5)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: NotIntegerError
======================================================================
FAIL: toRoman should fail with negative input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 112, in
testNegative
    self.assertRaises(roman1.OutOfRangeError, roman1.toRoman, -1)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
======================================================================
FAIL: toRoman should fail with large input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 104, in
testTooLarge
    self.assertRaises(roman1.OutOfRangeError, roman1.toRoman, 4000)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
======================================================================
FAIL: toRoman should fail with 0 input
```

❶

```
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 108, in
testZero
    self.assertRaises(roman1.OutOfRangeError, roman1.toRoman, 0)
```

```
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
❷
----------------------------------------------------------------------
Ran 12 tests in 0.040s
❸

FAILED (failures=10, errors=2)
❹
```

❶ La ejecución del script lanza `unittest.main()`, que a su vez llama a cada caso de prueba, que es lo mismo que decir cada método definido dentro de `romantest.py`. Por cada caso de uso, imprime la `cadena de documentación` del método y si ha pasado o fallado la prueba. Como esperábamos, no ha pasado ninguna de ellas.

❷ Por cada prueba fallida, `unittest` imprime la información de traza que muestra qué sucedió. En este caso, la llamada a `assertRaises` (llamada también `failUnlessRaises`) lanzó una `AssertionError` porque esperaba que `toRoman` lanzase una `OutOfRangeError` y no lo hizo.

❸ Tras los detalles, `unittest` muestra un resumen de cuántas pruebas ha realizado y cuánto tiempo le ha tomado.

❹ En general, la prueba unitaria ha fallado porque no ha pasado al menos uno de los casos de prueba. Cuando un caso de prueba no pasa, `unittest` distingue entre fallos y errores. Un fallo es una llamada a un método `assertXYZ`, como `assertEqual` o `assertRaises`, que falla porque la condición asertada no es cierta o no se lanzó la excepción esperada. Un error es cualquier otro tipo de excepción lanzada en el código que está probando o del propio caso unitario de prueba. Por ejemplo, el método `testFromRomanCase` ("`fromRoman` sólo debería aceptar entrada en mayúsculas") dejó un error, porque la llamada a `numeral.upper()` lanzó una excepción `AttributeError`, ya que se suponía que `toRoman` debería devolver una cadena pero no lo hizo. Pero `testZero` ("`toRoman` debería fallar con una entrada 0") dejó un fallo, ya que la llamada a `fromRoman` no lanzó la excepción

`InvalidRomanNumeral` que estaba esperando `assertRaises`.

Footnotes

[17] El nombre de esta técnica se toma de la Programación Extrema (o XP)

14.2. `roman.py`, fase 2

Ahora que tenemos preparado el marco de trabajo del módulo `roman`, es hora de empezar a escribir código y pasar algunos casos de prueba.

Ejemplo 14.3. `roman2.py`

Este fichero está disponible en `py/roman/stage2/` dentro del directorio de ejemplos.

Si aún no lo ha hecho, puede descargar éste ejemplo y otros usados en este libro.

```
"""Convert to and from Roman numerals"""

#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Define digit mapping
romanNumeralMap = (('M',  1000), ❶
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
```

```
                    ('IV', 4),
                    ('I',  1))

def toRoman(n):
    """convert integer to Roman numeral"""
    result = ""
    for numeral, integer in romanNumeralMap:
        while n >= integer:      ❷
            result += numeral
            n -= integer
    return result

def fromRoman(s):
    """convert Roman numeral to integer"""
    pass
```

❶ `romanNumeralMap` es una tupla de tuplas que define tres cosas:

1. La representación en caracteres de los números romanos más básicos. Observe que no son sólo los números romanos de un solo carácter; también se definen pares de dos caracteres como `CM` (“cien menos que mil”); esto hará más adelante el código de `toRoman` más simple.
2. El orden de los números romanos. Están listados en orden de valor descendente, desde `M` hasta `I`.
3. El valor de cada número romano. Cada tupla interna es un par (*`numeral`*, *`valor`*).

❷ Aquí es donde compensa nuestra rica estructura de datos, porque no necesitamos ninguna lógica especial para tratar la regla de sustracción. Para convertir a números romanos, simplemente iteramos sobre `romanNumeralMap` buscando el valor entero más grande menor o igual que la entrada. Una vez encontrado, añadimos la representación en romanos del número al final de la salida, restamos el valor entero de la entrada, encerar, pulir, repetir.

Ejemplo 14.4. Cómo funciona `toRoman`

Si no tiene claro cómo funciona `toRoman`, añada una sentencia `print` al final del bucle `while`:

```
        while n >= integer:
            result += numeral
            n -= integer
            print 'restando', integer, 'de la entrada,
añadiendo',numeral, 'a la salida'
>>> import roman2
>>> roman2.toRoman(1424)
restando 1000 de la entrada, añadiendo M a la salida
restando 400 de la entrada, añadiendo CD a la salida
restando 10 de la entrada, añadiendo X a la salida
restando 10 de la entrada, añadiendo X a la salida
restando 4 de la entrada, añadiendo IV a la salida
'MCDXXIV'
```

Parece que `toRoman` funciona, al menos en esta comprobación manual. Pero, ¿pasará la prueba unitaria? Bien no, al menos no del todo.

Ejemplo 14.5. Salida de `romantest2.py` frente a `roman2.py`

Recuerde ejecutar `romantest2.py` con la opción `-v` para activar el modo prolijo.

```
fromRoman should only accept uppercase input ... FAIL
toRoman should always return uppercase ... ok                  ❶
fromRoman should fail with malformed antecedents ... FAIL
fromRoman should fail with repeated pairs of numerals ... FAIL
fromRoman should fail with too many repeated numerals ... FAIL
fromRoman should give known result with known input ... FAIL
toRoman should give known result with known input ... ok        ❷
fromRoman(toRoman(n))==n for all n ... FAIL
toRoman should fail with non-integer input ... FAIL            ❸
toRoman should fail with negative input ... FAIL
toRoman should fail with large input ... FAIL
toRoman should fail with 0 input ... FAIL
```

❶ Efectivamente, `toRoman` devuelve siempre mayúsculas, porque `romanNumeralMap` define las representaciones numéricas romanas como mayúsculas. Así que esta prueba pasa.

❷ Aquí están las grandes noticias: esta versión de `toRoman` pasa la prueba de valores conocidos. Recuerde, no es exhaustiva, pero pone a la función contra las cuerdas con una cierta varidad de entradas correctas, incluyendo entradas

que producen cada número romano de un solo carácter, la entrada más grande posible (`3999`), y la entrada que produce el número romano posible más largo (`3888`). Llegados a este punto, podemos confiar razonablemente en que la función es correcta con cualquier valor de entrada correcto que le pasemos.

❸ Sin embargo, la función no "trabaja" con valores incorrectos; falla todas las pruebas de entrada incorrecta. Esto tiene sentido, porque no hemos incluido ninguna comprobación en ese sentido. Estas pruebas buscan el lanzamiento de excepciones específicas (mediante `assertRaises`), y no las estamos lanzando. Lo haremos en la siguiente etapa.

Aquí está el resto de la salida de la prueba unitaria, listando los detalles de todos los fallos. Lo hemos reducido a 10.

```
======================================================================
FAIL: fromRoman should only accept uppercase input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 156, in
testFromRomanCase
    roman2.fromRoman, numeral.lower())
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with malformed antecedents
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 133, in
testMalformedAntecedent
    self.assertRaises(roman2.InvalidRomanNumeralError,
roman2.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with repeated pairs of numerals
```

```
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 127, in
testRepeatedPairs
    self.assertRaises(roman2.InvalidRomanNumeralError,
roman2.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with too many repeated numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 122, in
testTooManyRepeatedNumerals
    self.assertRaises(roman2.InvalidRomanNumeralError,
roman2.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 99, in
testFromRomanKnownValues
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
======================================================================
FAIL: fromRoman(toRoman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 141, in
testSanity
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
======================================================================
FAIL: toRoman should fail with non-integer input
```

```
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 116, in
testNonInteger
    self.assertRaises(roman2.NotIntegerError, roman2.toRoman, 0.5)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: NotIntegerError
======================================================================
FAIL: toRoman should fail with negative input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 112, in
testNegative
    self.assertRaises(roman2.OutOfRangeError, roman2.toRoman, -1)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
======================================================================
FAIL: toRoman should fail with large input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 104, in
testTooLarge
    self.assertRaises(roman2.OutOfRangeError, roman2.toRoman, 4000)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
======================================================================
FAIL: toRoman should fail with 0 input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 108, in
testZero
    self.assertRaises(roman2.OutOfRangeError, roman2.toRoman, 0)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
----------------------------------------------------------------------
Ran 12 tests in 0.320s

FAILED (failures=10)
```

14.3. `roman.py`, fase 3

Ahora que `toRoman` se comporta correctamente con entradas correctas (enteros del `1` al `3999`), es hora de hacer que se comporte correctamente con entradas incorrectas (todo lo demás).

Ejemplo 14.6. `roman3.py`

Este fichero está disponible en `py/roman/stage3/` dentro del directorio de ejemplos.

Si aún no lo ha hecho, puede descargar éste ejemplo y otros usados en este libro.

```
"""Convert to and from Roman numerals"""

#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Define digit mapping
romanNumeralMap = (('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))

def toRoman(n):
    """convert integer to Roman numeral"""
```

```
    if not (0 < n < 4000):
❶
        raise OutOfRangeError, "number out of range (must be 1..3999)"
❷
    if int(n) <> n:
❸
        raise NotIntegerError, "non-integers can not be converted"

    result = ""
❹
    for numeral, integer in romanNumeralMap:
        while n >= integer:
            result += numeral
            n -= integer
    return result

def fromRoman(s):
    """convert Roman numeral to integer"""
    pass
```

❶ Éste es un interesante atajo Pythonico: múltiples comparaciones simultáneas. Es equivalente a `if not ((0 < n) and (n < 4000))`, pero es mucho más fácil de leer. Es la comprobación de rango, y debería capturar entradas que sean demasiado grandes, negativas o cero.

❷ Lanzamos excepciones nosotros mismos con la sentencia `raise`. Podemos lanzar cualquiera de las excepciones que incorpora el lenguaje, o las propias que hemos definido. El segundo parámetro, el mensaje de error, es opcional; si lo damos, se muestra en el volcado de pila que se imprime en caso de no tratar la excepción.

❸ Ésta es la prueba de no enteros. Los números que no son enteros no se pueden convertir en romanos.

❹ El resto de la función queda igual.

Ejemplo 14.7. Veamos a `toRoman` tratando entradas incorrectas

```
>>> import roman3
>>> roman3.toRoman(4000)
Traceback (most recent call last):
```

```
  File "<interactive input>", line 1, in ?
  File "roman3.py", line 27, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999)
>>> roman3.toRoman(1.5)
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
  File "roman3.py", line 29, in toRoman
    raise NotIntegerError, "non-integers can not be converted"
NotIntegerError: non-integers can not be converted
```

Ejemplo 14.8. Salida ed `romantest3.py` frente a `roman3.py`

```
fromRoman should only accept uppercase input ... FAIL
toRoman should always return uppercase ... ok
fromRoman should fail with malformed antecedents ... FAIL
fromRoman should fail with repeated pairs of numerals ... FAIL
fromRoman should fail with too many repeated numerals ... FAIL
fromRoman should give known result with known input ... FAIL
toRoman should give known result with known input ... ok ❶
fromRoman(toRoman(n))==n for all n ... FAIL
toRoman should fail with non-integer input ... ok        ❷
toRoman should fail with negative input ... ok           ❸
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
```

❶ `toRoman` sigue pasando la prueba de valores conocidos, lo cual es reconfortante. Todas las pruebas que se pasaron en la fase 2 siguen haciéndolo, de manera que el código nuevo no ha roto nada.

❷ Más excitante es el hecho de que ahora pasan todas las pruebas de entrada incorrecta. Esta prueba, `testNonInteger`, pasa debido a la comprobación `int(n) <> n`. Cuando se le pasa un número que no es entero a `toRoman`, la comprobación `int(n) <> n` lo advierte y lanza la excepción `NotIntegerError`, que es lo que espera `testNonInteger`.

❸ Esta prueba, `testNegative`, pasa debido a la comprobación `not (0 < n < 4000)`, que lanza una excepción `OutOfRangeError` que es lo que buscaba `testNegative`.

```
======================================================================
```

```
FAIL: fromRoman should only accept uppercase input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 156, in
testFromRomanCase
    roman3.fromRoman, numeral.lower())
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with malformed antecedents
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 133, in
testMalformedAntecedent
    self.assertRaises(roman3.InvalidRomanNumeralError,
roman3.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with repeated pairs of numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 127, in
testRepeatedPairs
    self.assertRaises(roman3.InvalidRomanNumeralError,
roman3.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with too many repeated numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 122, in
testTooManyRepeatedNumerals
    self.assertRaises(roman3.InvalidRomanNumeralError,
roman3.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
```

```
======================================================================
FAIL: fromRoman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 99, in
testFromRomanKnownValues
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
======================================================================
FAIL: fromRoman(toRoman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 141, in
testSanity
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
----------------------------------------------------------------------
Ran 12 tests in 0.401s

FAILED (failures=6) ❶
```

❶ Ya nos quedan 6 fallos, y todos implican a `fromRoman`: los valores conocidos, las tres pruebas de entradas incorrectas, la prueba de mayúsculas y la de cordura. Esto significa que `toRoman` ha pasado todas las pruebas que podía pasar por sí sola (está incluida en la prueba de cordura, pero esto también necesita que esté escrita `fromRoman` y aún no lo está). Lo que significa que debemos dejar de escribir en `toRoman` ahora. Nada de ajustarla, ni modificarla, ni pruebas adicionales "por si acaso". Pare. Ahora. Retírese del teclado.

La cosa más importante que le puede decir una prueba unitaria exhaustiva es cuándo debe dejar de programar. Cuando una función pase todas sus pruebas unitarias, deje de programarla. Cuando un módulo pase todas sus pruebas unitarias, deje de programar en él.

14.4. `roman.py`, fase 4

Ahora que está hecha `toRoman` es hora de empezar a programar `fromRoman`. Gracias a la rica estructura de datos que relaciona cada número romano a un valor entero, no es más difícil que la función `toRoman`.

Ejemplo 14.9. `roman4.py`

Este fichero está disponible en `py/roman/stage4/` dentro del directorio de ejemplos.

Si aún no lo ha hecho, puede descargar éste ejemplo y otros usados en este libro.

```
"""Convert to and from Roman numerals"""

#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Define digit mapping
romanNumeralMap = (('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))

# toRoman function omitted for clarity (it hasn't changed)

def fromRoman(s):
```

```
    """convert Roman numeral to integer"""
    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral: ❶
            result += integer
            index += len(numeral)
    return result
```

❶ Aquí el patrón es el mismo que en `toRoman`. Iteramos sobre la estructura de números romanos (una tupla de tuplas), y en lugar de buscar el número entero más alto tan frecuentemente como sea posible, buscamos la cadena del número romano "más alto" lo más frecuentemente posible.

Ejemplo 14.10. Cómo funciona `fromRoman`

Si no tiene claro cómo funciona `fromRoman`, añada una sentencia `print` al final del bucle `while`:

```
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
            print 'encontré', numeral, 'de longitud', len(numeral), ',
sumando', integer
>>> import roman4
>>> roman4.fromRoman('MCMLXXII')
encontré M , de longitud 1, sumando 1000
encontré CM , de longitud 2, sumando 900
encontré L , de longitud 1, sumando 50
encontré X , de longitud 1, sumando 10
encontré X , de longitud 1, sumando 10
encontré I , de longitud 1, sumando 1
encontré I , de longitud 1, sumando 1
1972
```

Ejemplo 14.11. Salida de `romantest4.py` frente a `roman4.py`

```
fromRoman should only accept uppercase input ... FAIL
toRoman should always return uppercase ... ok
fromRoman should fail with malformed antecedents ... FAIL
fromRoman should fail with repeated pairs of numerals ... FAIL
```

```
fromRoman should fail with too many repeated numerals ... FAIL
fromRoman should give known result with known input ... ok ❶
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok                  ❷
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
```

❶ Dos noticias excitantes aquí. La primera es que `fromRoman` funciona con entradas correctas, al menos con todos los valores conocidos que probamos.

❷ La segunda es que también pasa la prueba de cordura. Combinándola con las pruebas de valores conocidos podemos estar razonablemente seguros de que tanto `toRoman` como `fromRoman` funcionan adecuadamente con todos los valores buenos posibles. (No está garantizado; es teóricamente posible que `toRoman` tenga un fallo que produzca el número romano incorrecto para un conjunto particular de entradas, *y* que `fromRoman` tenga un fallo recíproco que produzca el los mismos valores enteros errónes para exactamente el mismo conjunto de números romanos que `toRoman` genera incorrectamente. Dependiendo de su aplicación y sus requisitos, esto probablemente le moleste; si lo hace, escriba casos de prueba más exhaustivos hasta que deje de molestarle).

```
======================================================================
FAIL: fromRoman should only accept uppercase input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage4\romantest4.py", line 156, in
testFromRomanCase
    roman4.fromRoman, numeral.lower())
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with malformed antecedents
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage4\romantest4.py", line 133, in
testMalformedAntecedent
```

```
    self.assertRaises(roman4.InvalidRomanNumeralError,
roman4.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with repeated pairs of numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage4\romantest4.py", line 127, in
testRepeatedPairs
    self.assertRaises(roman4.InvalidRomanNumeralError,
roman4.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with too many repeated numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage4\romantest4.py", line 122, in
testTooManyRepeatedNumerals
    self.assertRaises(roman4.InvalidRomanNumeralError,
roman4.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
----------------------------------------------------------------------
Ran 12 tests in 1.222s

FAILED (failures=4)
```

14.5. `roman.py`, fase 5

Ahora que `fromRoman` funciona adecuadamente con entradas correctas es el momento de encajar la última pieza del puzzle: hacer que funcione adecuadamente con entradas incorrectas. Esto implica buscar una manera de mirar una cadena y determinar si es un número romano válido. Esto es inherentemente más difícil que validar entrada numérica en `toRoman`, pero tenemos una herramienta potente a nuestra disposición: expresiones regulares.

Si no le son familiares las expresiones regulares y no ha leído el capítulo Capítulo 7, *Expresiones regulares*, ahora puede ser un buen momento.

Como vio en Sección 7.3, "Caso de estudio: números romanos", hay varias reglas simples para construir un número romano usando las letras `M`, `D`, `C`, `L`, `X`, `V`, e `I`. Revisemos las reglas:

1. Los caracteres son aditivos. `I` es `1`, `II` es `2`, y `III` es `3`. `VI` es `6` (literalmente, "`5` y `1`"), `VII` es `7`, y `VIII` es `8`.
2. Los caracteres de unos (`I`, `X`, `C`, y `M`) se pueden repetir hasta tres veces. Para `4`, necesitamos restar del siguiente carácter de cinco más cercano. No podemos representar `4` como `IIII`; en su lugar se representa como `IV` ("`1` menos que `5`"). `40` se escribe como `XL` ("`10` menos que `50`"), `41` como `XLI`, `42` como `XLII`, `43` como `XLIII`, y luego `44` como `XLIV` ("`10` menos que `50` y `1` menos que `5`").
3. De forma similar, para `9` necesitamos restar del siguiente carácter de uno más grande: `8` es `VIII`, pero `9` es `IX` ("`1` menos que `10`"), no `VIIII` (ya que el carácter `I` no se puede repetir cuatro veces). `90` es `XC`, `900` es `CM`.
4. Los caracteres de cinco no se pueden repetir. `10`siempre se representa como `X`, nuncac como `VV`. `100` es siempre `C`, nunca `LL`.
5. Los números romanos siempre se escriben de mayor a menor, y se leen de izquierda a derecha, así que el orden de los caracteres importa mucho. `DC` es `600`; `CD` es un número completamente diferente (`400`, "`100` menos que `500`"). `CI` es `101`; `IC` ni siquiera es un número romano válido (porque no se puede restar `1` directamente de `100`; necesitamos escribirlo como `XCIX`, "`10` menos que `100` y `1` menos que `10`").

Ejemplo 14.12. `roman5.py`

Este fichero está disponible en `py/roman/stage5/` dentro del directorio de ejemplos.

Si aún no lo ha hecho, puede descargar éste ejemplo y otros usados en este libro.

```
"""Convert to and from Roman numerals"""
import re

#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Define digit mapping
romanNumeralMap = (('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))

def toRoman(n):
    """convert integer to Roman numeral"""
    if not (0 < n < 4000):
        raise OutOfRangeError, "number out of range (must be 1..3999)"
    if int(n) <> n:
        raise NotIntegerError, "non-integers can not be converted"

    result = ""
    for numeral, integer in romanNumeralMap:
        while n >= integer:
            result += numeral
            n -= integer
    return result

#Define pattern to detect valid Roman numerals
romanNumeralPattern =
'^M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$' ❶
```

```
def fromRoman(s):
    """convert Roman numeral to integer"""
    if not re.search(romanNumeralPattern, s):
❷
        raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' %
s

    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

❶ Esto es sólo una continuación del patrón que comentamos en Sección 7.3, "Caso de estudio: números romanos". Los lugares de las decenas son `XC` (`90`), `XL` (`40`), o una `L` opcional seguida de 0 a 3 caracteres `X` opcionales. El lugar de las unidades es `IX` (`9`), `IV` (`4`), o una `V` opcional seguida de 0 a 3 caracteres `I` opcionales.

❷ Habiendo codificado todas esta lógica en una expresión regular, el código para comprobar un número romano inválido se vuelve trivial. Si `re.search` devuelve un objeto entoncesl a expresión regular ha coincidido y la entrada es válida; si no, la entrada es inválida.

Llegados aquí, se le permite ser excéptico al pensar que esa expresión regular grande y fea pueda capturar posiblemente todos los tipos de números romanos no válidos. Pero no se limite a aceptar mi palabra, mire los resultados:

Ejemplo 14.13. Salida de `romantest5.py` frente a `roman5.py`

```
fromRoman should only accept uppercase input ... ok          ❶
toRoman should always return uppercase ... ok
fromRoman should fail with malformed antecedents ... ok      ❷
fromRoman should fail with repeated pairs of numerals ... ok ❸
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
```

```
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 12 tests in 2.864s

OK                                                                    ❹
```

❶ Una cosa que no mencioné sobre las expresiones regulares es que, por omisión, diferencian las mayúsculas de las minúsculas. Como la expresión regular `romanNumeralPattern` se expresó en letras mayúsculas, la comprobación `re.search` rechazará cualquier entrada que no esté completamente en mayúsculas. Así que pasa el test de entrada en mayúsculas.

❷ Más importante, también pasa la prueba de entrada incorrecta. Por ejemplo, la prueba de antecedentes mal formados comprueba casos como `MCMC`. Como ya ha visto esto no coincide con la expresión regular, así que `fromRoman` lanza una excepción `InvalidRomanNumeralError` que es lo que el caso de prueba de antecedentes mal formados estaba esperando, así que la prueba pasa.

❸ De hecho, pasan todas las pruebas de entrada incorrecta. Esta expresión regular captura todo lo que podríamos pensar cuando hicimos nuestros casos de pruebas.

❹ Y el premio anticlimático del año se lo lleva la palabra "`OK`" que imprime el módulo `unittest` cuando pasan todas las pruebas.

Cuando hayan pasado todas sus pruebas, deje de programar.

Capítulo 15. Refactorización

15.1. Gestión de fallos

A pesar de nuestros mejores esfuerzos para escribir pruebas unitarias exhaustivas, los fallos aparecen. ¿A qué me refiero con "un fallo"? Un fallo es un caso de prueba que aún no hemos escrito.

Ejemplo 15.1. El fallo

```
>>> import roman5
>>> roman5.fromRoman("") ❶
0
```

❶ ¿Recuerda cuando en la sección anterior vimos que una cadena vacía coincidía con la expresión regular que estábamos usando para los números romanos válidos? Bien, resulta que esto sigue siendo cierto para la versión final de la expresión regular. Y esto es un fallo; queremos que una cadena vacía lance una excepción `InvalidRomanNumeralError` igual que cualquier otra secuencia de caracteres que no represente un número romano válido.

Tras reproducir el fallo, y antes de arreglarlo, debería escribir un caso de prueba que falle que ilustre el fallo.

Ejemplo 15.2. Prueba del fallo (`romantest61.py`)

```
class FromRomanBadInput(unittest.TestCase):

    # se omiten los casos de prueba anteriores por claridad (no han
```

```
    # cambiado)

    def testBlank(self):
        """fromRoman should fail with blank string"""
        self.assertRaises(roman.InvalidRomanNumeralError,
roman.fromRoman, "") ❶
```

❶ Cosas muy simples aquí. Invocamos `fromRoman` con una cadena vacía y nos aseguramos de que lanza una excepción `InvalidRomanNumeralError`. La parte dura fue encontrar el fallo; ahora que lo sabemos, probarlo es la parte sencilla.

Como el código tiene un fallo, y ahora tenemos un caso de prueba que busca ese fallo, la prueba fallará:

Ejemplo 15.3. Salida de `romantest61.py` frente a `roman61.py`

```
fromRoman should only accept uppercase input ... ok
toRoman should always return uppercase ... ok
fromRoman should fail with blank string ... FAIL
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok

======================================================================
FAIL: fromRoman should fail with blank string
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage6\romantest61.py", line 137, in
testBlank
    self.assertRaises(roman61.InvalidRomanNumeralError,
roman61.fromRoman, "")
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
```

```
AssertionError: InvalidRomanNumeralError
----------------------------------------------------------------------
Ran 13 tests in 2.864s

FAILED (failures=1)
```

Ahora podemos arreglar el fallo.

Ejemplo 15.4. Arreglo del fallo (`roman62.py`)

Este fichero está disponible en `py/roman/stage6/` dentro del directorio de ejemplos.

```
def fromRoman(s):
    """convert Roman numeral to integer"""
    if not s: ❶
        raise InvalidRomanNumeralError, 'Input can not be blank'
    if not re.search(romanNumeralPattern, s):
        raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' %
s

    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

❶ Sólo se necesitan dos líneas de código: una comprobación explícita en busca de una cadena vacía, y una sentencia `raise`.

Ejemplo 15.5. Salida de `romantest62.py` frente a `roman62.py`

```
fromRoman should only accept uppercase input ... ok
toRoman should always return uppercase ... ok
fromRoman should fail with blank string ... ok ❶
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
```

```
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 13 tests in 2.834s

OK ❷
```

❶ El caso de prueba de la cadena vacía pasa ahora, así que se ha corregido el fallo.

❷ Todos los demás casos de prueba siguen pasando, lo que significa que este arreglo no ha roto nada más. Deje de programar.

Programar de esta manera no hace más fácil arreglar fallos. Los fallos simples (como éste) precisan casos de prueba sencillos; los fallos complejos precisan casos de prueba complejos. En un entorno centrado en las pruebas puede *parecer* que se tarda más en corregir un fallo, ya que se necesita articular en el código exactamente cual es el fallo (escribir el caso de prueba) y luego corregir el fallo en sí. Si el caso de prueba no tiene éxito ahora, tendremos que averiguar si el arreglo fue incorrecto o si el propio caso de prueba tiene un fallo. Sin embargo, a la larga, este tira y afloja entre código de prueba y código probado se amortiza por sí solo. Además, ya que podemos ejecutar de nuevo de forma sencilla *todos* los casos de prueba junto con el código nuevo, es mucho menos probable que rompamos código viejo al arreglar el nuevo. La prueba unitaria de hoy es la prueba de regresión de mañana.

15.2. Tratamiento del cambio de requisitos

A pesar de nuestros mejores esfuerzos para agarrar a nuestros clientes y extraerles requisitos exactos usando el dolor de cosas horribles que impliquen tijeras y cera caliente, los requisitos cambiarán. La mayoría de los clientes no

sabe lo que quiere hasta que lo ve, e incluso si lo saben, no son buenos explicando qué quieren de forma lo suficientemente precisa como para que sea útil. Así que prepárese para actualizar loscasos de prueba según cambien los requisitos.

Suponga, por ejemplo, que quería expandir el rango de las funciones de conversión de números romanos. ¿Recuerda la regla que decía que ningún carácter podía repetirse más de tres veces? Bien, los romanos quisieron hacer una excepción a esta regla teniendo 4 caracteres `M` seguidos para representar `4000`. Si hace este cambio, será capaz de expandir el rango de los números convertibles de `1..3999` a `1..4999`. Pero primero necesitará hacer algunos cambios en los casos de prueba.

Ejemplo 15.6. Modificación de los casos de prueba por nuevos requisitos (`romantest71.py`)

Este fichero está disponible en `py/roman/stage7/` dentro del directorio de ejemplos.

Si aún no lo ha hecho, puede descargar éste ejemplo y otros usados en este libro.

```
import roman71
import unittest

class KnownValues(unittest.TestCase):
    knownValues = ( (1, 'I'),
                    (2, 'II'),
                    (3, 'III'),
                    (4, 'IV'),
                    (5, 'V'),
                    (6, 'VI'),
                    (7, 'VII'),
                    (8, 'VIII'),
                    (9, 'IX'),
                    (10, 'X'),
                    (50, 'L'),
```

```
(100, 'C'),
(500, 'D'),
(1000, 'M'),
(31, 'XXXI'),
(148, 'CXLVIII'),
(294, 'CCXCIV'),
(312, 'CCCXII'),
(421, 'CDXXI'),
(528, 'DXXVIII'),
(621, 'DCXXI'),
(782, 'DCCLXXXII'),
(870, 'DCCCLXX'),
(941, 'CMXLI'),
(1043, 'MXLIII'),
(1110, 'MCX'),
(1226, 'MCCXXVI'),
(1301, 'MCCCI'),
(1485, 'MCDLXXXV'),
(1509, 'MDIX'),
(1607, 'MDCVII'),
(1754, 'MDCCLIV'),
(1832, 'MDCCCXXXII'),
(1993, 'MCMXCIII'),
(2074, 'MMLXXIV'),
(2152, 'MMCLII'),
(2212, 'MMCCXII'),
(2343, 'MMCCCXLIII'),
(2499, 'MMCDXCIX'),
(2574, 'MMDLXXIV'),
(2646, 'MMDCXLVI'),
(2723, 'MMDCCXXIII'),
(2892, 'MMDCCCXCII'),
(2975, 'MMCMLXXV'),
(3051, 'MMMLI'),
(3185, 'MMMCLXXXV'),
(3250, 'MMMCCL'),
(3313, 'MMMCCCXIII'),
(3408, 'MMMCDVIII'),
(3501, 'MMMDI'),
(3610, 'MMMDCX'),
(3743, 'MMMDCCXLIII'),
(3844, 'MMMDCCCXLIV'),
```

```
                    (3888, 'MMMDCCCLXXXVIII'),
                    (3940, 'MMMCMXL'),
                    (3999, 'MMMCMXCIX'),
                    (4000, 'MMMM'),
❶
                    (4500, 'MMMMD'),
                    (4888, 'MMMMDCCCLXXXVIII'),
                    (4999, 'MMMMCMXCIX'))

    def testToRomanKnownValues(self):
        """toRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman71.toRoman(integer)
            self.assertEqual(numeral, result)

    def testFromRomanKnownValues(self):
        """fromRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman71.fromRoman(numeral)
            self.assertEqual(integer, result)

class ToRomanBadInput(unittest.TestCase):
    def testTooLarge(self):
        """toRoman should fail with large input"""
        self.assertRaises(roman71.OutOfRangeError, roman71.toRoman,
5000) ❷

    def testZero(self):
        """toRoman should fail with 0 input"""
        self.assertRaises(roman71.OutOfRangeError, roman71.toRoman, 0)

    def testNegative(self):
        """toRoman should fail with negative input"""
        self.assertRaises(roman71.OutOfRangeError, roman71.toRoman, -
1)

    def testNonInteger(self):
        """toRoman should fail with non-integer input"""
        self.assertRaises(roman71.NotIntegerError, roman71.toRoman,
0.5)

class FromRomanBadInput(unittest.TestCase):
```

```
    def testTooManyRepeatedNumerals(self):
        """fromRoman should fail with too many repeated numerals"""
        for s in ('MMMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):
❸
            self.assertRaises(roman71.InvalidRomanNumeralError,
roman71.fromRoman, s)

    def testRepeatedPairs(self):
        """fromRoman should fail with repeated pairs of numerals"""
        for s in ('CMCM', 'CDCD', 'XCXC', 'XLXL', 'IXIX', 'IVIV'):
            self.assertRaises(roman71.InvalidRomanNumeralError,
roman71.fromRoman, s)

    def testMalformedAntecedent(self):
        """fromRoman should fail with malformed antecedents"""
        for s in ('IIMXCC', 'VX', 'DCM', 'CMM', 'IXIV',
                  'MCMC', 'XCX', 'IVI', 'LM', 'LD', 'LC'):
            self.assertRaises(roman71.InvalidRomanNumeralError,
roman71.fromRoman, s)

    def testBlank(self):
        """fromRoman should fail with blank string"""
        self.assertRaises(roman71.InvalidRomanNumeralError,
roman71.fromRoman, "")

class SanityCheck(unittest.TestCase):
    def testSanity(self):
        """fromRoman(toRoman(n))==n for all n"""
        for integer in range(1, 5000):
❹
            numeral = roman71.toRoman(integer)
            result = roman71.fromRoman(numeral)
            self.assertEqual(integer, result)

class CaseCheck(unittest.TestCase):
    def testToRomanCase(self):
        """toRoman should always return uppercase"""
        for integer in range(1, 5000):
            numeral = roman71.toRoman(integer)
            self.assertEqual(numeral, numeral.upper())

    def testFromRomanCase(self):
```

```
        """fromRoman should only accept uppercase input"""
        for integer in range(1, 5000):
            numeral = roman71.toRoman(integer)
            roman71.fromRoman(numeral.upper())
            self.assertRaises(roman71.InvalidRomanNumeralError,
                              roman71.fromRoman, numeral.lower())

if __name__ == "__main__":
    unittest.main()
```

❶ Los valores conocidos existentes no cambian (siguen siendo valores razonables que probar), pero necesitamos añadir unos cuántos más en el rango de `4000`. He incluído `4000` (el más corto), `4500` (el segundo más corto), `4888` (el más largo) y `4999` (el más grande).

❷ La definición de "entrada grande" ha cambiado. Esta prueba llamaba a `toRoman` con `4000` y esperaba un error; ahora que `4000-4999` son valores buenos debemos subir esto a `5000`.

❸ La definición de "demasiados números repetidos" también ha cambiado. Esta prueba llamaba a `fromRoman` con `'MMMM'` y esperaba un error; ahora que `MMMM` se considera un número romano válido, necesitamos subir esto a `'MMMMM'`.

❹ La prueba de cordura y la de mayúsculas iteran sobre cada número del rango, desde `1` hasta `3999`. Como el rango se ha expandido estos bucles `for` necesitan actualizarse igualmente para llegar hasta `4999`.

Ahora nuestros casos de prueba están actualizados a los nuevos requisitos, pero nuestro código no, así que espere que varios de ellos fallen.

Ejemplo 15.7. Salida de `romantest71.py` frente a `roman71.py`

```
fromRoman should only accept uppercase input ... ERROR        ❶
toRoman should always return uppercase ... ERROR
fromRoman should fail with blank string ... ok
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
```

```
fromRoman should give known result with known input ... ERROR ❷
toRoman should give known result with known input ... ERROR   ❸
fromRoman(toRoman(n))==n for all n ... ERROR                  ❹
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
```

❶ Nuestra comprobación de mayúsculas falla ahora porque itera de `1` a `4999`, pero `toRoman` sólo acepta números de `1` a `3999`, así que fallará encuanto el caso de prueba llegue a `4000`.

❷ La prueba de valores conocidos de `fromRoman` fallará en cuanto llegue a `'MMMM'`, porque `fromRoman` sigue pensando que esto no es un número romano válido.

❸ La prueba de valores conocidos de `toRoman` fallará en cuanto se tope con `4000`, porque `toRoman` sigue pensando que está fuera de rango.

❹ La prueba de cordura también fallará en cuanto llegue a `4000`, porque `toRoman` piensa que está fuera de rango.

```
======================================================================
ERROR: fromRoman should only accept uppercase input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 161, in
testFromRomanCase
    numeral = roman71.toRoman(integer)
  File "roman71.py", line 28, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999)
======================================================================
ERROR: toRoman should always return uppercase
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 155, in
testToRomanCase
    numeral = roman71.toRoman(integer)
  File "roman71.py", line 28, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999)
```

```
======================================================================
ERROR: fromRoman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 102, in
testFromRomanKnownValues
    result = roman71.fromRoman(numeral)
  File "roman71.py", line 47, in fromRoman
    raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' % s
InvalidRomanNumeralError: Invalid Roman numeral: MMMM
======================================================================
ERROR: toRoman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 96, in
testToRomanKnownValues
    result = roman71.toRoman(integer)
  File "roman71.py", line 28, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999)
======================================================================
ERROR: fromRoman(toRoman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 147, in
testSanity
    numeral = roman71.toRoman(integer)
  File "roman71.py", line 28, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999)
----------------------------------------------------------------------
Ran 13 tests in 2.213s

FAILED (errors=5)
```

Ahora que tenemos casos de prueba que fallan debido a los nuevos requisitos, debemos pensar en arreglar el código para que quede de acuerdo a estos casos de prueba. (Una cosa a la que lleva tiempo acostumbrase cuando se empieza a trabajar usando pruebas unitarias es que el código probado nunca está "por delante" de los casos de prueba. Mientras esté por detrás sigue habiendo trabajo que hacer, y en cuanto alcanzan a los casos de uso, se deja de programar).

Ejemplo 15.8. Programación de los nuevos requisitos (`roman72.py`)

Este fichero está disponible en `py/roman/stage7/` dentro del directorio de ejemplos.

```
"""Convert to and from Roman numerals"""
import re

#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Define digit mapping
romanNumeralMap = (('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))

def toRoman(n):
    """convert integer to Roman numeral"""
    if not (0 < n < 5000):
❶
        raise OutOfRangeError, "number out of range (must be 1..4999)"
    if int(n) <> n:
        raise NotIntegerError, "non-integers can not be converted"

    result = ""
    for numeral, integer in romanNumeralMap:
        while n >= integer:
```

```
            result += numeral
            n -= integer
    return result

#Define pattern to detect valid Roman numerals
romanNumeralPattern =
'^M?M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$' ❷

def fromRoman(s):
    """convert Roman numeral to integer"""
    if not s:
        raise InvalidRomanNumeralError, 'Input can not be blank'
    if not re.search(romanNumeralPattern, s):
        raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' %
s

    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

❶ `toRoman` sólo necesita un pequeño cambio en la comprobación de rango. Donde solíamos comprobar `0 < n < 4000` ahora comprobamos `0 < n < 5000`. Y cambiamos el mensaje de error que lanzamos con `raise` para reflejar el nuevo rango aceptable (`1..4999` en lugar de `1..3999`). No necesita hacer ningún cambio al resto de la función; ya considera los nuevos casos. (Añade felizmente `'M'` por cada millar que encuentra; dado `4000`, devolverá `'MMMM'`. La única razón por la que no lo hacía antes es porque le detuvimos explícitamente con la comprobación de rango).

❷ En realidad no necesita hacer cambios a `fromRoman`. El único cambio es para `romanNumeralPattern`; si observa atentamente advertirá que hemos añadido otra `M` opcional en la primera sección de la expresión regular. Esto permitirá 4 caracteres `M` opcionales en lugar de 3, lo que significa que permitiremos los equivalentes en romanos de `4999` en lugar de `3999`. La función `fromRoman` en sí es completamente general; simplemente busca números romanos repetidos

y los suma, sin importar cuántas veces aparecen. La única razón de que no tratara `'MMMM'` antes es que se lo prohibimos explícitamente con el patrón de la expresión regular.

Puede que no se crea que estos dos pequeños cambios sean todo lo que necesitamos. ¡Hey!, no tiene por qué creer en mi palabra; véalo usted mismo:

Ejemplo 15.9. Salida de `romantest72.py` frente a `roman72.py`

```
fromRoman should only accept uppercase input ... ok
toRoman should always return uppercase ... ok
fromRoman should fail with blank string ... ok
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 13 tests in 3.685s

OK ❶
```

❶ Todos los casos de prueba pasan. Deje de programar.

Tener pruebas unitarias exhaustivas significa no tener nunca que depender de un programador que diga “Confíe en mí.”

15.3. Refactorización

Lo mejor de las pruebas unitarias exhaustivas no es la sensación que le queda cuando todos los casos de prueba terminan por pasar, o incluso la que le llega cuando alguien le acusa de romper su código y usted puede *probar* realmente

que no lo hizo. Lo mejor de las pruebas unitarias es que le da la libertad de refactorizar sin piedad.

La refactorización es el proceso de tomar código que funciona y hacer que funcione mejor. Normalmente "mejor" significa "más rápido", aunque puede significar también que "usa menos memoria" o "usa menos espacio en disco" o simplemente "es más elegante". Independientemente de lo queu signifique para usted, su proyecto, en su entorno, la refactorización es importante para la salud a largo plazo de cualquier programa.

Aquí, "mejor" significa "más rápido". Específicamente, la función `fromRoman` es más lenta de lo que podría ser debido a esa expresión regular tan grande y fea que usamos para validar los números romanos. Probablemente no merece la pena eliminar completametne las expresiones regulares (sería difícil, y podría acabar por no ser más rápido), pero podemos acelerar la función compilando la expresión regular previamente.

Ejemplo 15.10. Compilación de expresiones regulares

```
>>> import re
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'M')                ❶
<SRE_Match object at 01090490>
>>> compiledPattern = re.compile(pattern) ❷
>>> compiledPattern
<SRE_Pattern object at 00F06E28>
>>> dir(compiledPattern)                  ❸
['findall', 'match', 'scanner', 'search', 'split', 'sub', 'subn']
>>> compiledPattern.search('M')           ❹
<SRE_Match object at 01104928>
```

❶ Ésta es la sintaxis que ha visto anteriormente: `re.search` toma una expresión regular como una cadena (`pattern`) y una cadena con la que compararla (`'M'`). Si el patrón coincide, la función devuelve un objeto al que se le puede preguntar para saber exactamente qué coincidió y cómo.

❷ Ésta es la nueva sintaxis: `re.compile` toma una expresión regular como

cadena y devuelve un objeto de patrón. Observe que aquí no hay cadena con la que comparar. Compilar la expresión regular no tiene nada que ver con compararla con ninguna cadena específica (como `'M'`); sólo implica a la propia expresión regular.

❸ El objeto de patrón compilado que devuelve `re.compile` tiene varias funciones que parecen útiles, incluyendo varias que están disponibles directamente en el módulo `re` (como `search` y `sub`).

❹ Llamar a la función `search` del objeto patrón compilado con la cadena `'M'` cumple la misma función que llamar a `re.search` con la expresión regular y la cadena `'M'`. Sólo que mucho, mucho más rápido. (En realidad, la función `re.search` simplemente compila la expresión regular y llama al método `search` del objeto patrón resultante por usted).

Siempre que vaya a usar una expresión regular más de una vez, debería compilarla para obtener un objeto patrón y luego llamar directamente a los métodos del patrón.

Ejemplo 15.11. Expresiones regulares compiladas en `roman81.py`

Este fichero está disponible en `py/roman/stage8/` dentro del directorio de ejemplos.

Si aún no lo ha hecho, puede descargar éste ejemplo y otros usados en este libro.

```
# toRoman and rest of module omitted for clarity

romanNumeralPattern = \

re.compile('^M?M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$
') ❶

def fromRoman(s):
    """convert Roman numeral to integer"""
    if not s:
```

```
        raise InvalidRomanNumeralError, 'Input can not be blank'
    if not romanNumeralPattern.search(s):
❷
        raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' %
s

    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

❶ Esto parece muy similar, pero en realidad ha cambiado mucho. `romanNumeralPattern` ya no es una cadena; es un objeto de patrón que devolvió `re.compile`.

❷ Eso significa que podemos invocar métodos directamente sobre `romanNumeralPattern`. Esto será mucho, mucho más rápido que invocar cada vez a `re.search`. La expresión regular se compila una vez y se almacena en `romanNumeralPattern` cuando se importa el módulo por primera vez; luego, cada vez que llamamos a `fromRoman` comparamos inmediatamente la cadena de entrada con la expresión regular, sin ningún que ocurra paso intermedio.

¿Cuánto más rápido es compilar las expresiones regulares? Véalo por sí mismo:

Ejemplo 15.12. Salida de `romantest81.py` frente a `roman81.py`

```
.............          ❶
----------------------------------------------------------------------
Ran 13 tests in 3.385s ❷

OK                     ❸
```

❶ Sólo una nota al respecto: esta vez he ejecutado la prueba unitaria *sin* la opción `-v`, así que en lugar de la `cadena de documentación` completa por cada prueba lo que obtenemos es un punto cada vez que pasa una. (Si una prueba falla obtendremos una `F` y si tiene un error veremos una `E`. Seguimos

obteniendo volcados de pila completos por cada fallo y error, para poder encontrar cualquier problema).

❷ Ejecutamos 13 pruebas en 3.385 segundos, comparado con los 3.685 segundos sin precompilar las expresiones regulares. Eso es una mejora global del 8%, y recuerde que la mayoría del tiempo de la prueba se pasa haciendo otras cosa. (He probado separadamente las expresiones regulares por sí mismas, aparte del resto de pruebas unitarias, y encontré que compilar esta expresión regular acelera la búsqueda -search- una media del 54%). No está mal para una simple corrección.

❸ Oh, y en caso de que se lo esté preguntando, precompilar la expresión regular no rompió nada como acabo de probar.

Hay otra optimización de rendimiento que me gustaría probar. Dada la complejidad de la sintaxis de las expresiones regulares, no debería sorprenderle que con frecuencia haya más de una manera de escribir la misma expresión. Tras discutir un poco este módulo en comp.lang.python alguien me sugirió que probase la sintaxis {*m*,*n*} con los caracteres repetidos opcionales.

Ejemplo 15.13. roman82.py

Este fichero está disponible en py/roman/stage8/ dentro del directorio de ejemplos.

Si aún no lo ha hecho, puede descargar éste ejemplo y otros usados en este libro.

```
# se omite el resto del programa por claridad

#versión antigua
#romanNumeralPattern = \
#
re.compile('^M?M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$
')

#versión nueva
```

```
romanNumeralPattern = \

re.compile('^M{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$')
❶
```

❶ Ahora hemos sustituído `M?M?M?M?` por `M{0,4}`. Ambas significan lo mismo: "coincidencia de 0 a 4 caracteres `M`". De forma similar, `C?C?C?` se vuelve `C{0,3}` (" coincidencia de 0 a 3 caracteres `C`") y lo mismo con `X` e `I`.

Esta forma de expresión regular es un poco más corta (aunque no más legible). La gran pregunta es, ¿es más rápida?

Ejemplo 15.14. Salida de `romantest82.py` frente a `roman82.py`

```
.............
----------------------------------------------------------------------
Ran 13 tests in 3.315s ❶

OK                     ❷
```

❶ Globalemente, esta prueba unitaria funciona un 2% más rápida con esta forma de expresión regular. No suena muy excitante, pero recuerde que la función `search` es una parte pequeña del total de la prueba unitaria; la mayoría del tiempo se lo pasa haciendo otras cosas. (Por separado, he comprobado sólo las expresiones regulares y encontré que la función `search` es un `11%` más rápida con esta sintaxis). Precompilando la expresión regular y reescribiendo parte de ella para que use esta nuevas sintaxis hemos mejorado el rendimiento de la expresión regular más de un `60%`, y mejorado el rendimiento general de toda la prueba unitaria en más del `10%`.

❷ Más importante que cualquier mejora de rendimiento es el hecho de que el módulo sigue funcionando perfectamente. Ésta es la libertad de la que hablaba antes: libertad de ajustar, cambiar o reescribir cualquier parte del módulo y verificar que no hemos estropeado nada en el proceso. Esto no es da licencia para modificar sin fin el código sólo por el hecho de modificarlo; debe tener un objetivo muy específico ("hacer `fromRoman` más rápida"), y debe ser capaz de cumplir ese objetivo sin el menor atisbo de duda sobre

estar introduciendo nuevos fallos en el proceso.

Me gustaría hacer otra mejora más y entonces prometo que pararé de refactorizar y dejaré este módulo. Como ha visto repetidas veces, las expresiones regulares pueden volverse rápidamente bastante incomprensibles e ilegibles. No me gustaría volver sobre este módulo en seis meses y tener que mantenerlo. Por supuesto, los casos de prueba pasan todos así que sé que funciona, pero si no puedo imaginar *cómo* funciona va a ser difícil añadir nuevas características, arreglar nuevos fallos o simplemente mantenerlo. Como vio en Sección 7.5, "Expresiones regulares prolijas", Python proporciona una manera de documentar la lógica línea por línea.

Ejemplo 15.15. `roman83.py`

Este fichero está disponible en `py/roman/stage8/` dentro del directorio de ejemplos.

Si aún no lo ha hecho, puede descargar éste ejemplo y otros usados en este libro.

```
# se omite el resto del programa por claridad

#versión antigua
#romanNumeralPattern = \
#
re.compile('^M{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$')

#versión nueva
romanNumeralPattern = re.compile('''
    ^                   # beginning of string
    M{0,4}              # thousands - 0 to 4 M's
    (CM|CD|D?C{0,3})    # hundreds - 900 (CM), 400 (CD), 0-300 (0 to 3
C's),
                        #            or 500-800 (D, followed by 0 to 3
C's)
    (XC|XL|L?X{0,3})    # tens - 90 (XC), 40 (XL), 0-30 (0 to 3 X's),
                        #        or 50-80 (L, followed by 0 to 3 X's)
    (IX|IV|V?I{0,3})    # ones - 9 (IX), 4 (IV), 0-3 (0 to 3 I's),
```

```
                    #       or 5-8 (V, followed by 0 to 3 I's)
    $               # end of string
    ''', re.VERBOSE) ❶
```

❶ La función `re.compile` acepta un segundo argumento que es un conjunto de uno o más indicadores que controlan varias opciones sobre la expresión regular compilada. Aquí estamos especificando el indicador `re.VERBOSE`, que le dice a Python que hay comentarios en línea dentro de la propia expresión regular. *Ni* los comentarios *ni* los espacios en blanco alrededor de ellos se consideran parte de la expresión regular; la función `re.compile` los elimina cuando compila la expresión. Esta nueva versión "prolija" es idéntica a la antigua, pero infinitamente más legible.

Ejemplo 15.16. Salida de `romantest83.py` frente a `roman83.py`

```
.............
----------------------------------------------------------------------
Ran 13 tests in 3.315s ❶

OK                     ❷
```

❶ Esta versión nueva y "prolija" se ejecuta exactamente a la misma velocidad que la antigua. En realidad, los objetos de patrón compilado son los mismos, ya que la función `re.compile` elimina todo el material que hemos añadido.

❷ Esta versión nueva y "prolija" pasa todas las mismas pruebas que la antigua. No ha cambiado nad excepto que el programador que vuelva sobre este módulo dentro de seis meses tiene la oportunidad de comprender cómo trabaja la función.

15.4. Epílogo

El lector inteligente leería la sección anterior y la llevaría al siguiente nivel. El mayor dolor de cabeza (y lastre al rendimiento) en el programa tal como está escrito es la expresión regular, que precisamos porque no tenemos otra manera de hacer disección de un número romano. Pero sólo hay 5000; ¿por qué no podemos generar una tabla de búsqueda una vez y luego limitarnos a leerla? La

idea se vuelve aún mejor cuando nos damos cuenta de que no necesitamos usar expresiones regulares para nada. Mientras creamos la tabla para convertir enteros a números romanos podemos construir la tabla inversa que convierta romanos a enteros.

Y lo mejor de todo, ya tenemos un juego completo de pruebas unitarias. Ha cambiado la mitad del código del módulo, pero las pruebas unitarias siguen igual, así que podrá probar si su código funciona igual de bien que el original.

Ejemplo 15.17. `roman9.py`

Este fichero está disponible en `py/roman/stage9/` dentro del directorio de ejemplos.

Si aún no lo ha hecho, puede descargar éste ejemplo y otros usados en este libro.

```
#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Roman numerals must be less than 5000
MAX_ROMAN_NUMERAL = 4999

#Define digit mapping
romanNumeralMap = (('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))

#Create tables for fast conversion of roman numerals.
#See fillLookupTables() below.
toRomanTable = [ None ]  # Skip an index since Roman numerals have no
zero
fromRomanTable = {}

def toRoman(n):
    """convert integer to Roman numeral"""
    if not (0 < n <= MAX_ROMAN_NUMERAL):
        raise OutOfRangeError, "number out of range (must be 1..%s)" %
MAX_ROMAN_NUMERAL
    if int(n) <> n:
        raise NotIntegerError, "non-integers can not be converted"
    return toRomanTable[n]

def fromRoman(s):
    """convert Roman numeral to integer"""
    if not s:
        raise InvalidRomanNumeralError, "Input can not be blank"
    if not fromRomanTable.has_key(s):
        raise InvalidRomanNumeralError, "Invalid Roman numeral: %s" %
s
    return fromRomanTable[s]

def toRomanDynamic(n):
    """convert integer to Roman numeral using dynamic programming"""
    result = ""
    for numeral, integer in romanNumeralMap:
        if n >= integer:
            result = numeral
            n -= integer
            break
    if n > 0:
        result += toRomanTable[n]
    return result

def fillLookupTables():
    """compute all the possible roman numerals"""
    #Save the values in two global tables to convert to and from
integers.
```

```
    for integer in range(1, MAX_ROMAN_NUMERAL + 1):
        romanNumber = toRomanDynamic(integer)
        toRomanTable.append(romanNumber)
        fromRomanTable[romanNumber] = integer

fillLookupTables()
```

¿Cómo es de rápido?

Ejemplo 15.18. Salida de `romantest9.py` frente a `roman9.py`

```
.............
----------------------------------------------------------------------
Ran 13 tests in 0.791s

OK
```

Recuerde, el mejor rendimiento que llegó a conseguir con la versión original fue de 13 pruebas en 3.315 segundos. Por supuesto, no es una comparación completamente justa por que esta versión tardará más en importar (cuando llena las tablas de búsqueda). Pero como la importación se hace una sola vez, esto es obviable a la large.

¿La moral de la historia?

- La simplicidad es una virtud.
- Especialmente cuando hay expresiones regulares de por medio.
- Y las pruebas unitarias pueden darle la confianza de hacer refactorización a gran escala... incluso si no escribió usted el código original.

15.5. Resumen

Las pruebas unitarias son un concepto podente que, si se implementa adecuadamente, puede tanto reducir el coste de mantenimiento como incrementar la flexibilidad en cualquier proyecto a largo plazo. Es importante

también entender que las pruebas unitarias no son la panacea, un Resolutor Mágico de Problemas o una bala de plata. Escribir buenos casos de prueba es duro, y mantenerlos actualizados necesita disciplina (especialmente cuando los clientes están gritando que quieren arreglos para fallos críticos). Las pruebas unitarias no sustituyen otras formas de comprobación, incluyendo las pruebas funcionales, las de integración y las de aceptación del cliente. Pero son factibles y funcionan, y una vez que las ha visto trabajando uno se pregunta cómo ha podido pasar sin ellas hasta ahora.

Este capítulo ha cubeirto mucho terreno del que bastante ni siquiera era específico de Python. Hay infraestructura de pruebas unitarias para muchos lenguajes, y todos precisan que entienda los mismos conceptos básicos:

- Diseñe casos de prueba que sean específicos, automáticos e independientes
- Escriba los casos de prueba *antes* que el código van a probar
- Escriba pruebas que pasen entradas correctas y comprueben que los resultados son adecuados
- Escriba pruebas que pasen entradas incorrectas y comprueben que aparecen los fallos adecuados
- Escriba y actualice casos de prueba para ilustrar fallos o reflejar nuevos requisitos
- Refactorice sin piedad para mejorar el rendimiento, la escalabilidad, legibilidad, mantenibilidad o cualquier otra -idad que le falte

Además, debería sentirse cómodo haciendo las siguientes cosas específicas a Python:

- Derivar `unittest.TestCase` y escribir métodos para casos de prueba individuales
- Usar `assertEqual` para comprobar que una función devuelve un valor conocido
- Usar `assertRaises` para comprobar que una función lanza una excepción conocida

- Invocar a `unittest.main()` desde su cláusula `if __name__` para ejecutar todos los casos de prueba a la vez
- Ejecutar pruebas unitarias en modo prolijo o normal

Lecturas complementarias

- XProgramming.com tiene enlaces para descargar infraestructuras de prueba unitaria para muchos lenguajes distintos.

Capítulo 16. Programación Funcional

16.1. Inmersión

En Capítulo 13, *Pruebas unitarias (Unit Testing)* aprendimos la filosofía de las pruebas unitarias. En Capítulo 14, *Programación Test-First* implementamos paso a paso una prueba unitaria en Python. En Capítulo 15, *Refactorización* vimos cómo las pruebas unitarias hacen más sencilla la refactorización a gran escala. Este capítulo se apoyará en esos programas de ejemplo, pero aquí nos centraremos en técnicas más avanzadas específicas de Python, en lugar de en las pruebas unitarias en sí.

Lo que sigue es un programa de Python completo que funciona como infraestructura de pruebas de regresión rudimientaria. Toma las pruebas unitarias que hemos escrito para módulos individuales, los junta todos en una gran batería de pruebas y los ejecuta todos a la vez. Yo mismo uso este *script* como parte del proceso de compilación de este libro; tengo pruebas unitarias para muchos de los programas de ejemplo (no sólo el módulo `roman.py` que aparece en Capítulo 13, *Pruebas unitarias (Unit Testing)*), y la primera cosa que hace mi *script* de compilación automático es ejecutar este programa para asegurarse de que todos mis ejemplos siguen funcionando. Si esta prueba de regresión fallase, la compilación se detiene automáticamente. No quiero publicar ejemplos que no funcionen más de lo que usted querría bajarlos y

sentarse rascándose la cabeza y gritándole al monitor mientras se pregunta por qué no funcionan.

Ejemplo 16.1. `regression.py`

Si aún no lo ha hecho, puede descargar éste ejemplo y otros usados en este libro.

```
"""Regression testing framework

This module will search for scripts in the same directory named
XYZtest.py.  Each such script should be a test suite that tests a
module through PyUnit.  (As of Python 2.1, PyUnit is included in
the standard library as "unittest".)  This script will aggregate all
found test suites into one big test suite and run them all at once.
"""

import sys, os, re, unittest

def regressionTest():
    path = os.path.abspath(os.path.dirname(sys.argv[0]))
    files = os.listdir(path)
    test = re.compile("test\.py$", re.IGNORECASE)
    files = filter(test.search, files)
    filenameToModuleName = lambda f: os.path.splitext(f)[0]
    moduleNames = map(filenameToModuleName, files)
    modules = map(__import__, moduleNames)
    load = unittest.defaultTestLoader.loadTestsFromModule
    return unittest.TestSuite(map(load, modules))

if __name__ == "__main__":
    unittest.main(defaultTest="regressionTest")
```

Ejecutar este *script* en el mismo directorio del resto de los *script* de ejemplo del libro hará que encuentre todas las pruebas unitarias, llamadas `módulotest.py`, las ejecutará como una sola prueba y todas pasarán o fallarán a la vez.

Ejemplo 16.2. Salida de ejemplo de `regression.py`

```
[usted@localhost py]$ python regression.py -v
```

```
help should fail with no object ... ok                                  ❶
help should return known result for apihelper ... ok
help should honor collapse argument ... ok
help should honor spacing argument ... ok
buildConnectionString should fail with list input ... ok                ❷
buildConnectionString should fail with string input ... ok
buildConnectionString should fail with tuple input ... ok
buildConnectionString handles empty dictionary ... ok
buildConnectionString returns known result with known input ... ok
fromRoman should only accept uppercase input ... ok                     ❸
toRoman should always return uppercase ... ok
fromRoman should fail with blank string ... ok
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
kgp a ref test ... ok
kgp b ref test ... ok
kgp c ref test ... ok
kgp d ref test ... ok
kgp e ref test ... ok
kgp f ref test ... ok
kgp g ref test ... ok

----------------------------------------------------------------------
Ran 29 tests in 2.799s

OK
```

❶ The first 5 tests are from `apihelpertest.py`, which tests the example script from Capítulo 4, *El poder de la introspección*.

❷ Las 5 pruebas siguientes son de `odbchelpertest.py`, que comprueba el *script* de ejemplo de Capítulo 2, *Su primer programa en Python*.

❸ El resto son de `romantest.py`, que estudiamos en profundidad en

Capítulo 13, *Pruebas unitarias (Unit Testing)*.

16.2. Encontrar la ruta

A veces es útil, cuando ejecutamos *scripts* de Python desde la línea de órdenes, saber dónde está situado en el disco el *scripts* que está en marcha.

Éste es uno de esos pequeños trucos que es virtualmente imposible que averigüe por su cuenta, pero que es simple de recordar una vez lo ha visto. La clave es `sys.argv`. Como vio en Capítulo 9, *Procesamiento de XML*, almacena una lista de los argumentos de la línea de órdenes. Sin embargo, también contiene el nombre del *scripts* que está funcionando, exactamente igual que se le invocó en la línea de órdenes, y esto es información suficiente para determinar su situación.

Ejemplo 16.3. `fullpath.py`

Si aún no lo ha hecho, puede descargar éste ejemplo y otros usados en este libro.

```
import sys, os

print 'sys.argv[0] =', sys.argv[0]              ❶
pathname = os.path.dirname(sys.argv[0])        ❷
print 'path =', pathname
print 'full path =', os.path.abspath(pathname) ❸
```

❶ Independientemente de la manera en que ejecute un *scripts*, `sys.argv[0]` contendrá siempre el nombre del *scripts* exactamente tal como aparece en la línea de órdenes. Esto puede que incluya información de la ruta o no, como veremos en breve.

❷ `os.path.dirname` toma una cadena con un nombre de fichero y devuelve la porción correspondiente a la ruta. Si el nombre de fichero no incluye información de ruta, `os.path.dirname` devuelve una cadena vacía.

❸ Aquí lo importante es `os.path.abspath`. Toma el nombre de una ruta, que puede ser parcial o incluso estar vacía, y devuelve un nombre de ruta totalmente calificado.

`os.path.abspath` merece una mejor explicación. Es muy flexible; puede tomar cualquier tipo de ruta.

Ejemplo 16.4. Más explicaciones sobre `os.path.abspath`

```
>>> import os
>>> os.getcwd()                          ❶
/home/usted
>>> os.path.abspath('')                  ❷
/home/usted
>>> os.path.abspath('.ssh')              ❸
/home/usted/.ssh
>>> os.path.abspath('/home/usted/.ssh') ❹
/home/usted/.ssh
>>> os.path.abspath('.ssh/../foo/')     ❺
/home/usted/foo
```

❶ `os.getcwd()` devuelve el directorio de trabajo actual.

❷ Invocar a `os.path.abspath` con una cadena vacía devuelve el directorio actual de trabajo, igual que `os.getcwd()`.

❸ Una invocación a `os.path.abspath` pasando una ruta parcial construye un nombre de ruta completamente calificada, basándose en el directorio actual de trabajo.

❹ Llamar a `os.path.abspath` con una ruta completa la devuelve sin modificaciones.

❺ `os.path.abspath` también *normaliza* el nombre de ruta que devuelve. Observe que este ejemplo ha funcionado incluso aunque no tengo un directorio "foo". `os.path.abspath` nunca comprueba el disco realmente; es todo cuestión de manipulación de cadenas.

No hace falta que existan los nombres de rutas y ficheros que se le pasan a

`os.path.abspath`.

`os.path.abspath` no se limita a construir nombres completos de rutas, también los normaliza. Eso significa que si está en el directorio `/usr/`, `os.path.abspath('bin/../local/bin')` devolverá `/usr/local/bin`. Normaliza la ruta haciéndola lo más sencilla posible. Si sólo quiere normalizar una ruta así, sin convertirla en completa, use `os.path.normpath`.

Ejemplo 16.5. Salida de ejemplo de `fullpath.py`

```
[usted@localhost py]$ python
/home/usted/diveintopython/common/py/fullpath.py ❶
sys.argv[0] = /home/usted/diveintopython/common/py/fullpath.py
path = /home/usted/diveintopython/common/py
full path = /home/usted/diveintopython/common/py
[usted@localhost diveintopython]$ python common/py/fullpath.py
❷
sys.argv[0] = common/py/fullpath.py
path = common/py
full path = /home/usted/diveintopython/common/py
[usted@localhost diveintopython]$ cd common/py
[usted@localhost py]$ python fullpath.py
❸
sys.argv[0] = fullpath.py
path =
full path = /home/usted/diveintopython/common/py
```

❶ En el primer caso, `sys.argv[0]` incluye la ruta completa del *script*. Puede usar la función `os.path.dirname` para eliminar el nombre del *script* y devolver el nombre completo del directorio, y `os.path.abspath` nos devuelve simplemente lo mismo que le pasamos.

❷ Si se ejecuta el *script* usando una ruta parcial, `sys.argv[0]` seguirá conteniendo exactamente lo que aparece en la línea de órdenes. `os.path.dirname` nos devolverá una ruta parcial (relativa al directorio actual) y `os.path.abspath` construirá una ruta completa partiendo de la parcial.

❸ Si se ejecuta el *script* desde el directorio actual sin dar ninguna ruta,

`os.path.dirname` devolverá una cadena vacía. Dada una cadena vacía, `os.path.abspath` devuelve el directorio actual, que es lo que queríamos ya que el *script* se ejecutó desde el directorio actual.

`os.path.abspath` es multiplataforma igual que las otras funciones del os módulos `os` y `os.path`. Los resultados parecerán ligeramente diferentes que en mis ejemplos si está usted usando Windows (que usa barras invertidas como separador de ruta) o Mac OS (que usa dos puntos - :), pero seguirá funcionando. Ésa es la idea principal del módulo `os`.

Adenda. Un lector no estaba satisfecho con esta solución y quería poder ejecutar todas las pruebas unitarias en el directorio actual, no en aquél donde se encontrase `regression.py`. Sugirió este enfoque alternativo:

Ejemplo 16.6. Ejecución de *scripts* en el directorio actual

```
import sys, os, re, unittest

def regressionTest():
    path = os.getcwd()       ❶
    sys.path.append(path)    ❷
    files = os.listdir(path) ❸
```

❶ En lugar de poner en `path` el directorio donde está situado el script que está ejecutándose, ponemos el directorio actual de trabajo. Éste será el directorio donde estuviésemos en el momento de ejecutarlo, y no tiene por qué coincidir necesariamente con el del *script* (lea esta frase un par de veces hasta que la entienda bien).

❷ Añada este directorio a la ruta de búsqueda de bibliotecas de Python, para que Python pueda encontrar los módulos de pruebas unitarias cuando los vaya a importar dinámicamente. No necesitamos hacer esto cuando `path` coincide con el directorio del *script* en ejecución, porque Python siempre mira en ese directorio.

❸ El resto de la función es igual.

Esta técnica le permitirá reutilizar este *script* `regression.py` en varios proyectos. Simplemente lo pone en un directorio común, y cambia al directorio del proyecto antes de ejecutarlo. Encontrará y ejecutará todas las pruebas unitarias de ese proyecto en lugar de las que se encuentren en el mismo directorio de `regression.py`.

16.3. Revisión del filtrado de listas

Ya está familiarizado con el uso de listas por comprensión para filtrar listas. Hay otra manera de conseguir lo mismo que algunas personas consideran más expresiva.

Python incorpora una función `filter` que toma dos argumentos, una función y una lista, y devuelve una lista.[18] La función que se pasa como primer argumento a `filter` debe a su vez tomar un argumento, y la lista que devuelva `filter` contendrá todos los elementos de la lista que se le pasó a `filter` para los cuales la función del primer argumento devuelve verdadero.

¿Lo entendió? No es tan difícil como suena.

Ejemplo 16.7. Presentación de `filter`

```
>>> def odd(n):                ❶
...     return n % 2
...
>>> li = [1, 2, 3, 5, 9, 10, 256, -3]
>>> filter(odd, li)            ❷
[1, 3, 5, 9, -3]
>>> [e for e in li if odd(e)]  ❸
>>> filteredList = []
>>> for n in li:               ❹
...     if odd(n):
...         filteredList.append(n)
...
>>> filteredList
[1, 3, 5, 9, -3]
```

❶ `odd` usa la función incorporada de módulo "`%`" para devolver `True` si `n` es impar y `False` si `n` es par.

❷ `filter` toma dos argumentos, una función (`odd`) y una lista (`li`). Itera sobre la lista e invoca `odd` con cada elemento. Si `odd` devuelve verdadero (recuerde, cualquier valor diferente a cero es verdadero en Python), entonces se incluye el elemento en la lista devuelta, en caso contrario se filtra. El resultado es una lista con sólo los números impares que había en la original, en el mismo orden en que aparecían allí.

❸ Puede conseguir lo mismo usando listas por comprensión, como vio en Sección 4.5, "Filtrado de listas".

❹ Podría hacer lo mismo también con un bucle `for`. Dependiendo de su bajage como programador, esto puede parecer más "directo", pero las funciones como `filter` son mucho más expresivas. No sólo son más fáciles de escribir, sino también de leer. Leer el bucle `for` es como estar demasiado cerca de una pintura; ve todos los detalles pero puede llevarle unos segundos dar unos pasos atrás y ver todo el cuadro: "¡Oh, está filtrando la lista!"

Ejemplo 16.8. `filter` en `regression.py`

```
files = os.listdir(path)                                    ❶
test = re.compile("test\.py$", re.IGNORECASE)               ❷
files = filter(test.search, files)                          ❸
```

❶ Como vio en Sección 16.2, "Encontrar la ruta", `path` puede contener la ruta completa o parcial del directorio del *script* que se está ejecutando o una cadena vacía si se ejecutó desde el directorio actual. De cualquier manera, `files` acabará teniendo los nombres de los ficheros que están en el mismo directorio que el *script*.

❷ Esto es una expresión regular compilada. Como vio en Sección 15.3, "Refactorización", si va a utilizar la misma expresión regular una y otra vez debería compilarla para que rinda mejor. El fichero compilado tiene un método `search` que toma un solo argumento, la cadena a buscar. Si la expresión regular coincide con la cadena, el método `search` devuelve un

objeto `Match` que contiene información sobre la coincidencia; en caso contrario devuelve `None`, el valor nulo de Python.

❸ Vamos a invocar el método `search` del objeto que contiene la expresión regular compilada (`test`) con cada elemento de la lista `files`. Si coincide la expresión regular, el método devolverá un objeto `Match` que Python considera verdadero, así que el elemento se incluirá en la lista devuelta por `filter`. Si la expresión regular no coincide el método `search` devolverá `None` que Python considera falso, así que no se incluirá el elemento.

Nota histórica. Las versiones de Python anteriores a la 2.0 no tienen listas por comprensión, así que no podría filtrar usándolas; la función `filter`, era lo único a lo que se podía jugar. Incluso cuando se introdujeron las listas por comprensión en 2.0 algunas personas siguieron prefiriendo la vieja `filter` (y su compañera `map`, que veremos en este capítulo). Ambas técnicas funcionan y la que deba usar es cuestión de estilo. Se está comentando que `map` y `filter` podrían quedar obsoletas en una versión futura de Python, pero no se ha tomado aún una decisión.

Ejemplo 16.9. Filtrado usando listas de comprensión esta vez

```
files = os.listdir(path)
test = re.compile("test\.py$", re.IGNORECASE)
files = [f for f in files if test.search(f)] ❶
```

❶ Con esto conseguirá exactamente el mismo resultado que si usase la función `filter`. ¿Cual es más expresiva? Eso lo decide usted.

Footnotes

[18] Técnicamente, el segundo argumento a `filter` puede ser cualquier secuencia, incluyendo listas, tuplas y clases que actúen como listas definiendo el método especial `__getitem__`. Si es posible, `filter` devolverá el mismo tipo que le pasó, así que filtrar una lista devuelve una lista pero filtrar una tupla devuelve una tupla.

16.4. Revisión de la relación de listas

Ya está familiarizado con el uso de listas por comprensión para hacer corresponder una a otra. Hay otra manera de conseguir lo mismo usando la función incorporada `map`. Funciona muy parecido a `filter`.

Ejemplo 16.10. Presentación de `map`

```
>>> def double(n):
...     return n*2
...
>>> li = [1, 2, 3, 5, 9, 10, 256, -3]
>>> map(double, li)                          ❶
[2, 4, 6, 10, 18, 20, 512, -6]
>>> [double(n) for n in li]                  ❷
[2, 4, 6, 10, 18, 20, 512, -6]
>>> newlist = []
>>> for n in li:                             ❸
...     newlist.append(double(n))
...
>>> newlist
[2, 4, 6, 10, 18, 20, 512, -6]
```

❶ `map` toma una función y una lista[19] y devuelve una lista nueva invocando la función sobre cada elemento de la lista, por orden. En este caso la función simplemente multiplica cada elemento por 2.

❷ Podría conseguir lo mismo con una lista por comprensión. Las listas por comprensión se introdujeron por primera vez en Python 2.0; `map` lleva ahí desde el principio.

❸ Podría usar un bucle `for` para conseguir lo mismo, si insistiera en pensar como un programador de Visual Basic.

Ejemplo 16.11. `map` con listas de tipos de datos distintos

```
>>> li = [5, 'a', (2, 'b')]
>>> map(double, li)                          ❶
[10, 'aa', (2, 'b', 2, 'b')]
```

❶ Como nota de interés, quisiera señalar que `map` funciona también con listas de tipos datos distintos, siempre que la función los trate correctamente. En este caso la función `double` se limita a multiplicar un argumento dado por 2, y Python hace Lo Correcto dependiendo del tipo de dato del argumento. Para los enteros significa multiplicarlos por 2; para las cadenas significa concadenar la cadena consigo misma; para las tiplas significa crear una nueva tupla que tenga todos los elementos de la original, y luego de nuevo esos elementos.

Bien, basta de jugar. Veamos algo de código real.

Ejemplo 16.12. `map` en `regression.py`

```
filenameToModuleName = lambda f: os.path.splitext(f)[0] ❶
moduleNames = map(filenameToModuleName, files)          ❷
```

❶ Como vio en Sección 4.7, "Utilización de las funciones lambda", `lambda` define una función en línea (*inline*). Y también vio en Ejemplo 6.17, "Dividir nombres de rutas" que `os.path.splitext` toma el nombre de un fichero y devuelve una tupla (`nombre, extensión`). Así que `filenameToModuleName` es una función que tomará un nombre de fichero y eliminará la extensión para devolver sólo el nombre.

❷ La invocación a `map` toma cada nombre de fichero listado en `files`, lo pasa a la función `filenameToModuleName`, y devuelve una lista de los valores de retorno de cada una de estas invocaciones. En otras palabras, elimina la extensión de fichero de cada nombre y almacena la lista de estos nombres sin extensión en `moduleNames`.

Como verá en el resto del capítulo, podemos extender este tipo de forma de pensar centrada en los datos hasta el objetivo final, que es definir y ejecutar una única batería de pruebas que contenga las pruebas de todas esas otras baterías a menor escala.

Footnotes

[19] De nuevo, debería señalar que `map` puede tomar una lista, una tupla o cualquier objeto que actúe como una secuencia. Vea la nota anterior sobre `filter`.

16.5. Programación "datocéntrica"

Ahora es probable que esté rascándose la cabeza preguntándose por qué es mejor esto que usar bucles `for` e invocar directamente a las funciones. Y es una pregunta perfectamente válida. En su mayoría es una cuestión de perspectiva. Usar `map` y `filter` le fuerza a centrar sus pensamientos sobre los datos.

En este caso hemos empezado sin datos; lo primero que hicimos fue obtener la ruta del directorio del *script* en ejecución y la lista de ficheros en ese directorio. Eso fue la preparación y nos dio datos reales con los que trabajar: una lista de nombres de fichero.

Sin embargo, sabíamos que no nos interesaban todos esos ficheros sino sólo los que hacen las baterías de prueba. Teníamos *demasiados datos*, así que necesitabamos `filter`-arlos. ¿Cómo sabíamos qué datos mantener? Necesitábamos una prueba que lo decidiese así que definimos uno y se lo pasamos a la función `filter`. En este caso hemos usado una expresión regular para decidir, pero el concepto sería el mismo independientemente de la manera en que construyésemos la prueba.

Ahora teníamos los nombres de fichero de cada una de las baterías de prueba (y sólo eso, ya que hemos filtrado el resto), pero lo que queríamos en realidad eran nombres de módulos. Teníamos la cantidad de datos correctos, pero estaban *en el formato erróneo*. Así que definimos una función que transformara un nombre de fichero en el de un módulo, y pasamos la función sobre la lista entera. De un nombre de fichero podemos sacar el nombre de un módulo; de una lista de nombres de ficheros podemos obtener una lista de nombres de módulos.

Podíamos haber usado un bucle `for` con una sentencia `if` en lugar de `filter`. En lugar de `map` podríamos haber usado un bucle `for` con una llamada a

función. Pero usar bucles `for` de esa manera es trabajoso. En el mejor de los casos simplemente es un desperdicio de tiempo; y en el peor introduce fallos difíciles de detectar. Por ejemplo, necesitamos imaginar la manera de comprobar la condición "¿este fichero es una batería de pruebas?" de todas maneras; és la lógica específica de la aplicación y ningún lenguaje puede escribir eso por nosotros. Pero una vez que ya lo tiene, ¿de verdad quiere tener que crear una nueva lista vacía y escribir un bucle `for` y una sentencia `if` y luego llamar manualmente a `append` por cada elemento de la nueva lista si pasa la condición y llevar un seguimiento de qué variable lleva los datos filtrados y cual los que no están filtrados? ¿Por qué no limitarse a definir la condición de prueba y dejar que Python haga el resto del trabajo por nosotros?

Sí, claro, podría intentar ser imaginativo y borrar elementos de la lista original sin crear una nueva. Pero eso seguro que ya se ha quemado con eso antes. Intentar modificar una estructura de datos sobre la que se está iterando puede tener truco. Borramos un elemento, iteramos sobre el siguiente y de repente nos hemos saltado uno. ¿Es Python uno de los lenguajes que funciona así? ¿Cuánto nos llevaría averiguarlo? ¿Estamos seguros de recordar si era seguro la siguiente vez que lo intentemos? Los programadores pierden tanto tiempo y cometente tantos errores tratando con problemas puramente técnicos como este, y encima no tiene sentido. No hace avanzar nuestro programa; es todo trabajo en vano.

Me resistí a las listas por comprensión cuando aprendí Python, y me resistí a `filter` y `map` todavía más. Insistí en hacer mi vida más complicada quedándome con las familiares formas de los bucles `for` y las sentencias `if` y la programación paso a paso centrada en el código. Y mis programas en Python se parecían mucho a los de Visual Basic, detallando cada paso de cada operación de cada función. Y todos tenían los mismos tipos de pequeños problemas y fallos complicados de encontrar. Y nada de esto tenía sentido.

Déjelo todo. Ese trabajo burocrático no es importante. Lo importante son los datos. Y los datos no son difíciles. Son sólo datos. Si tiene muchos, fíltrelos. Si

no son los que quiere, “mapéelos”. Céntrese en los datos; deje atrás el trabajo pesado.

16.6. Importación dinámica de módulos

Bien, basta de filosofar. Hablemos sobre la importación dinámica de módulos.

Miremos primero la manera normal de importar módulos. La sintaxis de `import módulo` localiza por su nombre el módulo especificado en la ruta de búsqueda y lo importa. Incluso puede importar varios módulos de una vez de esta manera, con una lista separada por comas. Hicimos esto en la primera línea del *script* de este capítulo.

Ejemplo 16.13. Importación de varios módulos a la vez

```
import sys, os, re, unittest ❶
```

❶ Esto importa cuatro módulos de unavez: `sys` (con las funciones del sistema y acceso a parámetros de línea de órdenes), `os` (con funciones del sistema operativo como listado de directorios), `re` (para las expresiones regulares) y `unittest` (para las pruebas unitarias).

Ahora haremos lo mismo pero de forma dinámica.

Ejemplo 16.14. Importación dinámica de módulos

```
>>> sys = __import__('sys')          ❶
>>> os = __import__('os')
>>> re = __import__('re')
>>> unittest = __import__('unittest')
>>> sys                              ❷
>>> <module 'sys' (built-in)>
>>> os
>>> <module 'os' from '/usr/local/lib/python2.2/os.pyc'>
```

❶ La función incorporada `__import__` cumple el mismo objetivo de la sentencia `import`, pero es una función y toma una cadena como argumento.

❷ Ahora la variable `sys` es el módulo `sys`, igual que si hubiera escrito `import sys`. La variable `os` es ahora el módulo `os`, *etc.*.

Así que `__import__` importa un módulo, pero toma una cadena como argumento para hacerlo. En este caso el módulo que importó era sólo una cadena fija, pero podría ser también una variable o el resultado de invocar una función. Y la variable a la que asignemos el módulo tampoco tiene por qué coincidir con el nombre del módulo. Podría importar varios módulos y asignarlos a una lista.

Ejemplo 16.15. Importación de una lista de módulos de forma dinámica

```
>>> moduleNames = ['sys', 'os', 're', 'unittest'] ❶
>>> moduleNames
['sys', 'os', 're', 'unittest']
>>> modules = map(__import__, moduleNames)        ❷
>>> modules                                       ❸
[<module 'sys' (built-in)>,
<module 'os' from 'c:\Python22\lib\os.pyc'>,
<module 're' from 'c:\Python22\lib\re.pyc'>,
<module 'unittest' from 'c:\Python22\lib\unittest.pyc'>]
>>> modules[0].version                            ❹
'2.2.2 (#37, Nov 26 2002, 10:24:37) [MSC 32 bit (Intel)]'
>>> import sys
>>> sys.version
'2.2.2 (#37, Nov 26 2002, 10:24:37) [MSC 32 bit (Intel)]'
```

❶ `moduleNames` es una lista de cadenas. Nada del otro mundo si exceptuamos que las cadenas resultan ser los nombres de módulos que podríamos importar, si así quisiéramos.

❷ Sorpresa, queríamos importarlos, y lo hicimos relacionando la función `__import__` sobre la lista. Recuerde que esto toma cada elemento de la lista (`moduleNames`) e invoca la función (`__import__`) una y otra vez, sobre cada elemento de la lista; construye una lista de valores de retorno y devuelve el resultado.

❸ Así que partiendo de una lista de cadenas hemos creado una lista de módulos. (Las rutas de usted pueden ser diferentes, dependiendo del sistema operativo, dónde haya instalado Python, la fase de la luna, *etc.*).

❹ Para comprobar que estos son módulos reales, veamos algunos de sus atributos. Recuerde que `modules[0]` *es* el módulo `sys`, así que `modules[0].version` *es* `sys.version`. También están disponibles todos los otros atributos y métodos de esos módulos. No hay nada mágico respecto a la sentencia `import`, y tampoco respecto a los módulos. Los módulos son objetos. Todo es un objeto.

Ahora debería ser capaz de juntar todo esto e imaginarse lo que hace la mayoría del código de ejemplo de este capítulo.

16.7. Todo junto

Hemos aprendido suficiente para hacer disección de las primeras siete líneas del código de ejemplo de este capítulo: lectura de un directorio e importación de los módulos seleccionados dentro de él.

Ejemplo 16.16. La función `regressionTest`

```
def regressionTest():
    path = os.path.abspath(os.path.dirname(sys.argv[0]))
    files = os.listdir(path)
    test = re.compile("test\.py$", re.IGNORECASE)
    files = filter(test.search, files)
    filenameToModuleName = lambda f: os.path.splitext(f)[0]
    moduleNames = map(filenameToModuleName, files)
    modules = map(__import__, moduleNames)
load = unittest.defaultTestLoader.loadTestsFromModule
return unittest.TestSuite(map(load, modules))
```

Veámoslo línea por línea, de forma interactiva. Asuma que el directorio actual es `c:\diveintopython\py`, que contiene los ejemplos que acompañan a este libro incluido el *script* de este capítulo. Como vimos en Sección 16.2, “Encontrar la

ruta" el directorio del *script* acabará en la variable `path`, así que empecemos a trabajar sobre eso y partamos desde ahí.

Ejemplo 16.17. Paso 1: Obtener todos los ficheros

```
>>> import sys, os, re, unittest
>>> path = r'c:\diveintopython\py'
>>> files = os.listdir(path)
>>> files ❶
['BaseHTMLProcessor.py', 'LICENSE.txt', 'apihelper.py',
'apihelpertest.py',
'argecho.py', 'autosize.py', 'builddialectexamples.py', 'dialect.py',
'fileinfo.py', 'fullpath.py', 'kgptest.py', 'makerealworddoc.py',
'odbchelper.py', 'odbchelpertest.py', 'parsephone.py', 'piglatin.py',
'plural.py', 'pluraltest.py', 'pyfontify.py', 'regression.py',
'roman.py', 'romantest.py',
'uncurly.py', 'unicode2koi8r.py', 'urllister.py', 'kgp', 'plural',
'roman',
'colorize.py']
```

❶ `files` es una lista de todos los ficheros y directorios dentro de aquél donde se encuentra el *script*. (Si ha estado ejecutando algunos de los ejemplos puede que ya haya visto ficheros `.pyc` también).

Ejemplo 16.18. Paso 2: Filtrar para obtener los ficheros que nos interesan

```
>>> test = re.compile("test\.py$", re.IGNORECASE)          ❶
>>> files = filter(test.search, files)                     ❷
>>> files                                                  ❸
['apihelpertest.py', 'kgptest.py', 'odbchelpertest.py',
'pluraltest.py', 'romantest.py']
```

❶ Esta expresión regular coincidirá con cualquier cadena que termine en `test.py`. Observe que debe de "escapar" el punto, ya que en una expresión regular normalmente significa "coincide con cualquier carácter individual", pero en realidad queremos que represente un punto literal.

❷ La expresión regular compilada actúa como una función, así que podemos usarla para filtrar la gran lista de ficheros y directorios, para encontrar los

que coinciden con ella.

❸ Y nos queda la lista de *scripts* de pruebas unitarias, que eran los únicos con nombres `ALGOtest.py`.

Ejemplo 16.19. Paso 3: Relacionar los nombres de fichero a los de módulos

```
>>> filenameToModuleName = lambda f: os.path.splitext(f)[0] ❶
>>> filenameToModuleName('romantest.py')                    ❷
'romantest'
>>> filenameToModuleName('odchelpertest.py')
'odbchelpertest'
>>> moduleNames = map(filenameToModuleName, files)          ❸
>>> moduleNames                                             ❹
['apihelpertest', 'kgptest', 'odbchelpertest', 'pluraltest',
'romantest']
```

❶ Como vio en Sección 4.7, "Utilización de las funciones lambda", `lambda` es una manera sucia y rápida de crear una función de una sola línea. Ésta toma un nombre de fichero con extensión y devuelve sólo la parte del nombre de fichero, usando la función estándar de biblioteca `os.path.splitext` que ya vio en Ejemplo 6.17, "Dividir nombres de rutas".

❷ `filenameToModuleName` es una funci`ón. No hay nada mágico en las funciones `lambda` respecto a las funciones normales que definimos con una sentencia `def`. Podemos invocar la función `filenameToModuleName` como cualquier otra y hace justo lo que queríamos que hiciese: elimina la extensión de su argumento.

❸ Ahora podemos aplicar esta función a cada fichero de la lista de ficheros de pruebas unitarias, usando `map`.

❹ Y el resultado es justo lo que queríamos: una lista de módulos como cadenas.

Ejemplo 16.20. Paso 4: Relacionar los nombres de módulos a módulos

```
>>> modules = map(__import__, moduleNames)                  ❶
```

```
>>> modules                                                    ❷
[<module 'apihelpertest' from 'apihelpertest.py'>,
<module 'kgptest' from 'kgptest.py'>,
<module 'odbchelpertest' from 'odbchelpertest.py'>,
<module 'pluraltest' from 'pluraltest.py'>,
<module 'romantest' from 'romantest.py'>]
>>> modules[-1]                                               ❸
<module 'romantest' from 'romantest.py'>
```

❶ Como vio en Sección 16.6, "Importación dinámica de módulos", podemos usar una combinación de `map` e `__import__` para relacionar una lista de nombres de módulos (como cadenas) en módulos reales (a los que podemos invocar o acceder igual que a cualquier otro).

❷ Ahora `modules` es una lista de módulos, completametne accesible igual que cualquier otro módulo.

❸ El siguiente módulo de la lista *es* el `romantest`, como si hubiéramos dicho `import romantest`.

Ejemplo 16.21. Paso 5: Cargar los módulos en la batería de pruebas

```
>>> load = unittest.defaultTestLoader.loadTestsFromModule
>>> map(load, modules)                                        ❶
[<unittest.TestSuite tests=[
  <unittest.TestSuite tests=[<apihelpertest.BadInput
testMethod=testNoObject>]>,
  <unittest.TestSuite tests=[<apihelpertest.KnownValues
testMethod=testApiHelper>]>,
  <unittest.TestSuite tests=[
    <apihelpertest.ParamChecks testMethod=testCollapse>,
    <apihelpertest.ParamChecks testMethod=testSpacing>]>,
    ...
  ]
]
>>> unittest.TestSuite(map(load, modules))                    ❷
```

❶ Éstos son objetos de módulo reales. No sólo puede acceder a ellos igual que a cualquier otro módulo, instanciar clases y llamar a funciones, sino que puede introspeccionar el módulo y averiguar qué clases y funciones contiene. Eso es

lo que hace el método `loadTestsFromModule`: introspecciona cada módulo y devuelve un objeto `unittest.TestSuite` por cada uno. Cada objeto `TestSuite` contiene realmente una lista de objetos `TestSuite`, uno por cada clase `TestCase` del módulo, y cada uno de esos objetos `TestSuite` contiene una lista de pruebas, una por cada método de prueba del módulo.

❷ Por último, metemos la lista de objetos `TestSuite` dentro de la gran batería de pruebas. El módulo `unittest` no tiene problemas en navegar por el árbol de baterías de pruebas anidadas dentro de otras; para acabar llegando a un método indivual de prueba y ejecutarlo, verificar si tiene éxito o falla, y pasar al siguiente.

Este proceso de introspección es lo que hace normalmente el módulo `unittest`. ¿Recuerda aquella función `unittest.main()` casi mágica que invocaban nuestros módulos de prueba individuales para poner en marcha todo? `unittest.main()` crea en realidad una instancia de `unittest.TestProgram`, que a su vez crea una instancia de un `unittest.defaultTestLoader` y lo carga con el módulo que la invocó. (¿Cómo obtiene una referencia al módulo que la invocó si no se le hemos facilitado uno? Usando la orden de mismo aspecto mágico `__import__('__main__')`, que importa dinámicamente el módulo que está ejecutándose ahora. Podría escribir un libro con todos los trucos y técnicas que se usan en el módulo `unittest`, pero entonces no acabaría éste nunca).

Ejemplo 16.22. Paso 6: Decirle a `unittest` que use nuestra batería de pruebas

```
if __name__ == "__main__":
    unittest.main(defaultTest="regressionTest") ❶
```

❶ En lugar de permitir que el módulo `unittest` haga toda su magia por nosotros, hemos hecho casi todo nosotros mismos. Hemos creado una función (`regressionTest`) que importa los módulos, invocamos a `unittest.defaultTestLoader` y juntamos todo en una batería de pruebas. Ahora todo lo que necesitamos hacer es decirle a `unittest` que en lugar de

buscar pruebas y construir una batería de pruebas de la manera normal, debería llamar simplemente a la función `regressionTest` que devuelve una `TestSuite` lista para usar.

16.8. Resumen

El programa `regression.py` y sus salidas deberían tener perfecto sentido.

Debería sentirse cómodo haciendo todas estas cosas:

- Manipular información de ruta desde la línea de órdenes.
- Filtrar listas usando `filter` en lugar de listas por comprensión.
- Relacionar listas usando `map` en lugar de listas por comprensión.
- Importar módulos de forma dinámica.

Capítulo 17. Funciones dinámicas

17.1. Inmersión

Quiero hablar sobre los sustantivos en plural. También sobre funciones que devuelven otras funciones, expresiones regulares avanzadas y generadores. Los generadores son nuevos en Python 2.3. Pero primero hablemos sobre cómo hacer nombres en plural[20].

Si no ha leído Capítulo 7, *Expresiones regulares* ahora es el momento. Este capítulo asume que comprende lo básico de las expresiones regulares, y desciende rápidamente a usos más avanzados.

El inglés es un idioma esquizofrénico que toma préstamos de otros muchos, y las reglas para hacer plurales los sustantivos singulares son variadas y complejas. Hay reglas, y luego excepciones a esas reglas, y excepciones a las excepciones.

Si se crió en un país de habla inglesa o aprendió inglés durante su educación formal, probablemente conozca las reglas básicas:

1. Si una palabra termina en S, X o Z, añada ES. “Bass” pasa a ser “bases”, “fax” se convierte en “faxes” y “waltz” será “waltzes”.
2. Si una palara termina en una H sonora, añada ES; si acaba en una H sorda, añada simplemente S. ¿Qué es una H sonora? Una que puede ser

combinada con otras letras para hacer un sonido audible. Así que "coach" será "coaches" y "rash" pasa a ser "rashes", porque puede oír los sonidos CH y SH cuando los pronuncia. Pero "cheetah" se convierte en "cheetahs", porque la H es sorda.

3. Si una palabra termina en Y que suena como I, cambie la Y por IES; si la Y se combina con una vocal y suena de otra manera, añada sólo S. Así que "vacancy" pasa a ser "vacancies", pero "day" será "days".
4. Si todo lo demás falla, añada una S y espere que sea así.

(Lo sé, hay un montón de excepciones. "Man" es "men" en plural y "woman" es "women", pero "human" es "humans". "Mouse" es "mice" y "louse" es "lice", pero "house" se convierte en "houses". "Knife" será "knives" y "wife", "wives", pero "lowlife" en plural es "lowlifes". Y no quiero empezar con las palabras que son su propio plural como "sheep", "deer" y "haiku").

En otros idiomas es completamente diferente, por supuesto.

Diseñemos un módulo que pluralice sustantivos. Empecemos sólo sustantivos ingleses y sólo estas cuatro reglas, pero pensemos que inevitablemente vamos a necesitar añadir más reglas, y puede que más adelante necesitemos añadir más idiomas.

Footnotes

[20] N. del T.: por supuesto, Mark se refiere a los plurales en inglés. Téngalo en cuenta el resto del capítulo, puesto que no voy a adaptarlo a los sustantivos en castellano para no tener que modificar los ejemplos.

17.2. `plural.py`, fase 1

Así que estamos mirando las palabras, que al menos en inglés son cadenas de caracteres. Y tenemos reglas que dicen que debe encontrar diferentes combinaciones de caracteres y luego hacer cosas distintas con ellos. Esto suena a trabajo para las expresiones regulares.

Ejemplo 17.1. `plural1.py`

```
import re

def plural(noun):
    if re.search('[sxz]$', noun):                ❶
        return re.sub('$', 'es', noun)           ❷
    elif re.search('[^aeioudgkprt]h$', noun):
        return re.sub('$', 'es', noun)
    elif re.search('[^aeiou]y$', noun):
        return re.sub('y$', 'ies', noun)
    else:
        return noun + 's'
```

❶ Bien, esto es una expresión regular, pero usa una sintaxis que no vimos en Capítulo 7, *Expresiones regulares*. Los corchetes significan "coincide exactamente con uno de estos caracteres". Así que `[sxz]` significa "`s`, o `x`, o `z`", pero sólo una de ellas. La `$` debería serle conocida; coincide con el final de la cadena. Así que estamos buscando si `noun` termina en `s`, `x` o `z`.

❷ Esta función `re.sub` realiza sustituciones en cadenas basándose en expresiones regulares. Echemos un vistazo más detallado.

Ejemplo 17.2. Presentación de `re.sub`

```
>>> import re
>>> re.search('[abc]', 'Mark')    ❶
<_sre.SRE_Match object at 0x001C1FA8>
>>> re.sub('[abc]', 'o', 'Mark') ❷
'Mork'
>>> re.sub('[abc]', 'o', 'rock') ❸
'rook'
>>> re.sub('[abc]', 'o', 'caps') ❹
'oops'
```

❶ ¿Contiene la cadena `Mark` una `a`, `b` o `c`? Sí, contiene una `a`.

❷ Vale, ahora encontramos `a`, `b` o `c` y la sustituimos por `o`. `Mark` pasa a ser `Mork`.

❸ La misma función convierte `rock` en `rook`.

❹ Puede que piense que esto convertirá `caps` en `oaps`, pero no es así. `re.sub` sustituye *todas* las coincidencias, no sólo la primera. Así que esta expresión regular convierte `caps` en `oops`, porque tanto la `c` como la `a` se convierten en `o`.

Ejemplo 17.3. De vuelta a `plural1.py`

```
import re

def plural(noun):
    if re.search('[sxz]$', noun):
        return re.sub('$', 'es', noun)          ❶
    elif re.search('[^aeioudgkprt]h$', noun): ❷
        return re.sub('$', 'es', noun)          ❸
    elif re.search('[^aeiou]y$', noun):
        return re.sub('y$', 'ies', noun)
    else:
        return noun + 's'
```

❶ Volvamos a la función `plural`. ¿Qué estamos haciendo? Estamos cambiando el final de la cadena por `es`. En otras palabras, añadimos `es` a la cadena. Podríamos conseguir lo mismo concatenando cadenas, por ejemplo `noun + 'es'`, pero estamos usando expresiones regulares para todo por consistencia, por razones que aclararé más adelante en el capítulo.

❷ Mire atentamente, ésta es otra variación nueva. La `^` como primer carácter dentro de los corchetes significa algo especial: negación. `[^abc]` significa "cualquier carácter individual *excepto* `a`, `b` o `c`". Así que `[^aeioudgkprt]` significa cualquier carácter excepto `a`, `e`, `i`, `o`, `u`, `d`, `g`, `k`, `p`, `r` o `t`. Además ese carácter ha de ir seguido por `h`, y luego por el final de la cadena. Estamos buscando palabras que terminen en H en las que la H se pueda oír.

❸ El mismo patrón aquí: coincide con palabras que terminan en Y donde el carácter anterior a la Y *no* sea `a`, `e`, `i`, `o` o `u`. Estamos buscando palabras que terminen en una Y que suene como I.

Ejemplo 17.4. Más sobre la negación en expresiones regulares

```
>>> import re
>>> re.search('[^aeiou]y$', 'vacancy') ❶
<_sre.SRE_Match object at 0x001C1FA8>
>>> re.search('[^aeiou]y$', 'boy')     ❷
>>>
>>> re.search('[^aeiou]y$', 'day')
>>>
>>> re.search('[^aeiou]y$', 'pita')    ❸
>>>
```

❶ `vacancy` coincide con esta expresión regular porque termina en `cy`, y `c` no es `a`, `e`, `i`, `o` o `u`.

❷ `boy` no coincide porque termina en `oy`, y dijimos específicamente que el carácter antes de la `y` no podía ser `o`. `day` tampoco coincide porque termina en `ay`.

❸ `pita` no coincide porque no termina en `y`.

Ejemplo 17.5. Más sobre `re.sub`

```
>>> re.sub('y$', 'ies', 'vacancy')             ❶
'vacancies'
>>> re.sub('y$', 'ies', 'agency')
'agencies'
>>> re.sub('([^aeiou])y$', r'\1ies', 'vacancy') ❷
'vacancies'
```

❶ Esta expresión regular convierte `vacancy` en `vacancies` y `agency` en `agencies`, que es lo que queríamos. Observe que también convertiría `boy` en `boies`, pero eso no sucederá nunca en la función porque ya hicimos esa `re.search` antes para ver si debíamos ejecutar esta `re.sub`.

❷ Querría señalar sólo de pasada que es posible combinar estas dos expresiones regulares (una para ver si se aplica la regla y otra para aplicarla) en una sola expresión regular. Éste es el aspecto que tendría. En gran parte le será familiar: estamos usando grupos a recordar, que ya aprendimos en Sección 7.6, “Caso de estudio: análisis de números de teléfono”, para guardar el carácter antes de la `y`. Y en la cadena de sustitución usamos una sintaxis nueva, `\1`, que significa “eh, ¿sabes el primer grupo que recordaste? ponlo

aquí". En este caso hemos recordado la `c` antes de la `y` y luego cuando hacemos la sustitución ponemos `c` en el sitio de la `c`, e `ies` en el sitio de la `y` (si tiene más de un grupo que recordar puede usar `\2`, `\3`, *etc.*).

La sustitución de expresiones regulares es extremadamente potente, y la sintaxis `\1` la hace aún más poderosa. Pero combinar la operación entera en una sola expresión regular es mucho más complejo de leer y no se relaciona directamente con la manera en que describimos al principio las reglas de pluralización. Originalmente describimos las reglas como "si la palabra termina en S, X o Z, entonces añade ES". Y si miramos a esta función, tenemos dos líneas de código que dicen "si la palabra termina en S, X o Z, entonces añadeES". No se puede ser mucho más directo que eso.

17.3. `plural.py`, fase 2

Ahora vamos a añadir un nivel de abstracción. Empezamos definiendo una lista de reglas: si pasa esto, haz lo aquello, si no pasa la siguiente regla.
Compliquemos temporalmente parte del programa para poder simplificar otra.

Ejemplo 17.6. `plural2.py`

```
import re

def match_sxz(noun):
    return re.search('[sxz]$', noun)

def apply_sxz(noun):
    return re.sub('$', 'es', noun)

def match_h(noun):
    return re.search('[^aeioudgkprt]h$', noun)

def apply_h(noun):
    return re.sub('$', 'es', noun)

def match_y(noun):
```

```
    return re.search('[^aeiou]y$', noun)

def apply_y(noun):
    return re.sub('y$', 'ies', noun)

def match_default(noun):
    return 1

def apply_default(noun):
    return noun + 's'

rules = ((match_sxz, apply_sxz),
         (match_h, apply_h),
         (match_y, apply_y),
         (match_default, apply_default)
         )                                          ❶

def plural(noun):
    for matchesRule, applyRule in rules:            ❷
        if matchesRule(noun):                       ❸
            return applyRule(noun)                  ❹
```

❶ Esta versión parece más complicada (ciertamente es más larga), pero hace exactamente lo mismo: comprueba cuatro reglas en orden, y aplica la expresión regular apropiada cuando encuentra una coincidencia. La diferencia es que cada coincidencia y regla a aplicar individuales están definidas en su propia función, y las funciones están listadas en esta variable `rules` que es una tupla de tuplas.

❷ Usando un bucle `for` podemos tomar la coincidencia y las reglas a aplicar de dos en dos (una coincidencia, una regla) desde la tupla `rules`. Durante la primera iteración del bucle `for` `matchesRule` valdrá `match_sxz` y `applyRule` valdrá `apply_sxz`. En la segunda iteración (asumiendo que lleguemos tan lejos), se le asignará `match_h` a `matchesRule` y `apply_h` a `applyRule`.

❸ Recuerde que todo en Python es un objeto, incluidas las funciones. `rules` contiene funciones; no nombres de funciones sino funciones reales. Cuando se las asigna en el bucle `for`, `matchesRule` y `applyRule` son las funciones a las que llamamos. Así que en la primera iteración del bucle `for`, esto es

equivalente a invocar `matches_sxz(noun)`.

❹ En la primera iteración del bucle `for` esto es equivalente a invocar `apply_sxz(noun)`, *etc.*.

Si encuentra confuso este nivel de abstracción trate de "desenrollar" la función para ver la equivalencia. Este bucle `for` es el equivalente a lo siguiente;

Ejemplo 17.7. Desarrollo de la función `plural`

```
def plural(noun):
    if match_sxz(noun):
        return apply_sxz(noun)
    if match_h(noun):
        return apply_h(noun)
    if match_y(noun):
        return apply_y(noun)
    if match_default(noun):
        return apply_default(noun)
```

Aquí el beneficio es que esa función `plural` queda simplificada. Toma una lista de reglas definidas en otra parte e itera sobre ellas de manera genérica. Tomamos una regla de coincidencia. ¿Coincide? Entonces llamamos a la regla de modificación. Las reglas se pueden definir en cualquier sitio, de cualquier manera. A la función `plural` no le importa.

Ahora bien, ¿merecía la pena añadir este nivel de abstracción? Bueno, aún no. Consideremos lo que implicaría añadir una nueva regla a la función. En el ejemplo anterior hubiera precisado añadir una sentencia `if` a la función `plural`. En este ejemplo precisaría añadir dos funciones, `match_foo` y `apply_foo`, y luego actualizar la lista `rules` para especificar en qué orden deberían invocarse las nuevas funciones con relación a las otras reglas.

Esto es simplemente un hito en el camino a la siguiente sección. Prosigamos.

17.4. `plural.py`, fase 3

No es realmente necesario definir una función aparte para cada regla de coincidencia y modificación. Nunca las invocamos directamente; las definimos en la lista `rules` y las llamamos a través suya. Vamos a reducir el perfil de la definición de las reglas haciéndolas anónimas.

Ejemplo 17.8. `plural3.py`

```
import re

rules = \
  (
    (
     lambda word: re.search('[sxz]$', word),
     lambda word: re.sub('$', 'es', word)
    ),
    (
     lambda word: re.search('[^aeioudgkprt]h$', word),
     lambda word: re.sub('$', 'es', word)
    ),
    (
     lambda word: re.search('[^aeiou]y$', word),
     lambda word: re.sub('y$', 'ies', word)
    ),
    (
     lambda word: re.search('$', word),
     lambda word: re.sub('$', 's', word)
    )
   )                                                  ❶

def plural(noun):
    for matchesRule, applyRule in rules:              ❷
        if matchesRule(noun):
            return applyRule(noun)
```

❶ Éste es el mismo conjunto de reglas que definimos en la fase 2. La única diferencia es que en lugar de definir reglas por nombre como `match_sxz` y `apply_sxz` hemos puesto "en línea" esas definiciones de función directamente en la propia lista `rules`, usando funciones lambda.

❷ Observe que la función `plural` no ha cambiado en nada. Itera sobre un conjunto de funciones de reglas, prueba la primera regla y si devuelve verdadero llama a la segunda regla y devuelve el valor. Lo mismo que antes, palabra por palabra. La única diferencia es que las funciones de reglas se definieron en línea, de forma anónima, usando funciones lambda. Pero a la función `plural` le da igual cómo estén definidas; simplemente obtiene una lista de reglas y las usa ciegamente.

Ahora todo lo que tenemos que hacer para añadir una regla es definir las funciones directamente en la lista `rules`: una regla para coincidencia y otra de transformación. Pero definir las funciones de reglas en línea de esta manera deja muy claro que tenemos algo de duplicidad innecesaria. Tenemos cuatro pares de funciones y todas siguen el mismo patrón. La función de comparación es una simple llamada a `re.search`, y la función de transformación es una simple llamada a `re.sub`. Saquemos el factor común de estas similaridades.

17.5. `plural.py`, fase 4

Eliminemos la duplicación del código para que sea más sencillo definir nuevas reglas.

Ejemplo 17.9. `plural4.py`

```
import re

def buildMatchAndApplyFunctions((pattern, search, replace)):
    matchFunction = lambda word: re.search(pattern, word)      ❶
    applyFunction = lambda word: re.sub(search, replace, word) ❷
    return (matchFunction, applyFunction)                      ❸
```

❶ `buildMatchAndApplyFunctions` es una función que construye otras de forma dinámica. Toma `pattern`, `search` y `replace` (en realidad toma una tupla, pero hablaremos sobre eso en un momento), y podemos construir la función de comparación usando la sintaxis `lambda` para que sea una función que toma

un parámetro (`word`) e invoca a `re.search` con el `pattern` que se pasó a la función `buildMatchAndApplyFunctions`, y la `word` que se pasó a la función de comparación que estamos construyendo. ¡Guau!

❷ La construcción la función de transformación se realiza igual. La función de transformación toma un parámetros y llama a `re.sub` con los parámetros `search` y `replace` que se pasaron a la función `buildMatchAndApplyFunctions`, y la `word` que se pasó a la función de transformación que estamos creando. Este técnica de usar los valores de parámetros externos dentro de una función dinámica se denomina *closure*[21]. Esencialmente estamos definiendo constantes dentro de la función de transformación que estamos creando: toma un parámetro (`word`), pero luego actúa sobre otros dos valores más (`search` y `replace`) cuyos valores se fijaron cuando definimos la función de transformación.

❸ Por último, la función `buildMatchAndApplyFunctions` devuelve una tupla con los dos valores: las dos funciones que acabamos de crear. Las constantes que definimos dentro de esas funciones (`pattern` dentro de `matchFunction`, y `search` y `replace` dentro de `applyFunction`) se quedan en esas funciones incluso tras volver de `buildMatchAndApplyFunctions`. ¡Esto es de locura!

Si esto es increíblemente confuso (y debería sero, es materia muy absurda), puede que quede más claro cuando vea cómo usarlo.

Ejemplo 17.10. continuación de `plural4.py`

```
patterns = \
  (
    ('[sxz]$', '$', 'es'),
    ('[^aeioudgkprt]h$', '$', 'es'),
    ('(qu|[^aeiou])y$', 'y$', 'ies'),
    ('$', '$', 's')
  )                                                    ❶
rules = map(buildMatchAndApplyFunctions, patterns)     ❷
```

❶ Nuestras reglas de pluralización están definidas ahora como una serie de cadenas (y no funciones). La primera cadena es la expresión regular que

usaremos en `re.search` para ver si la regla coincide; la segunda y la tercera son las expresiones de búsqueda y sustitución que usaríamos en `re.sub` para aplicar realmente la regla que convierte un sustantivo en su plural.

❷ Esta línea es magia. Toma la lista de cadenas de `patterns` y las convierte en una lista de funciones. ¿Cómo? Relacionando las cadenas con la función `buildMatchAndApplyFunctions`, que resulta tomar tres cadenas como parámetros y devolver una tupla con dos funciones. Esto significa que `rules` acaba siendo exactamente lo mismo que en el ejemplo anterior: una lista de tuplas donde cada tupla es un par de funciones, en que la primera función esl a de comparación que llama a `re.search`, y la segunda función es la de transformación que llama a `re.sub`.

Le juro que no me estoy inventando esto: `rules` acaba siendo exactamente la misma lista de funciones del ejemplo anterior. Desenrolle la definición de `rules` y obtendrá esto:

Ejemplo 17.11. Desarrollo de la definición de reglas

```
rules = \
  (
    (
     lambda word: re.search('[sxz]$', word),
     lambda word: re.sub('$', 'es', word)
    ),
    (
     lambda word: re.search('[^aeioudgkprt]h$', word),
     lambda word: re.sub('$', 'es', word)
    ),
    (
     lambda word: re.search('[^aeiou]y$', word),
     lambda word: re.sub('y$', 'ies', word)
    ),
    (
     lambda word: re.search('$', word),
     lambda word: re.sub('$', 's', word)
    )
   )
```

Ejemplo 17.12. final de `plural4.py`

```
def plural(noun):
    for matchesRule, applyRule in rules:           ❶
        if matchesRule(noun):
            return applyRule(noun)
```

❶ Ya que la lista `rules` es la misma del ejemplo anterior, no debería sorprenderle que no haya cambiado la función `plural`. Recuerde: es completamente genérica; toma una lista de funciones de reglas y las llama por orden. No le importa cómo se hayan definido. En la fase 2 se definieron como funciones con nombres diferentes. En la fase 3 se definieron como funciones `lambda` anónimas. Ahora en la fase 4 las estamos construyendo de forma dinámica relacionando la función `buildMatchAndApplyFunctions` sobre una lista de cadenas sin procesar. No importa; la función `plural` sigue funcionando de la misma manera.

Sólo por si acaso su mente no está ya suficientemente machacada, debo confesarle que había una sutileza en la definición de `buildMatchAndApplyFunctions` que no he explicado. Volvamos atrás a echarle otro vistazo.

Ejemplo 17.13. Otra mirada sobre `buildMatchAndApplyFunctions`

```
def buildMatchAndApplyFunctions((pattern, search, replace)):    ❶
```

❶ ¿Advierte los dobles paréntesis? Esta función no toma tres parámetros en realidad; sólo toma uno, una tupla de tres elementos. Pero la tupla se expande al llamar a la función y los tres elementos de la tupla se asignan cada uno a una variable diferente. `pattern`, `search` y `replace`. ¿Sigue confuso? Veámoslo en acción.

Ejemplo 17.14. Expansión de tuplas al invocar funciones

```
>>> def foo((a, b, c)):
```

```
...     print c
...     print b
...     print a
>>> parameters = ('apple', 'bear', 'catnap')
>>> foo(parameters) ❶
catnap
bear
apple
```

❶ La manera correcta de invocar las función `foo` es con una tupla de tres elementos. Cuando se llama a la función se asignan los elementos a diferentes variables locales dentro de `foo`.

Ahora vamos a ver por qué era necesario este truco de autoexpansión de tupla. `patterns` es una lista de tuplas y cada tupla tiene tres elementos. Cuando se invoca `map(buildMatchAndApplyFunctions, patterns)`, eso significa que *no* estamos llamando a `buildMatchAndApplyFunctions` con tres parámetros. Si usamos `map` para relacionar una única lista a una función siempre invocamos la función con un único parámetros: cada elemento de la lista. En el caso de `patterns`, cada elemento de la lista es una tupla, así que a `buildMatchAndApplyFunctions` siempre se le llama con la tupla, y usamos el truco de autoexpansión de tuplas en la definición de `buildMatchAndApplyFunctions` para asignar los elementos de la tupla a variables con las que podemos trabajar.

Footnotes

[21] N. del T.: o *cierre*

17.6. `plural.py`, fase 5

Hemos eliminado casi todo el código duplicado y añadido suficientes abstracciones para que las reglas de pluralización se definan en una lista de cadenas. El siguiente paso lógico es tomar estas cadenas y ponerlas en un fichero aparte, donde se puedan mantener de forma separada al código que las usa.

Primero vamos a crear un fichero de texto que contiene las reglas que queremos. Nada de estructuras de datos sofisticadas, sólo cadenas en tres columnas delimitadas por espacios (o tabulaciones). Lo llamaremos `rules.en`; "en" significa idioma inglés[22]. Éstas son las reglas de pluralización de sustantivos en inglés. Podríamos añadir otros ficheros de reglas más adelante para otros idiomas.

Ejemplo 17.15. `rules.en`

```
[sxz]$                  $               es
[^aeioudgkprt]h$        $               es
[^aeiou]y$              y$              ies
$                       $               s
```

Ahora veamos cómo podemos usar este fichero de reglas.

Ejemplo 17.16. `plural5.py`

```
import re
import string

def buildRule((pattern, search, replace)):
    return lambda word: re.search(pattern, word) and re.sub(search,
replace, word) ❶

def plural(noun, language='en'):                                    ❷
    lines = file('rules.%s' % language).readlines()                 ❸
    patterns = map(string.split, lines)                             ❹
    rules = map(buildRule, patterns)                                ❺
    for rule in rules:
        result = rule(noun)                                         ❻
        if result: return result
```

❶ Aquí estamos usando todavía la técnica de *closures* (construir de forma dinámica una función que usa variables definidas fuera de la función), pero ahora hemos combinado las funciones de comparación y transformación en una sola (la razón de este cambio quedará clara en la siguiente sección). Esto le permitirá conseguir lo mismo que si tuviera funciones, pero la invocación

será diferente como verá en unos momentos.

❷ Ahora nuestra función `plural` toma un segundo parámetro opcional, `language`, que por omisión es `en`.

❸ Usamos el parámetro `language` para construir el nombre de un fichero, abrirlo y leer su contenido dentro de una lista. Si `language` es `en`, entonces abriremos el fichero `rules.en`, lo leeremos entero, lo cortaremos en los retornos de carro y devolveremos una lista. Cada línea del fichero será un elemento de la lista.

❹ Como pudo ver, cada línea del fichero tiene en realidad tres valores, pero están separados por espacios en blanco (tabuladores o espacios, no hay diferencia). Relacionar la función `string.split` sobre esta lista creará una lista nueva en la que cada elemento es una tupla de tres cadenas. De manera que una línea como `[sxz]$ $ es` quedará dividida en la tupla `('[sxz]$', '$', 'es')`. Esto significa que `patterns` contendrá una lista de tuplas, igual que habíamos programado en la fase 4.

❺ Si `patterns` es una lista de tuplas, entonces `rules` será una lista de las funciones creadas de forma dinámica con cada llamada a `buildRule`. Llamar a `buildRule(('[sxz]$', '$', 'es'))` devuelve una función que toma un único parámetros, `word`. Cuando se invoque esta función devuelta, ejecutará `re.search('[sxz]$', word) and re.sub('$', 'es', word)`.

❻ Al estar creando ahora una función de combinación y transformación combinada, necesitamos invocarla de forma diferente. Simplemente llamamos a la función y si devuelve algo, eso es el plural; si no devuelve nada (`None`), entonces la regla no coincidió con nada y necesitamos probar con otra más.

La mejora aquí es que hemos apartado completamente las reglas de pluralización a un fichero externo. No sólo se puede mantener el fichero de forma separada sdel código, sino que hemos creado un esquema de nombres donde la misma función `plural` puede usar diferentes listas de reglas, basándose en el parámetro `language`.

Lo malo aquí es que estamos leyendo el fichero cada vez que invocamos la función `plural`. Pensé que podría pasarme todo este libro sin usar la frase "lo dejo como ejercicio para el lector", pero ahí va: *se deja como ejercicio al lector* construir un mecanismo de caché para los ficheros específicos a un idioma que se autorefresque si el fichero de reglas cambia entre llamadas. Diviértase.

Footnotes

[22] N. del T.: *En*glish

17.7. `plural.py`, fase 6

Ahora está listo para que le hable de generadores.

Ejemplo 17.17. `plural6.py`

```
import re

def rules(language):
    for line in file('rules.%s' % language):
        pattern, search, replace = line.split()
        yield lambda word: re.search(pattern, word) and re.sub(search,
replace, word)

def plural(noun, language='en'):
    for applyRule in rules(language):
        result = applyRule(noun)
        if result: return result
```

Esto usa una técnica llamada generadores que no voy siquiera a intentar explicar hasta que eche un vistazo antes a un ejemplo más simple.

Ejemplo 17.18. Presentación de los generadores

```
>>> def make_counter(x):
...     print 'entering make_counter'
...     while 1:
...         yield x                  ❶
```

```
...         print 'incrementando x'
...         x = x + 1
...
>>> counter = make_counter(2) ❷
>>> counter                   ❸
<generator object at 0x001C9C10>
>>> counter.next()            ❹
entering make_counter
2
>>> counter.next()            ❺
incrementando x
3
>>> counter.next()            ❻
incrementando x
4
```

❶ La presencia de la palabra reservada `yield` en `make_counter` significa que esto no es una función normal. Es un tipo especial de función que genera valores uno por vez. Puede pensar en ella como una función de la que se puede salir a la mitad para volver luego al punto donde se dejó. Invocarla devolverá un generador que se puede usar para crear valores sucesivos de `x`.

❷ Para crear una instancia del generador `make_counter` basta invocarla como cualquier otra función. Advierta que esto no ejecuta realmente el código de la función. Lo sabemos porque la primera línea de `make_counter` es una sentencia `print`, pero aún no se ha mostrado nada.

❸ La función `make_counter` devuelve un objeto generador.

❹ La primera vez que llamamos al método `next()` del generador, ejecuta el código de `make_counter` hasta la sentencia `yield` y luego devuelve el valor cedido[23]. En este caso será `2`, porque originalmente creamos el generador llamando a `make_counter(2)`.

❺ Invocar `next()` repetidamente sobre el objeto generador *lo reanuda donde lo dejó* y continúa hasta que encontramos la siguiente sentencia `yield`. La siguiente línea de código que espera ser ejecutada es la sentencia `print` que imprime `incrementando x`, y tras esto la `x = x + 1` que la incrementa. Entonces entramos de nuevo en el bucle `while` y lo primero que hacemos es

`yield x`, que devuelve el valor actual de `x` (ahora 3).

❻ La segunda vez que llamamos a `counter.next()`, hacemos lo mismo pero esta vez `x` es `4`. Y así en adelante. Dado que `make_counter` crea un bucle infinito podríamos seguir esto para siempre (teóricamente), y se mantendría incrementando `x` y escupiendo valores. Pero en vez de eso, veamos usos más productivos de los generadores.

Ejemplo 17.19. Uso de generadores en lugar de recursividad

```
def fibonacci(max):
    a, b = 0, 1       ❶
    while a < max:
        yield a       ❷
        a, b = b, a+b ❸
```

❶ La sucesión de Fibonacci es una secuencia de números donde cada uno es la suma de los dos anteriores. Empieza en `0` y `1`, y crece lentamente al principio y luego cada vez más rápido. Para empezar la secuencia necesitamos dos variables: `a` empieza en `0`, y `b` empieza en `1`.

❷ `a` es el número actual de la secuencia, así que lo cedemos.

❸ `b` es el siguiente número de la secuencia así que se lo asignamos a `a`, pero también calculamos el valor siguiente (`a+b`) y se lo asignamos a `b` para más adelante. Observe que esto ocurre de forma paralela; si `a` es `3` y `b` es `5`, entonces `a, b = b, a+b` dejará `a` con `5` (el valor previo de `b`) y `b` con `8` (la suma de los dos valores anteriores de `a` y `b`).

Ahora tenemos una función que escupe valores sucesivos de Fibonacci. Bien, podíamos haber hecho eso con recursividad pero de esta manera es más fácil de leer. Además, funciona bien con bucles `for`.

Ejemplo 17.20. Generadores y bucles `for`

```
>>> for n in fibonacci(1000): ❶
...     print n,               ❷
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

❶ Podemos usar un generador como `fibonacci` directamente con un bucle `for`. El bucle `for` creará el objeto generador e invocará sucesivamente el método `next()` para obtener valores que asignar a la variable de bucle del `for` (`n`).

❷ Por cada iteración sobre el bucle `for` `n` obtiene un valor nuevo de la sentencia `yield` de `fibonacci`, y todo lo que hacemos es imprimirlo. Una vez `fibonacci` llega al límite (`a` se hace mayor que `max`, que en este caso es `1000`), entonces el bucle `for` termina sin novedad.

Bien, volvamos a la función `plural` y veamos cómo se usa esto.

Ejemplo 17.21. Generadores que generan funciones dinámicas

```
def rules(language):
    for line in file('rules.%s' % language):
❶
        pattern, search, replace = line.split()
❷
        yield lambda word: re.search(pattern, word) and re.sub(search,
replace, word) ❸

def plural(noun, language='en'):
    for applyRule in rules(language):  ❹
        result = applyRule(noun)
        if result: return result
```

❶ `for line in file(...)` es una construcción común que se usa para leer ficheros línea por línea. Funciona porque *`file` devuelve en realidad un generador* cuyo método `next()` devuelve la siguiente línea del fichero. Esto es inmensamente útil; se me hace la boca agua sólo de pensarlo.

❷ Aquí no hay magia. Recuerde que las líneas del fichero de reglas tienen tres valores separados por espacios en blanco, así que `line.split()` devuelve una tupla de 3 valores, y los asignamos a 3 variables locales.

❸ *Y entonces cedemos.* ¿Qué cedemos? Una función, construida de forma dinámica con `lambda`, que en realidad es un *closure* (usa las variables locales `pattern`, `search` y `replace` como constantes). En otras palabras, `rules` es un

generador que devuelve funciones de reglas.

❹ Dado que `rules` es un generador, podemos usarlo directamente en un bucle `for`. La primera iteración sobre el bucle `for` invocaremos la función `rules`, que abrirá el fichero de reglas, leerá la primera línea, construirá dinámicamente una función que compare y aplique la primera regla definida en el fichero de reglas y cederá la función dinámica. En la segunda iteración sobre el bucle `for` se retoma la secuencia donde `rules` lo dejó (o sea, en mitad del bucle `for line in file(...)`), se lee la segunda línea del fichero de reglas, se construye dinámicamente otra función que compare y transforme según la segunda regla definida en el fichero de reglas, y la cede. Y así hasta el final.

¿En qué ha mejorado frente a la fase 5? En la fase 5 leíamos el fichero de reglas entero y creábamos una lista de todas las posibles antes siquiera de probar la primera. Ahora con los generadores podemos hacerlo todo perezosamente: abrimos el fichero, leemos la primera regla y creamos la función que la va a probar, pero si eso funciona no leemos el resto del fichero ni creamos otras funciones.

Lecturas complementarias

- El PEP 255 define los generadores.
- El Python Cookbook tiene muchos más ejemplos sobre generadores.

Footnotes

[23] *yielded*

17.8. Resumen

En este capítulo hemos hablado sobre varias técnicas avanzadas diferentes. No todas son apropiadas para cada situación.

Ahora debería sentirse cómodo con las siguientes técnicas:

- Realizar sustitución de cadenas mediante expresiones regulares.
- Tratar funciones como objetos, almacenarlas en listas, asignarlas a variables e invocarlas mediante esas variables.
- Construir funciones dinámicas con `lambda`.
- Construir *closures*, funciones dinámicas que encierran como constantes las variables que tiene a su alrededor.
- Construir generadores, funciones que pueden ceder el control que realizan lógica incremental y devuelven valores diferentes cada vez que las invoca.

Añadir abstracciones, construir funciones dinámicamente, crear *closures* y usar generadores pueden hacer su código más simple, más legible y más flexible. Pero también pueden acabar haciendo más complicada la tarea de depuración. Es usted quien ha de encontrar el equilibrio entre simplicidad y potencia.

Capítulo 18. Ajustes de rendimiento

Los ajustes de rendimiento son tarea de muchos esplendores. El hecho de que Python sea un lenguaje interpretado no significa que no deba preocuparse por la optimización de código. Pero tampoco debe preocuparse *demasiado*.

18.1. Inmersión

Hay tantas trampas en el camino a optimizar el código, que es difícil saber dónde empezar.

Empecemos por aquí: *¿está seguro de que lo necesita?* ¿Es tan malo su código? ¿Merece la pena el tiempo de afinarlo? Durante el tiempo de vida de su aplicación, ¿cuánto tiempo va a pasar ese código ejecutándose, comparado al tiempo que habrá de esperar por una base de datos remota o por entrada de un usuario?

Segundo, *¿está seguro de que ya terminó de programar?* Optimizar prematuramente es como adornar un pastel a medio hornear. Pasará horas o días (incluso más) optimizando el código para mejorar el rendimiento, sólo para descubrir que no hace lo que necesitaba que hiciera. Es tiempo que se va a la basura.

Esto no quiere decir que la optimización de código no merezca la pena, pero hace falta mirar al sistema completo y decidir cual es la mejor manera de usar su tiempo. Cada minuto que pierda optimizando el código es tiempo que no

pasa añadiendo nuevas características, escribiendo documentación, jugando con sus hijos, o escribiendo pruebas unitarias.

Oh sí, pruebas unitarias. Casi se me pasa decir que necesitará un juego completo de pruebas unitarias antes de empezar el ajuste de rendimiento. Lo último que necesita es introducir nuevos fallos mientras juega con los algoritmos.

Con estas consideraciones en mente, veamos algunas técnicas para optimizar código en Python. El código en cuestión es una implementación del algoritmo Soundex. Soundex era un método que usaban a principios del siglo 20 para organizar apellidos en categorías en el censo de los Estados Unidos. Agrupaba juntos nombres que sonaban parecido, así que incluso aunque se deletrease mal un nombre, los investigadores tendrían alguna oportunidad de encontrarlo. Soundex se sigue usando hoy día por la misma ra'zon aunque, por supuesto, ahora usamos servidores de bases de datos computerizadas. La mayoría de los servidores de bases de datos incluyen una función Soundex.

Hay muchas variaciones sutiles sobre el algoritmo Soundex. En este capítulo usamos ésta:

1. Deje la primera letra del nombre tal cual.
2. Convierta el resto de letras en dígitos, de acuerdo a una tabla específica:
    - B, F, P, y V pasan a ser 1.
    - C, G, J, K, Q, S, X y Z pasan a ser 2.
    - D y T pasan a ser 3.
    - L pasa a ser 4.
    - M y N pasan a ser5 .
    - R pasa a ser 6.
    - El resto de las letras pasan a ser 9.
3. Elimine los duplicados consecutivos.
4. Elimine todos los 9.
5. Si el resultado es menor de cuatro caracteres (la primera letra más tres dígitos), añada ceros al resultado hasta ese tamaño.

6. si el resultado es mayor de cuatro caracteres, descarte todo tras el cuarto carácter.

Por ejemplo mi nombre, `Pilgrim`, se convierte en P942695. No tiene duplicados consecutivos así que no haré nada con eso. Ahora eliminamos los 9, dejando P4265. Eso es demasiado largo así que descartamos el carácter de exceso, dejando P426.

Otro ejemplo: `Woo` pasa a ser W99, que deriva en W9, que a su vez se convierte en W, que hay que rellenar con ceros dejando W000.

Éste es el primer intento de una función Soundex:

Ejemplo 18.1. `soundex/stage1/soundex1a.py`

Si aún no lo ha hecho, puede descargar éste ejemplo y otros usados en este libro.

```
import string, re

charToSoundex = {"A": "9",
                 "B": "1",
                 "C": "2",
                 "D": "3",
                 "E": "9",
                 "F": "1",
                 "G": "2",
                 "H": "9",
                 "I": "9",
                 "J": "2",
                 "K": "2",
                 "L": "4",
                 "M": "5",
                 "N": "5",
                 "O": "9",
                 "P": "1",
                 "Q": "2",
                 "R": "6",
```

```
                "S": "2",
                "T": "3",
                "U": "9",
                "V": "1",
                "W": "9",
                "X": "2",
                "Y": "9",
                "Z": "2"}

def soundex(source):
    "convert string to Soundex equivalent"

    # Soundex requirements:
    # source string must be at least 1 character
    # and must consist entirely of letters
    allChars = string.uppercase + string.lowercase
    if not re.search('^[%s]+$' % allChars, source):
        return "0000"

    # Soundex algorithm:
    # 1. make first character uppercase
    source = source[0].upper() + source[1:]

    # 2. translate all other characters to Soundex digits
    digits = source[0]
    for s in source[1:]:
        s = s.upper()
        digits += charToSoundex[s]

    # 3. remove consecutive duplicates
    digits2 = digits[0]
    for d in digits[1:]:
        if digits2[-1] != d:
            digits2 += d

    # 4. remove all "9"s
    digits3 = re.sub('9', '', digits2)

    # 5. pad end with "0"s to 4 characters
    while len(digits3) < 4:
        digits3 += "0"
```

```
    # 6. return first 4 characters
    return digits3[:4]

if __name__ == '__main__':
    from timeit import Timer
    names = ('Woo', 'Pilgrim', 'Flingjingwaller')
    for name in names:
        statement = "soundex('%s')" % name
        t = Timer(statement, "from __main__ import soundex")
        print name.ljust(15), soundex(name), min(t.repeat())
```

Lecturas complementarias sobre Soundex

- Soundexing and Genealogy proporciona una cronología de la evolución de Soundex y sus variantes regionales.

18.2. Uso del módulo `timeit`

La cosa más importante que debe saber sobre optimización de código en Python es que no debería escribir su propia función de cronometraje.

Cronometrar partes pequeñas de código es algo increíblemente complicado. ¿Cuánto tiempo dedica el computador a ejecutar este código? ¿Hay cosas ejecutándose en segundo plano? ¿Está seguro? Cada computador moderno tiene procesos en segundo plano, algunos todo el tiempo y otros intermitentemente. Se lanzan tareas de cron a intervalos regulares; se "despiertan" ocasionalmente servicios en segundo plano para hacer cosas útiles como buscar correo nuevo, conectar a servidores de mensajería instantánea, buscar actualizaciones de aplicaciones, buscar virus, comprobar si se ha insertado un disco en la unidad de CD en los últimos 100 nanosegundos, *etc.*. Antes de empezar a cronometrar, apague ese servicio que comprueba incesantemente si ha vuelto la conexión a red, y entonces...

Y entonces está el asunto de las variaciones que introduce el propio marco cronometrado. ¿Consulta el intérprete de Python los nombres de los métodos en una caché? ¿Hace caché de bloques de código compilado? ¿De expresiones

regulares? ¿Tendrá su código efectos secundarios si se ejecuta más de una vez? No olvide que estamos tratando con pequeñas fracciones de un segundo, así que incluso errores minúsculos en el cronometrado fastidiarán irreparablemente los resultados.

La comunidad de Python tiene un dicho: "Python incluye las baterías." No escriba su propia infraestructura para cronometrar. Python 2.3 incluye una perfectamente buena llamada `timeit`.

Ejemplo 18.2. Presentación de `timeit`

Si aún no lo ha hecho, puede descargar éste ejemplo y otros usados en este libro.

```
>>> import timeit
>>> t = timeit.Timer("soundex.soundex('Pilgrim')",
...     "import soundex")    ❶
>>> t.timeit()               ❷
8.21683733547
>>> t.repeat(3, 2000000)     ❸
[16.48319309109, 16.46128984923, 16.44203948912]
```

❶ El módulo `timeit` define una clase `Timer` que toma dos argumentos. Ambos argumentos son cadenas. El primero es una sentencia que deseamos cronometrar; en este caso cronometramos una llamada a la función Soundex dentro de `soundex` con `'Pilgrim'` como argumento. El segundo argumento de la clase `Timer` es la sentencia que importará el entorno de la ejecución. `timeit` crea internamente un entorno virtual aislado, ejecuta manualmente la sentencia de configuración (importa el módulo `soundex`), y luego compila y ejecuta manualmente la sentencia cronometrada (invocando la función Soundex).

❷ Una vez tenemos el objeto `Timer`, lo más sencillo es invocar `timeit()`, que llama a nuestra función 1 millón de veces y devuelve el número de segundos que le llevó hacerlo.

❸ El otro método principal del objeto `Timer` es `repeat()`, que toma dos

argumentos opcionales. El primero es el número de veces que habrá de repetir la prueba completa, y el segundo es el número de veces que ha de llamar a la sentencia cronometrada dentro de cada prueba. Ambos argumentos son opcionales y por omisión serán `3` y `1000000` respectivamente. El método `repeat()` devuelve una lista del tiempo en segundos que llevó terminar cada ciclo de prueba.

Podemos usar el módulo `timeit` desde la línea de órdenes para probar un programa de Python que ya exista, sin modificar el código. Vea http://docs.python.org/lib/node396.html si desea documentación sobre las opciones para línea de órdenes.

Observe que `repeat()` devuelve una lista de tiempos. Los tiempos serán diferentes casi siempre, debido a ligeras variaciones en la cantidad de tiempo de procesador que se le asigna al intérprete de Python (y esos dichosos procesos en segundo plano de los que no se pudo librar). Su primera idea podría ser decir "Hallemos la media, a la que llamaremos El Número Verdadero."

De hecho, eso es incorrecto casi con seguridad. Las pruebas que tardaron más no lo hicieron debido a variaciones en el código o en el intérprete de Python; sino debido a esos fastidiosos procesos en segundo plano, y otros factores ajenos al intérprete de Python que no se pudieron eliminar completamente. Si los diferentes resultados de cronometraje difieren por más de un pequeño porcentaje, aún tenemos demasiada variabilidad para confiar en los resultados. En caso contrario tome el valor más pequeño y descarte los demás.

Python tiene una función `min` bastante a mano que toma una lista y devuelve el valor más pequeño:

```
>>> min(t.repeat(3, 1000000))
8.22203948912
```

El módulo `timeit` sólo sirve de algo si ya sabe qué parte de su código

quiere optimizar. Si tiene un programa grande escrito en Python y no sabe dónde tiene los problemas de rendimiento, pruebe el módulo `hotshot`.

18.3. Optimización de expresiones regulares

Lo primero que comprueba la función Soundex es si la entrada es una cadena de letras que no esté vacía. ¿Cual es la mejor manera de hacerlo?

Si respondió: "expresiones regulares", siéntese en esa esquina y medite sobre sus malos instintos. Las expresiones regulares no son la respuesta correcta casi nunca; se deben evitar siempre que se pueda. No sólo por razones de rendimiento, sino simplemente porque son difíciles de depurar y mantener. Además de por razones de rendimiento.

Este fragmento de código de `soundex/stage1/soundex1a.py` comprueba si el argumento de la función `source` es una palabra hecha enteramente por letras, o por al menos una letra (no una cadena vacía):

```
    allChars = string.uppercase + string.lowercase
    if not re.search('^[%s]+$' % allChars, source):
        return "0000"
```

¿Qué tal se desempeña `soundex1a.py`? Por conveniencia, la sección `__main__` del *script* contiene este código que llama al módulo `timeit`, prepara una prueba de cronometrado con tres nombres diferentes, prueba cada nombre tres veces y muestra el tiempo mínimo de cada una:

```
if __name__ == '__main__':
    from timeit import Timer
    names = ('Woo', 'Pilgrim', 'Flingjingwaller')
    for name in names:
        statement = "soundex('%s')" % name
        t = Timer(statement, "from __main__ import soundex")
        print name.ljust(15), soundex(name), min(t.repeat())
```

Así que, ¿cómo rinde `soundex1a.py` con esta expresión regular?

```
C:\samples\soundex\stage1>python soundex1a.py
Woo             W000 19.3356647283
Pilgrim         P426 24.0772053431
Flingjingwaller F452 35.0463220884
```

Como era de esperar, el algoritmo tarda significativamente más cuando se le llama con nombres más largos. Hay pocas cosas que podamos hacer para hacer más pequeña esa brecha (hacer que la función tome menos tiempo relativo para entradas más largas), pero la naturaleza del algoritmo dicta que nunca se ejecutará en tiempo constante.

La otra cosa a tener en mente es que estamos probando una muestra representativa de ombres. `Woo` es algo así como un caso trivial, ya que se acorta a una sola letra y se rellena con ceros. `Pilgrim` es un caso normal, de longitud media y una mezcla de letras significativas e ignoradas. `Flingjingwaller` es extraordinariamente largo y contiene duplicados consecutivos. Otras pruebas podrían ser de ayuda igualmente, pero éstas ya cubren un buen rango de casos diferentes.

Entonces, ¿qué pasa con esa expresión regular? Bueno, no es eficiente. Dado que la expresión está buscando letras en rangos (`A-Z` en mayúsculas y `a-z` en minúsculas), podemos usar atajos en la sintaxis de la expresión regular. Éste es `soundex/stage1/soundex1b.py`:

```
    if not re.search('^[A-Za-z]+$', source):
        return "0000"
```

`timeit` dice que `soundex1b.py` es ligeramente más rápido que `soundex1a.py`, pero nada con lo que uno pueda excitarse terriblemente:

```
C:\samples\soundex\stage1>python soundex1b.py
Woo             W000 17.1361133887
Pilgrim         P426 21.8201693232
Flingjingwaller F452 32.7262294509
```

Ya vimos en Sección 15.3, “Refactorización” que las expresiones regulares se pueden compilar y ser reutilizadas para obtener resultados más rápidos. Dado

que esta expresión regular no cambia nunca entre llamadas a función, podemos compilarla una vez y usarla así. Éste es `soundex/stage1/soundex1c.py`:

```
isOnlyChars = re.compile('^[A-Za-z]+$').search
def soundex(source):
    if not isOnlyChars(source):
        return "0000"
```

Usar una versión compilada de la expresión regular en `soundex1c.py` es significativamente más rápido:

```
C:\samples\soundex\stage1>python soundex1c.py
Woo             W000 14.5348347346
Pilgrim         P426 19.2784703084
Flingjingwaller F452 30.0893873383
```

Pero, ¿es éste el camino equivocado? La lógica aquí es simple: la entrada `source` no puede estar vacía, y debe estar compuesta enterametne de letras. ¿No sería más rápido escribir un bucle que compruebe cada carácter y eliminar totalmente las expresiones regulares?

Here is `soundex/stage1/soundex1d.py`:

```
    if not source:
        return "0000"
    for c in source:
        if not ('A' <= c <= 'Z') and not ('a' <= c <= 'z'):
            return "0000"
```

Resulta que esta técnica aplicada en `soundex1d.py` *no* es más rápida que la de usar una expresión regular compilada (aunque es más rápida que usar una expresión regular sin compilar):

```
C:\samples\soundex\stage1>python soundex1d.py
Woo             W000 15.4065058548
Pilgrim         P426 22.2753567842
Flingjingwaller F452 37.5845122774
```

¿Por qué no es más rápido `soundex1d.py`? La respuesta está en la naturaleza interpretada de Python. El motor de expresiones regulares está escrito en C, y compilado para que funcione de forma nativa en su computador. Por otro lado, este bucle está escrito en Python y funciona mediante el intérprete. Incluso aunque el bucle es relativamente simple, no es suficientemente simple para compensar por el hecho de ser interpretado. Las expresiones regulares no son nunca la respuesta adecuada... excepto cuando lo son.

Resulta que Python ofrece un método de cadenas algo oscuro. Tiene excusa para no saberlo ya que no lo he mencionado en ninguna parte de este libro. El método se llama `isalpha()`, y comprueba si una cadena contiene sólo letras.

Éste es `soundex/stage1/soundex1e.py`:

```
    if (not source) and (not source.isalpha()):
        return "0000"
```

¿Cuánto hemos ganado usando este método específico en `soundex1e.py`? Bastante.

```
C:\samples\soundex\stage1>python soundex1e.py
Woo             W000 13.5069504644
Pilgrim         P426 18.2199394057
Flingjingwaller F452 28.9975225902
```

Ejemplo 18.3. El mejor resultado por mucho:

`soundex/stage1/soundex1e.py`

```
import string, re

charToSoundex = {"A": "9",
                 "B": "1",
                 "C": "2",
                 "D": "3",
                 "E": "9",
                 "F": "1",
                 "G": "2",
                 "H": "9",
```

```
                "I": "9",
                "J": "2",
                "K": "2",
                "L": "4",
                "M": "5",
                "N": "5",
                "O": "9",
                "P": "1",
                "Q": "2",
                "R": "6",
                "S": "2",
                "T": "3",
                "U": "9",
                "V": "1",
                "W": "9",
                "X": "2",
                "Y": "9",
                "Z": "2"}

def soundex(source):
    if (not source) and (not source.isalpha()):
        return "0000"
    source = source[0].upper() + source[1:]
    digits = source[0]
    for s in source[1:]:
        s = s.upper()
        digits += charToSoundex[s]
    digits2 = digits[0]
    for d in digits[1:]:
        if digits2[-1] != d:
            digits2 += d
    digits3 = re.sub('9', '', digits2)
    while len(digits3) < 4:
        digits3 += "0"
    return digits3[:4]

if __name__ == '__main__':
    from timeit import Timer
    names = ('Woo', 'Pilgrim', 'Flingjingwaller')
    for name in names:
        statement = "soundex('%s')" % name
        t = Timer(statement, "from __main__ import soundex")
```

```
print name.ljust(15), soundex(name), min(t.repeat())
```

18.4. Optimización de búsquedas en diccionarios

El segundo paso del algoritmo Soundex es convertir los caracteres en dígitos según un patrón específico. ¿Cual es la mejor manera de hacer esto?

La solución más obvia es definir un diccionario con caracteres individuales como claves y sus dígitos correspondientes como valores, y hacer consultas al diccionario por cada carácter. Esto es lo que tenemos en `soundex/stage1/soundex1c.py` (uno de los mejores resultados hasta ahora):

```
charToSoundex = {"A": "9",
                 "B": "1",
                 "C": "2",
                 "D": "3",
                 "E": "9",
                 "F": "1",
                 "G": "2",
                 "H": "9",
                 "I": "9",
                 "J": "2",
                 "K": "2",
                 "L": "4",
                 "M": "5",
                 "N": "5",
                 "O": "9",
                 "P": "1",
                 "Q": "2",
                 "R": "6",
                 "S": "2",
                 "T": "3",
                 "U": "9",
                 "V": "1",
                 "W": "9",
                 "X": "2",
                 "Y": "9",
                 "Z": "2"}
```

```
def soundex(source):
    # ... input check omitted for brevity ...
    source = source[0].upper() + source[1:]
    digits = source[0]
    for s in source[1:]:
        s = s.upper()
        digits += charToSoundex[s]
```

Ya hemos cronometrado `soundex1c.py`; éste es su rendimiento:

```
C:\samples\soundex\stage1>python soundex1c.py
Woo             W000 14.5341678901
Pilgrim         P426 19.2650071448
Flingjingwaller F452 30.1003563302
```

Este código es muy simple pero, ¿es la mejor solución? Invocar `upper()` sobre cada carácter individualmente no parece eficiente; probablemente sería mejor llamar a `upper()` una sola vez sobre la cadena entera.

Luego está el asunto de construir la cadena `digits` de forma incremental. La construcción incremental de cadenas es horriblemente ineficiente; de forma interna, el intérprete de Python necesita crear una cadena nueva cada vez dentro del bucle, y descartar la anterior.

Python es bueno con las listas, sin embargo. Podemos tratar automáticamente una cadena como una lista de caracteres. Y es fácil combinar de nuevo una lista en una cadena, usando el método de cadena `join()`.

Éste es `soundex/stage2/soundex2a.py`, que convierte las letras en dígitos usando ↦ y `lambda`:

```
def soundex(source):
    # ...
    source = source.upper()
    digits = source[0] + "".join(map(lambda c: charToSoundex[c],
source[1:]))
```

Sorprendentemente, `soundex2a.py` no es más rápida:

```
C:\samples\soundex\stage2>python soundex2a.py
Woo             W000 15.0097526362
Pilgrim         P426 19.254806407
Flingjingwaller F452 29.3790847719
```

La sobrecarga de la función anónima `lambda` destruye cualquier rendimiento que ganemos al tratar la cadena como una lista de caracteres.

`soundex/stage2/soundex2b.py` usa una lista por comprensión en lugar de ↦ y `lambda`:

```
    source = source.upper()
    digits = source[0] + "".join([charToSoundex[c] for c in
source[1:]])
```

Al usar una lista de comprensión `soundex2b.py` es más rápida que `soundex2a.py` usando ↦ y `lambda`, pero sigue sin ser más rápida que el código original (construir una cadena de forma incremental en `soundex1c.py`):

```
C:\samples\soundex\stage2>python soundex2b.py
Woo             W000 13.4221324219
Pilgrim         P426 16.4901234654
Flingjingwaller F452 25.8186157738
```

Es hora de dar un enfoque radicalmente diferente. Las búsquedas en diccionarios son herramientas de propósito general. Las claves de los diccionarios pueden ser cadenas de cualquier longitud (o muchos otros tpos de datos), pero en este caso estamos tratando con claves de un único carácter *y* valores de un único carácter. Resulta que Python tiene una función especializada exactamente para tratar esta situación: la función `string.maketrans`.

This is `soundex/stage2/soundex2c.py`:

```
allChar = string.uppercase + string.lowercase
```

```
charToSoundex = string.maketrans(allChar, "91239129922455912623919292"
* 2)
def soundex(source):
    # ...
    digits = source[0].upper() + source[1:].translate(charToSoundex)
```

¿Qué demonios está pasando aquí? `string.maketrans` crea una matriz de traducción entre dos cadenas: el primer argumento y el segundo. En este caso, el primer argumento es la cadena `ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz`, y el segundo la cadena `91239129922455912623919292912391299224559126239192`. ¿Ve el patrón? Es el mismo patrón de conversión que estamos haciendo caligráficamente con un diccionario. A se corresponde con 9, B con 1, C con 2, *etc.*. Pero no es un diccionario; es una estructura de datos especializada a la que podemos acceder usando el método de cadenas `translate`, que traduce cada carácter en el dígito correspondiente, de acuerdo a la matriz definida por `string.maketrans`.

`timeit` muestra que `soundex2c.py` es significativamente más rápida que si definiéramos un diccionario e iterásemos sobre la cadena para construir la salida de modo incremental:

```
C:\samples\soundex\stage2>python soundex2c.py
Woo             W000 11.437645008
Pilgrim         P426 13.2825062962
Flingjingwaller F452 18.5570110168
```

No va a conseguir nada mucho mejor que esto. Python tiene una función especializada que hace exactamente lo que quiere; úsela y no se rompa la cabeza.

Ejemplo 18.4. El mejor resultado hasta ahora:

`soundex/stage2/soundex2c.py`

```
import string, re

allChar = string.uppercase + string.lowercase
```

```
charToSoundex = string.maketrans(allChar, "91239129922455912623919292"
* 2)
isOnlyChars = re.compile('^[A-Za-z]+$').search

def soundex(source):
    if not isOnlyChars(source):
        return "0000"
    digits = source[0].upper() + source[1:].translate(charToSoundex)
    digits2 = digits[0]
    for d in digits[1:]:
        if digits2[-1] != d:
            digits2 += d
    digits3 = re.sub('9', '', digits2)
    while len(digits3) < 4:
        digits3 += "0"
    return digits3[:4]

if __name__ == '__main__':
    from timeit import Timer
    names = ('Woo', 'Pilgrim', 'Flingjingwaller')
    for name in names:
        statement = "soundex('%s')" % name
        t = Timer(statement, "from __main__ import soundex")
        print name.ljust(15), soundex(name), min(t.repeat())
```

18.5. Optimización de operaciones con listas

El tercer paso en el algoritmo Soundex es eliminar dígitos consecutivos duplicados. ¿Cual es la mejor manera de hacerlo?

Éste es el código que hemos usado hasta ahora, en `soundex/stage2/soundex2c.py`:

```
    digits2 = digits[0]
    for d in digits[1:]:
        if digits2[-1] != d:
            digits2 += d
```

Éstos son los resultados de rendimiento de `soundex2c.py`:

```
C:\samples\soundex\stage2>python soundex2c.py
```

```
Woo             W000 12.6070768771
Pilgrim         P426 14.4033353401
Flingjingwaller F452 19.7774882003
```

Lo primero que debemos considerar es si es eficiente comprobar `digits[-1]` cada vez dentro del bucle. ¿Son costosos los índices de las listas? ¿Sería mejor mantener el último dígito en una variable aparte y mirar eso en su lugar?

La respuesta a esa pregunta está en `soundex/stage3/soundex3a.py`:

```
    digits2 = ''
    last_digit = ''
    for d in digits:
        if d != last_digit:
            digits2 += d
            last_digit = d
```

`soundex3a.py` no va nada más rápido que `soundex2c.py`, e incluso puede que vaya algo más lento (aunque no lo suficiente como para asegurarlo):

```
C:\samples\soundex\stage3>python soundex3a.py
Woo             W000 11.5346048171
Pilgrim         P426 13.3950636184
Flingjingwaller F452 18.6108927252
```

¿Por qué no es más rápido `soundex3a.py`? Resulta que los índices de las listas en Python son extremadamente eficientes. Acceder varias veces a `digits2[-1]` no es problema. Por otro lado, mantener el último dígito visto en una variable aparte significa que tenemos *dos* asignaciones de variables por cada dígito que almacenamos, lo que elimina cualquier pequeña ganancia que hubiéramos obtenido por eliminar la búsqueda en la lista.

Probemos algo totalmente diferente. Si es posible tratar una cadena como una lista de caracteres, debería ser posible usar una lista por comprensión para iterar sobre la lista. El problema es que el código necesita acceder al carácter previo en la lista, y no es fácil hacerlo con una lista por comprensión sencilla.

Sin embargo, es posible crear una lista de números índice usando la función `range()`, y usar esos números de índice para buscar progresivamente en la lista y extraer cada carácter que sea diferente del anterior. Esto nos dará una lista de caracteres y podemos usar el método de cadena `join()` para reconstruir una cadena partiendo de ella.

Éste es `soundex/stage3/soundex3b.py`:

```
    digits2 = "".join([digits[i] for i in range(len(digits))
                       if i == 0 or digits[i-1] != digits[i]])
```

¿Es más rápido? En una palabra, no.

```
C:\samples\soundex\stage3>python soundex3b.py
Woo             W000 14.2245271396
Pilgrim         P426 17.8337165757
Flingjingwaller F452 25.9954005327
```

Es posible que las técnicas hasta ahora hayan sido "centradas en la caden". Python puede convertir una cadena en una lista de caracteres con una sola orden: `list('abc')` devuelve `['a', 'b', 'c']`. Más aún, las listas se pueden *modificar* muy rápidamente. En lugar de crear incrementalmente una nueva lista (o cadena) partiendo de la original, ¿por qué no trabajar con elementos dentro de una única lista?

Éste es `soundex/stage3/soundex3c.py`, que modifica una lista para eliminar elementos consecutivos duplicados:

```
    digits = list(source[0].upper() +
source[1:].translate(charToSoundex))
    i=0
    for item in digits:
        if item==digits[i]: continue
        i+=1
        digits[i]=item
    del digits[i+1:]
    digits2 = "".join(digits)
```

¿Es más rápido que `soundex3a.py` o `soundex3b.py`? No, de hecho es el método más lento:

```
C:\samples\soundex\stage3>python soundex3c.py
Woo             W000 14.1662554878
Pilgrim         P426 16.0397885765
Flingjingwaller F452 22.1789341942
```

No hemos progresado nada aquí, excepto que hemos probado y eliminado varias técnicas "inteligentes". El código más rápido que hemos visto hasta ahora fue el método original y más directo (`soundex2c.py`). Algunas veces ser inteligente no es una ventaja.

Ejemplo 18.5. El mejor resultado hasta ahora:

`soundex/stage2/soundex2c.py`

```
import string, re

allChar = string.uppercase + string.lowercase
charToSoundex = string.maketrans(allChar, "9123912992245591262391929 2"
* 2)
isOnlyChars = re.compile('^[A-Za-z]+$').search

def soundex(source):
    if not isOnlyChars(source):
        return "0000"
    digits = source[0].upper() + source[1:].translate(charToSoundex)
    digits2 = digits[0]
    for d in digits[1:]:
        if digits2[-1] != d:
            digits2 += d
    digits3 = re.sub('9', '', digits2)
    while len(digits3) < 4:
        digits3 += "0"
    return digits3[:4]

if __name__ == '__main__':
    from timeit import Timer
    names = ('Woo', 'Pilgrim', 'Flingjingwaller')
    for name in names:
```

```
        statement = "soundex('%s')" % name
        t = Timer(statement, "from __main__ import soundex")
        print name.ljust(15), soundex(name), min(t.repeat())
```

18.6. Optimización de manipulación de cadenas

El último paso del algoritmo Soundex es rellenar los resultados cortos con ceros y truncar los largos. ¿Cual es la mejor manera de hacerlo?

Esto es lo que tenemos hasta ahora, tomado de `soundex/stage2/soundex2c.py`:

```
    digits3 = re.sub('9', '', digits2)
    while len(digits3) < 4:
        digits3 += "0"
    return digits3[:4]
```

Éstos son los resultados de `soundex2c.py`:

```
C:\samples\soundex\stage2>python soundex2c.py
Woo             W000 12.6070768771
Pilgrim         P426 14.4033353401
Flingjingwaller F452 19.7774882003
```

La primera cosa que hemos de considerar es sustituir esa expresión regular con un bucle. Este código es de `soundex/stage4/soundex4a.py`:

```
    digits3 = ''
    for d in digits2:
        if d != '9':
            digits3 += d
```

¿Es más rápido `soundex4a.py`? Pues sí:

```
C:\samples\soundex\stage4>python soundex4a.py
Woo             W000 6.62865531792
Pilgrim         P426 9.02247576158
Flingjingwaller F452 13.6328416042
```

Pero espere un momento. ¿Un bucle para eliminar caracteres de una cadena? Podemos usar un simple método de cadenas para eso. Aquí está `soundex/stage4/soundex4b.py`:

```
    digits3 = digits2.replace('9', '')
```

¿Es más rápido `soundex4b.py`? Ésa es una pregunta interesante. Depende de la entrada:

```
C:\samples\soundex\stage4>python soundex4b.py
Woo             W000 6.75477414029
Pilgrim         P426 7.56652144337
Flingjingwaller F452 10.8727729362
```

El método de cadena de `soundex4b.py` es más rápido que el bucle en la mayoría de los casos, pero es ligeramente más lento que `soundex4a.py` en el caso trivial (de un nombre muy corto). Las optimizaciones de rendimiento no son siempre uniformes; el afinamiento que hace un caso más rápido a veces puede hacer más lentos otros casos. En este caso, la mayoría se beneficia del cambio, así que lo dejaremos como está, pero es un principio importante de recordar.

Por último pero no por ello menos, examinemos los dos pasos finales del algoritmo: rellenar los resultados cortos con ceros y truncar los resultados largos a cuatro caracteres. El código que vimos en `soundex4b.py` hace eso mismo, pero es horriblemente ineficiente. Eche un vistazo a `soundex/stage4/soundex4c.py` para ver por qué:

```
    digits3 += '000'
    return digits3[:4]
```

¿Par qué necesitamos un bucle `while` sólo para rellenar el resultado? Sabemos con antelación que vamos a truncar el resultado a cuatro caracteres, y ya sabemos que tenemos al menos un carácter (la letra inicial, que llega sin cambiar de la variable `source` original). Eso significa que podemos añadir simplemente tres ceros a la salida y luego truncarla. No se ciña totalmente al

enunciado exacto del problema; verlo desde un ángulo ligeramente diferente puede llevar a una solución más simple.

¿Cuánta velocidad ganamos en `soundex4c.py` eliminando el bucle `while`? Es significativa:

```
C:\samples\soundex\stage4>python soundex4c.py
Woo             W000 4.89129791636
Pilgrim         P426 7.30642134685
Flingjingwaller F452 10.689832367
```

Por último, aún hay una cosa que podemos hacer con esas tres líneas de código para acelerarlas: podemos combinarlas en una sola. Eche un vistazo a `soundex/stage4/soundex4d.py`:

```
    return (digits2.replace('9', '') + '000')[:4]
```

Poner todo este código en una sola línea en `soundex4d.py` es ligeramente más rápido que `soundex4c.py`:

```
C:\samples\soundex\stage4>python soundex4d.py
Woo             W000 4.93624105857
Pilgrim         P426 7.19747593619
Flingjingwaller F452 10.5490700634
```

También es bastante menos legible y no gana mucha velocidad. ¿Merece la pena? Espero que haya puesto buenos comentarios. El rendimiento no lo es todo. Sus esfuerzos de optimización siempre deben estar equilibrados frente a las amenazas a la legibilidad y mantenibilidad de los programas.

18.7. Resumen

Este capítulo ha ilustrado varios aspectos importantes del ajuste de rendimiento en Python y en general.

- Si necesita escoger entre expresiones regulares y escribir un bucle, escoja lo primero. El motor de expresiones regulares está compilado en C y se

ejecuta de forma nativa; el bucle está escrito en Python y se ejecuta en el intérprete.

- Si necesita escoger entre expresiones regulares y métodos de cadenas, escoja lo segundo. Ambos están compilados en C así que elija lo más sencillo.
- Las consultas a diccionarios de propósito general son rápidas, pero las funciones especializadas como `string.maketrans` y los métodos de cadena como `isalpha()` son más rápidos. Si Python tiene una función hecha a su medida, úsela.
- No intente ser demasiado listo. A veces el algoritmo más obvio es también el más rápido.
- No se esfuerce demasiado. El rendimiento no lo es todo.

No puedo subrayar este último punto lo suficiente. A lo largo de este capítulo hemos hecho esta función tres veces más rápida y ganado unos 20 segundos sobre 1 millón de llamadas a función. Maravilloso. Ahora piense, durante ese millón de llamadas, ¿cuántos segundos va a estar esperando la aplicación esperando por una conexión a base de datos? ¿O esperando por E/S de disco? ¿O entrada de datos de usuario? No pierda demasiado tiempo sobreoptimizando un algoritmo, o ignorará mejoras obvias en algún otro lugar. Desarrolle un instinto para el tipo de código que Python ejecuta bien, corrija errores de bulto si los encuentra, y deje el resto sin tocar.

Apéndice A. Lecturas complementarias

Capítulo 1. Instalación de Python

Capítulo 2. Su primer programa en Python

- 2.3. Documentación de funciones
  - PEP 257 define las conveciones al respecto de las `cadenas de documentación`.
  - La *Guía de estilo de Python* indica la manera de escribir una buena `cadena de documentación`.
  - El *Tutorial de Python* expone convenciones para el espaciado dentro de las `cadenas de documentación`.
- 2.4.2. ¿Qué es un objeto?
  - La *Referencia del lenguaje Python* explica exactamente lo que quiere decir que todo en Python es un objeto, porque algunas personas son pedantes y les gusta discutir este tipo de cosas hasta la muerte.
  - eff-bot hace un resumen sobre los objetos en Python.
- 2.5. Sangrado (indentado) de código
  - La *Referencia del lenguaje Python* comenta problemas de sangrado entre plataformas y muestra varios errores de identación.
  - La *Guía de estilo de Python* comenta buenos estilos de sangrado.
- 2.6. Prueba de módulos
  - La *Referencia del lenguaje Python* comenta los detalles de bajo nivel de la importación de módulos.

Capítulo 3. Tipos de dato nativos

- 3.1.3. Borrar elementos de diccionarios
  - *How to Think Like a Computer Scientist* le instruye sobre los diccionarios y le muestra cómo usarlos para modelar matrices dispersas.

    - La Python Knowledge Base tiene muchos ejemplos de código que usan diccionarios.
    - El Python Cookbook comenta cómo ordenar los valores de un diccionario por clave.
    - La *Referencia de bibliotecas de Python* lista todos los métodos de los diccionarios.
- 3.2.5. Uso de operadores de lista
    - *How to Think Like a Computer Scientist* le instruye sobre listas y señala algo importante al respecto de pasar listas como argumentos a funciones.
    - El *Tutorial de Python* muestra cómo usar listas como pilas y colas.
    - La Python Knowledge Base contesta preguntas frecuentes sobre listas y tiene un montón de código de ejemplo que usa listas.
    - La *Referencia de bibliotecas de Python* enumera todos los métodos de las listas.
- 3.3. Presentación de las tuplas
    - *How to Think Like a Computer Scientist* instruye sobre las tuplas y muestra cómo concatenarlas.
    - La Python Knowledge Base muestra cómo ordenar una tupla.
    - El *Tutorial de Python* muestra cómo definir una tupla con un elemento.
- 3.4.2. Asignar varios valores a la vez
    - La *Referencia del lenguaje Python* muestra ejemplos de cuándo puede pasar sin el carácter de continuación de línea y cuándo necesita usarlo.
    - *How to Think Like a Computer Scientist* le muestra cómo usar la asignación multivariable para intercambiar los valores de dos variables.
- 3.5. Formato de cadenas
    - La *Referencia de bibliotecas de Python* enumera todos los caracteres de formato de cadenas.

    - El *Effective AWK Programming* comenta todos los caracteres de formato y técnicas avanzadas de formato de cadenas como especificar ancho, precisión y rellenado con ceros.
- 3.6. Inyección de listas (mapping)
    - El *Tutorial de Python* comenta otra manera de inyectar listas usando la función incorporada `map` function.
    - El *Tutorial de Python* muestra cómo hacer listas por comprensión anidadas.
- 3.7. Unir listas y dividir cadenas
    - La Python Knowledge Base responde a las preguntas comunes sobre cadenas y tiene mucho código de ejemplo que usa cadenas.
    - La *Referencia de bibliotecas de Python* enumera todos los métodos de las cadenas.
    - La *Referencia de bibliotecas de Python* documenta el módulo `string`.
    - La *The Whole Python FAQ* explica por qué `join` es un método de cadena en lugar de ser un método de lista.

Capítulo 4. El poder de la introspección

- 4.2. Argumentos opcionales y con nombre
    - El *Tutorial de Python* explica exactamente cuándo y cómo se evalúan los argumentos por omisión, lo cual es interesante cuando el valor por omisión es una lista o una expresión con efectos colaterales.
- 4.3.3. Funciones incorporadas
    - La *Referencia de bibliotecas de Python* documenta todas las funciones incorporadas y todas las excepciones incorporadas.
- 4.5. Filtrado de listas
    - El *Tutorial de Python* expone otro modo de filtrar listas utilizando la función incorporada `filter`.
- 4.6.1. Uso del truco the and-or
    - Python Cookbook expone alternativas al truco `and-or`.
- 4.7.1. Funciones lambda en el mundo real

    - La Python Knowledge Base explica el uso de funciones `lambda` para llamar funciones indirectamente..
    - El *Tutorial de Python* muestra cómo \ acceder a variables externas desde dentro de una función `lambda`. (PEP 227 expone cómo puede cambiar esto en futuras versiones de Python.)
    - *The Whole Python FAQ* tiene ejemplos de códigos confusos de una línea que utilizan funciones `lambda`.

Capítulo 5. Objetos y orientación a objetos

- 5.2. Importar módulos usando from módulo import
    - eff-bot tiene más cosas que decir sobre `import módulo` frente a `from módulo import`.
    - El *Tutorial de Python* comenta técnicas avanzadas de importación, incluyendo `from módulo import *`.
- 5.3.2. Saber cuándo usar self e __init__
    - *Learning to Program* contiene una introducción a las clases más suave.
    - *How to Think Like a Computer Scientist* muestra cómo usar clases pra modelar tipos de datos compuestos.
    - El *Tutorial de Python* da un vistazo en profundidad a clases, espacios de nombres y herencia.
    - La Python Knowledge Base responde preguntas comunes sobre clases.
- 5.4.1. Recolección de basura
    - La *Referencia de bibliotecas de Python* habla sobre atributos incorporados como `__class__`.
    - La *Referencia de bibliotecas de Python* documenta el módulo `>gc`, que le da control a bajo nivel sobre la recolección de basura de Python.
- 5.5. Exploración de UserDict: Una clase cápsula
    - La*Referencia de bibliotecas de Python* documenta el módulo `UserDict` y el módulo `copy`.
- 5.7. Métodos especiales avanzados

    - La *Referencia del lenguaje Python* documenta todos los métodos especiales de clase.
- 5.9. Funciones privadas
    - El *Tutorial de Python* expone los detalles de las variables privadas.

Capítulo 6. Excepciones y gestión de ficheros

- 6.1.1. Uso de excepciones para otros propósitos
    - El *Tutorial de Python* expone la definición y lanzamiento de sus propias excepciones, y la gestión de varias al mismo tiempo.
    - La *Referencia de bibliotecas de Python* enumera todas las excepciones incorporadas.
    - La *Referencia de bibliotecas de Python* documenta el módulo getpass.
    - La *Referencia de bibliotecas de Python* documenta el módulo `traceback`, que proporciona acceso a bajo nivel a los atributos de las excepciones tras haberse lanzado una.
    - La *Referencia del lenguaje Python* discute los entresijos del bloque `try...except`.
- 6.2.4. Escribir en ficheros
    - El *Tutorial de Python* comenta la lectura y escritura de ficheros, incluyendo la manera de leer un fichero una línea por vez guardándolo en una lista.
    - eff-bot comenta la eficiencia y rendimiento de varias maneras de leer un fichero.
    - La Python Knowledge Base responde preguntas frecuentes sobre ficheros.
    - La *Referencia de bibliotecas de Python* enumera todosl os métodos del objeto de fichero.
- 6.4. Uso de sys.modules
    - El *Tutorial de Python* comenta exactamente cuándo y cómo se evalúan los argumentos por omisión.
    - La *Referencia de bibliotecas de Python* documenta el módulo `sys`.
- 6.5. Trabajo con directorios

    - La Python Knowledge Base contesta preguntas sobre el módulo `os`.
    - La *Referencia de bibliotecas de Python* documenta los módulos `os` y `os.path`.

Capítulo 7. Expresiones regulares

- 7.6. Caso de estudio: análisis de números de teléfono
    - El Regular Expression HOWTO le instruye sobre expresiones regulares y su uso en Python.
    - La *Referencia de bibliotecas de Python* expone el módulo `re`.

Capítulo 8. Procesamiento de HTML

- 8.4. Presentación de BaseHTMLProcessor.py
    - W3C expone las referencias a caracteres y entidades.
    - La *Referencia de bibliotecas de Python* confirmará sus sospechas al respecto de que el módulo `htmlentitydefs` es exactamente lo que parece.
- 8.9. Todo junto
    - A lo mejor pensaba que estaba de broma con la idea del *server-side scripting*. También lo creía yo hasta que encontré esta web dialectizadora. Por desgracia, no parece estar disponible el código fuente.

Capítulo 9. Procesamiento de XML

- 9.4. Unicode
    - Unicode.org es la página del estándar unicode, e incluye una breve introducción técnica.
    - El Unicode Tutorial tiene muchos más ejemplos sobre el uso de funciones unicode de Python, incluyendo la manera de forzar a Python a convertir unicode en ASCII incluso cuando él probablemente no querría.

    - El PEP 263 entra en detalles sobre cómo y cuándo definir una codificación de caracteres en sus ficheros `.py`.

Capítulo 10. Scripts y flujos

Capítulo 11. Servicios Web HTTP

- 11.1. Inmersión
    - Paul Prescod cree que los servichos web HTTP puros son el futuro de Internet.

Capítulo 12. Servicios web SOAP

- 12.1. Inmersión
    - http://www.xmethods.net/ es un repositorio de servicios web SOAP de acceso público.
    - La especificación de SOAP es sorprendentemente legible, si es que le gustan ese tipo de cosas.
- 12.8. Solución de problemas en servicios web SOAP
    - New developments for SOAPpy analiza algunos intentos de conectar a otro servicio SOAP que no funciona exactamente tal como se esperaba.

Capítulo 13. Pruebas unitarias (Unit Testing)

- 13.1. Introducción a los números romanos
    - Este sitio cuenta más cosas sobre los números romanos, incluyendo una fascinante historia sobre la manera en que los romanos y otras civilizaciones los usaban (versión corta: descuidada e inconsistentemente).
- 13.3. Presentación de romantest.py
    - La página de PyUnit tiene una discusión a fondo del uso de la infraestructura de `unittest`, incluyendo características avanzadas que no cubrimos en este capítulo.

    - Las FAQ de PyUnit explican por qué los casos de prueba se almacenan aparte del código sobre el que prueban.
    - La *Referencia de bibliotecas de Python* expone el módulo `unittest`.
    - ExtremeProgramming.org argumenta las razones por las que debe escribir pruebas unitarias.
    - El Portland Pattern Repository tiene una discusión en marcha sobre pruebas unitarias, que incluyen una definicióne stándar, razones por las que debería escribir primero las pruebas unitarias, y varios casos de estudio en profundidad.

Capítulo 14. Programación Test-First

Capítulo 15. Refactorización

- 15.5. Resumen
    - XProgramming.com tiene enlaces para descargar infraestructuras de prueba unitaria para muchos lenguajes distintos.

Capítulo 16. Programación Funcional

Capítulo 17. Funciones dinámicas

- 17.7. plural.py, fase 6
    - El PEP 255 define los generadores.
    - El Python Cookbook tiene muchos más ejemplos sobre generadores.

Capítulo 18. Ajustes de rendimiento

- 18.1. Inmersión
    - Soundexing and Genealogy proporciona una cronología de la evolución de Soundex y sus variantes regionales.

Apéndice B. Repaso en 5 minutos

Capítulo 1. Instalación de Python

- 1.1. ¿Qué Python es adecuado para usted?

  La primera cosa que debe hacer con Python es instalarlo. ¿O no?

- 1.2. Python en Windows

  En Windows debe hacer un par de elecciones antes de instalar Python.

- 1.3. Python en Mac OS X

  En Mac OS X cuenta con dos opciones para instalar Python: instalarlo o no instalarlo. Probablemente quiera instalarlo.

- 1.4. Python en Mac OS 9

  Mac OS 9 no incluye una versión de Python pero instalarla es muy sencillo, y sólo hay una opción.

- 1.5. Python en RedHat Linux

  Descarge el último RPM de Python yendo a http://www.python.org/ftp/python/ y escogiendo el número de versión más alto en la lista, y dentro de ahí, el directorio `rpms/`. Entonces descargue el RPM con el número de versión más alto. Puede instalarlo con la orden **rpm**, como se muestra aquí:

- 1.6. Python en Debian GNU/Linux

- 3.3. Presentación de las tuplas

  Una tupla es una lista inmutable. Una tupla no puede cambiar de ninguna manera una vez creada.

- 3.4. Declaración de variables

  Python tiene variables locales y globales como casi todo el resto de lenguajes, pero no tiene declaración explícitade variables. Las variables cobran existencia al asignársele un valor, y se destruyen automáticamente al salir de su ámbito.

- 3.5. Formato de cadenas

  Python admite dar formato a valores dentro de cadenas. Aunque esto puede incluir expresiones muy complicadas, el uso más básico es insertar valores dentro de una cadena con el sustituto `%s`.

- 3.6. Inyección de listas (mapping)

  Una de las carácterísticas más potentes de Python es la lista por comprensión (*list comprehension*), que proporciona una forma compacta de inyectar una lista en otra aplicando una función a cada uno de sus elementos.

- 3.7. Unir listas y dividir cadenas

  Tenemos una lista de pares clave-valor de forma `clave=valor`, y queremos juntarlos en una sola cadena. Para juntar una lita de cadenas en una sola, usaremos el método `join` de un objeto de cadena.

- 3.8. Resumen

Ya sabe usted que las funciones de Python son objetos. Lo que no sabe es que se puede obtener una referencia a una función sin necesidad de saber su nombre hasta el momento de la ejecución, utilizando la función `getattr`.

- 4.5. Filtrado de listas

  Como ya sabe, Python tiene potentes capacidades para convertir una lista en otra por medio de las listas por comprensión (Sección 3.6, "Inyección de listas (mapping)"). Esto puede combinarse con un mecanismo de filtrado en el que se van a tomar algunos elementos de la lista mientras otros se pasarán por alto.

- 4.6. La peculiar naturaleza de and y or

  En Python, `and` y `or` realizan las operaciones de lógica booleana como cabe esperar, pero no devuelven valores booleanos; devuelven uno de los valores que están comparando.

- 4.7. Utilización de las funciones lambda

  Python admite una interesante sintaxis que permite definir funciones mínimas, de una línea, sobre la marcha. Tomada de Lisp, se trata de las denominadas funciones `lambda`, que pueden utilizarse en cualquier lugar donde se necesite una función.

- 4.8. Todo junto

  La última línea de código, la única que no hemos desmenuzado todavía, es la que hace todo el trabajo. Pero el trabajo ya es fácil, porque todo lo que necesitamos está dispuesto de la manera en que lo necesitamos. Las fichas de

dominó están en su sitio; lo que queda es golpear la primera.

- 4.9. Resumen

  El programa `apihelper.py` y su salida deberían entenderse ya perfectamente.

Capítulo 5. Objetos y orientación a objetos

- 5.1. Inmersión

  Aquí tiene un programa en Python completo y funcional. Lea las `cadenas de documentación` del módulo, las clases, y las funciones para obtener una idea general de lo que hace el programa y cómo funciona. Como de costumbre, no se preocupe por lo que no entienda; para eso está el resto del capítulo.

- 5.2. Importar módulos usando from módulo import

  Python tiene dos maneras de importar módulos. Ambas son útiles, y debe saber cuándo usar cada cual. Una de las maneras, `import` *`módulo`*, ya la hemos visto en Sección 2.4, "Todo es un objeto". La otra hace lo mismo, pero tiene diferencias sutiles e importantes.

- 5.3. Definición de clases

  Python está completamente orientado a objetos: puede definir sus propias clases, heredar de las que usted defina o de las incorporadas en el lenguaje, e instanciar las clases que haya definido.

- 5.4. Instanciación de clases

La instanciación de clases en Python es trivial. Para instanciar una clase, simplemente invoque a la clase como si fuera una función, pasando los argumentos que defina el método `__init__`. El valor de retorno será el objeto recién creado.

- 5.5. Exploración de UserDict: Una clase cápsula

  Como ha podido ver, `FileInfo` es una clase que actúa como un diccionario. Para explorar esto un poco más, veamos la clase `UserDict` del módulo `UserDict`, que es el ancestro de la clase `FileInfo` . No es nada especial; la clase está escrita en Python y almacenada en un fichero `.py`, igual que cualquier otro código Python. En particular, está almacenada en el directorio `lib` de su instalación de Python.

- 5.6. Métodos de clase especiales

  Además de los métodos normales, existe un cierto número de métodos especiales que pueden definir las clases de Python. En lugar de llamarlos directamente desde su código (como los métodos normales), los especiales los invoca Python por usted en circunstancias particulares o cuando se use una sintaxis específica.

- 5.7. Métodos especiales avanzados

  Python tiene más métodos especiales aparte de `__getitem__` y `__setitem__`. Algunos de ellos le permiten emular funcionalidad que puede que aún ni conozca.

- 5.8. Presentación de los atributos de clase

especiales, o si las tiene que combinar con `split` y `join` y listas por comprensión de formas oscuras e ilegibles, puede que deba pasarse a las expresiones regulares.

- 7.2. Caso de estudio: direcciones de calles

  Esta serie de ejemplos la inspiró un problema de la vida real que surgió en mi trabajo diario hace unos años, cuando necesité limpiar y estandarizar direcciones de calles exportadas de un sistema antiguo antes de importarlo a un nuevo sistema (vea que no me invento todo esto; es realmente útil). Este ejemplo muestra la manera en que me enfrenté al problema.

- 7.3. Caso de estudio: números romanos

  Seguramente ha visto números romanos, incluso si no los conoce. Puede que los haya visto en copyrights de viejas películas y programas de televisión ("Copyright `MCMXLVI`" en lugar de "Copyright `1946`"), o en las placas conmemorativas en bibliotecas o universidades ("inaugurada en `MDCCCLXXXVIII`" en lugar de "inaugurada en `1888`"). Puede que incluso los haya visto en índices o referencias bibliográficas. Es un sistema de representar números que viene de los tiempos del antiguo imperio romano (de ahí su nombre).

- 7.4. Uso de la sintaxis {n,m}

  En la sección anterior, tratamos con un patrón donde el mismo carácter podía repetirse hasta tres veces. Hay otra manera de expresar esto con expresiones regulares, que algunas personas encuentran más legible. Primero mire el método que hemos usado ya en los ejemplos anteriores.

- 7.5. Expresiones regulares prolijas

  Hasta ahora sólo hemos tratado con lo que llamaremos expresiones regulares “compactas”. Como verá, son difíciles de leer, e incluso si uno sabe lo que hacen, eso no garantiza que seamos capaces de comprenderlas dentro de seis meses. Lo que estamos necesitando es documentación en línea.

- 7.6. Caso de estudio: análisis de números de teléfono

  Por ahora se ha concentrado en patrones completos. Cada patrón coincide, o no. Pero las expresiones regulares son mucho más potentes que eso. Cuando una expresión regular *coincide*, puede extraer partes concretas. Puede saber qué es lo que causó la coincidencia.

- 7.7. Resumen

  Ésta es sólo la minúscula punta del iceberg de lo que pueden hacer las expresiones regulares. En otras palabras, incluso aunque esté completamente saturado con ellas ahora mismo, créame, todavía no ha visto nada.

Capítulo 8. Procesamiento de HTML

- 8.1. Inmersión

  A menudo veo preguntas en comp.lang.python parecidas a “¿Cómo puedo obtener una lista de todas las [cabeceras | imágenes | enlaces] en mi documento HTML?” “¿Cómo analizo/traduzco/manipulo el texto de mi documento HTML pero sin tocar las etiquetas?” “¿Cómo puedo añadir/eliminar/poner comillas a los atributos de

mis etiquetas HTML de una sola vez?" Este capítulo responderá todas esas preguntas.

- 8.2. Presentación de sgmllib.py

  El procesamiento de HTML se divide en tres pasos: obtener del HTML sus partes constitutivas, manipular las partes y reconstituirlas en un documento HTML. El primero paso lo realiza `sgmllib.py`, una parte de la biblioteca estándar de Python.

- 8.3. Extracción de datos de documentos HTML

  Para extraer datos de documentos HTML derivaremos la clase `SGMLParser` y definiremos métodos para cada etiqueta o entidad que queramos capturar.

- 8.4. Presentación de BaseHTMLProcessor.py

  `SGMLParser` no produce nada por sí mismo. Analiza, analiza y analiza, e invoca métodos por cada cosa interesante que encuentra, pero los métodos no hacen nada. `SGMLParser` es un *consumidor* de HTML: toma un HTML y lo divide en trozos pequeños y estructurados. Como vio en la sección anterior, puede derivar `SGMLParser` para definir clases que capturen etiquetas específicas y produzcan cosas útiles, como una lista de todos los enlaces en una página web. Ahora llevará esto un paso más allá, definiendo una clase que capture todo lo que `SGMLParser` le lance, reconstruyendo el documento HTML por completo. En términos técnicos, esta clase será un *productor* de HTML.

- 8.5. locals y globals

Hagamos por un minuto un inciso entre tanto procesamiento de HTML y hablemos sobre la manera en que Python gestiona las variables. Python incorpora dos funciones, `locals` y `globals`, que proporcionan acceso de tipo diccionario a las variables locales y globales.

- 8.6. Cadenas de formato basadas en diccionarios

  Hay una forma alternativa de dar formato a cadenas que usa diccionarios en lugar de tuplas de valores.

- 8.7. Poner comillas a los valores de los atributos

  Una pregunta habitual en comp.lang.python es "Tengo unos documentos HTML con valores de atributos sin comillas, y quiero corregirlos todos. ¿Cómo puedo hacerlo?"[9] (Generalmente sucede cuando un jefe de proyecto que ha abrazado la religión HTML-es-un-estándar se une a un proyecto y proclama que todas las páginas deben pasar el validador de HTML. Los valores de atributos sin comillas son una violación habitual del estándar HTML). Cualquiera que sea la razón, es fácil corregir el problema de los valores de atributo sin comillas alimentando a `BaseHTMLProcessor` con HTML.

- 8.8. Presentación de dialect.py

  `Dialectizer` es una descendiente sencilla (y tonta) de `BaseHTMLProcessor`. Hace una serie de sustituciones sobre bloques de texto, pero se asegura de que cualquier cosa dentro de un bloque `<pre>...</pre>` pase sin alteraciones.

- 8.9. Todo junto

Es hora de darle buen uso a todo lo que ha aprendido hasta ahora. Espero que haya prestado atención.

- 8.10. Resumen

  Python le proporciona una herramienta potente, `sgmllib.py`, para manipular HTML volviendo su estructura en un modelo de objeto. Puede usar esta herramienta de diferentes maneras.

Capítulo 9. Procesamiento de XML

- 9.1. Inmersión

  Hay dos maneras básicas de trabajar con XML. Una se denomina SAX ("Simple API for XML"), y funciona leyendo un poco de XML cada vez, invocando un método por cada elemento que encuentra (si leyó Capítulo 8, *Procesamiento de HTML*, esto debería serle familiar, porque es la manera en que trabaja el módulo `sgmllib`). La otra se llama DOM ("Document Object Model"), y funciona leyendo el documento XML completo para crear una representación interna utilizando clases nativas de Python enlazadas en una estructura de árbol. Python tiene módulos estándar para ambos tipos de análisis, pero en este capítulo sólo trataremos el uso de DOM.

- 9.2. Paquetes

  Analizar un documentl XML es algo muy sencillo: una línea de código. Sin embargo, antes de llegar a esa línea de código hará falta dar un pequeño rodeo para hablar sobre los paquetes.

- 9.3. Análisis de XML

Como iba diciendo, analizar un documento XML es muy sencillo: una línea de código. A dónde ir partiendo de eso es cosa suya.

- 9.4. Unicode

  Unicode es un sistema para representar caracteres de todos los diferentes idiomas en el mundo. Cuando Python analiza un documento XML, todos los datos se almacenan en memoria como unicode.

- 9.5. Búsqueda de elementos

  Recorrer un documento XML saltando por cada nodo puede ser tedioso. Si está buscando algo en particular, escondido dentro del documento XML, puede usar un atajo para encontrarlo rápidamente: `getElementsByTagName`.

- 9.6. Acceso a atributos de elementos

  Los elementos de XML pueden tener uno o más atributos, y es increíblemente sencillo acceder a ellos una vez analizado el documento XML.

- 9.7. Transición

  Bien, esto era el material más duro sobre XML. El siguiente capítulo continuará usando estos mismos programas de ejemplo, pero centrándose en otros aspectos que hacen al programa más flexible: uso de flujos[12] para proceso de entrada, uso de `getattr` para despachar métodos, y uso de opciones en la línea de órdenes para permitir a los usuarios reconfigurar el programa sin cambiar el código.

Capítulo 10. Scripts y flujos

- 10.1. Abstracción de fuentes de datos

  Uno de los puntos más fuertes de Python es su enlace dinámico, y un uso muy potente del enlace dinámico es el *objeto tipo fichero*.

- 10.2. Entrada, salida y error estándar

  Los usuarios de UNIX ya estarán familiarizados con el concepto de entrada estándar, salida estándar y salida estándar de error. Esta sección es para los demás.

- 10.3. Caché de búsqueda de nodos

  `kgp.py` emplea varios trucos que pueden o no serle útiles en el procesamiento de XML. El primero aprovecha la estructura consistente de los documentos de entrada para construir una caché de nodos.

- 10.4. Encontrar hijos directos de un nodo

  Otra técnica útil cuando se analizan documentos XML es encontrar todos los elementos que sean hijos directos de un elemento en particular. Por ejemplo, en los ficheros de gramáticas un elemento `ref` puede tener varios elementos `p`, cada uno de los cuales puede contener muchas cosas, incluyendo otros elementos `p`. Queremos encontrar sólo los elementos `p` que son hijos de `ref`, no elementos `p` que son hijos de otros elementos `p`.

- 10.5. Creación de manejadores diferentes por tipo de nodo

  El tercer consejo útil para procesamiento de XML implica separar el código en funciones lógicas, basándose en tipos de nodo y nombres de elemento. Los documentos XML

analizados se componen de varios tipos de nodos que representa cada uno un objeto de Python. El nivel raíz del documento en sí lo representa un objeto `Document`. El `Document` contiene uno o más objetos `Element` (por las etiquetas XML), cada uno de los cuales contiene otros objetos `Element`, `Text` (para zonas de texto) o `Comment` (para los comentarios). Python hace sencillo escribir algo que separe la lógica por cada tipo de nodo.

- 10.6. Tratamiento de los argumentos en línea de órdenes

  Python admite lacreación de programas que se pueden ejecutar desde la línea de órdenes, junto con argumentos y opciones tanto de estilo corto como largo para especificar varias opciones. Nada de esto es específico al XML, pero este *script* hace buen uso del tratamiento de la línea de órdenes, así que parece buena idea mencionarlo.

- 10.7. Todo junto

  A cubierto mucho terreno. Volvamos atrás y comprobemos cómo se unen todas las piezas.

- 10.8. Resumen

  Python incluye bibliotecas potentes para el análisis y la manipulación de documentos XML. `minidom` toma un fichero XML y lo convierte en objetos de Python, proporcionando acceso aleatorio a elementos arbitrarios. Aún más, este capítulo muestra cómo se puede usar Python para crear un *script* de línea de órdenes, completo con sus opciones, argumentos, gestión de errores, e incluso la capacidad de tomar como entrada el resultado de un programa anterior mediante una tubería.

Capítulo 11. Servicios Web HTTP

- 11.1. Inmersión

  Hemos aprendido cosas sobre procesamiento de HTML y de XML, y por el camino también vio cómo descargar una página web y analizar XML de una URL, pero profundicemos algo más en el tema general de los servicios web HTTP.

- 11.2. Cómo no obtener datos mediante HTTP

  Digamos que quiere descargar un recurso mediante HTTP, tal como un *feed* sindicado Atom. Pero no sólo queremos descargarlo una vez; queremos descargarlo una y otra vez, cada hora, para obtener las últimas noticias de un sitio que nos ofrece noticias sindicadas. Hagámoslo primero a la manera sucia y rápida, y luego veremos cómo hacerlo mejor.

- 11.3. Características de HTTP

  Hay cinco características importantes de HTTP que debería tener en cuenta.

- 11.4. Depuración de servicios web HTTP

  Primero vamos a activar las características de depuración de la biblioteca de HTTP de Python y veamos qué está viajando por los cables. Esto será útil durante el capítulo según añadamos características.

- 11.5. Establecer el User-Agent

  El primer paso para mejorar nuestro cliente de servicios web HTTP es identificarnos adecuadamente con `User-`

`Agent`. Para hacerlo nos hace falta pasar de la `urllib` básica y zambullirnos en `urllib2`.

- 11.6. Tratamiento de Last-Modified y ETag

  Ahora que sabe cómo añadir cabeceras HTTP personalizadas a la consulta al servicio web, veamos cómo añadir la funcionalidad de las cabeceras `Last-Modified` y `ETag`.

- 11.7. Manejo de redirecciones

  Puede admitir redirecciones permanentes y temporales usando un tipo diferente de manipulador de URL.

- 11.8. Tratamiento de datos comprimidos

  La última característica importante de HTTP que queremos tratar es la compresión. Muchos servicios web tienen la capacidad de enviar los datos comprimidos, lo que puede rebajar en un 60% o más la cantidad de datos a enviar. Esto se aplica especialmente a los servicios web XML, ya que los datos XML se comprimen bastante bien.

- 11.9. Todo junto

  Ya hemos visto todas las partes necesarias para construir un cliente inteligente de servicios web HTTP. Ahora veamos cómo encaja todo.

- 11.10. Resumen

  `openanything.py` y sus funciones deberían tener sentido ahora.

Capítulo 12. Servicios web SOAP

- 12.1. Inmersión

  Usted usa Google, ¿cierto? Es un motor de búsquedas popular. ¿Ha deseado alguna vez poder acceder a los resultados de búsquedas de Google de forma programática? Ahora puede. Aquí tiene un programa que busca en Google desde Python.

- 12.2. Instalación de las bibliotecas de SOAP

  Al contrario que el resto de código de este libro, este capítulo se apoya en bibliotecas que no se incluyen preinstaladas den Python.

- 12.3. Primeros pasos con SOAP

  El corazón de SOAP es la capacidad de invocar funciones remotas. Hay varios servidores SOAP de acceso público que proporcionan funciones simples con propósito de demostración.

- 12.4. Depuración de servicios web SOAP

  Las bibliotecas de SOAP proporcionan una manera sencilla de comprobar lo que está sucediendo tras la escena.

- 12.5. Presentación de WSDL

  La clase `SOAPProxy` hace proxy de llamadas locales a métodos y las convierte de forma transparente en invocaciones a métodos SOAP remotos. Como ya ha visto esto lleva mucho trabajo, y el `SOAPProxy` lo hace rápida y transparentemente. Lo que no hace es proporcionarnos ningún tipo de método para introspección.

- 12.6. Introspección de servicios web SOAP con WSDL

Como muchas otras cosas en el campo de los servicios web, WSDL tiene una historia larga y veleidosa, llena de conflictos e intrigas políticas. Me saltaré toda esta historia ya que me aburre hasta lo indecible. Hay otros estándares que intentaron hacer cosas similares, pero acabó ganando WSDL así que aprendamos cómo usarlo.

- 12.7. Búsqueda en Google

  Volvamos finalmente al ejemplo de código que vio al comienzo de este capítulo, que hace algo más útil y excitante que obtener la temperatura.

- 12.8. Solución de problemas en servicios web SOAP

  Por supuesto, el mundo de los servicios web SOAP no es todo luz y felicidad. Algunas veces las cosas van mal.

- 12.9. Resumen

  Los servicios web SOAP son muy complicados. La especificación es muy ambiciosa e intenta cubrir muchos casos de uso de servicios web. Este capítulo ha tocado algunos de los casos más sencillos.

Capítulo 13. Pruebas unitarias (Unit Testing)

- 13.1. Introducción a los números romanos

  En capítulos interiores, se “sumergió” dando un vistazo rápido al código e intentándo comprenderlo lo más rápidamente posible. Ahora que tiene bastante Python a sus espaldas, vamos a retroceder y ver los pasos que se dan *antes* de escribir el código.

- 13.2. Inmersión

Ahora que ya hemos definido completamente el comportamiento esperado de nuestras funciones de conversión, vamos a hacer algo un poco inesperado: vamos a escribir una batería de pruebas que ponga estas funciones contra las cuerdas y se asegure de que se comportan de la manera en que queremos. Ha leído bien: va a escribir código que pruebe código que aún no hemos escrito.

- 13.3. Presentación de romantest.py

  Ésta es la batería de pruebas completa para las funciones de conversión de números romanos, que todavía no están escritas, pero en el futuro estarán en `roman.py`. No es inmediatamente obvio cómo encaja todo esto; ninguna de las clases o funciones hace referencia a las otras. Hay buenas razones para esto, como veremos en breve.

- 13.4. Prueba de éxito

  La parte más fundamental de una prueba unitaria es la construcción de casos de prueba individuales. Un caso de prueba responde a una única pregunta sobre el código que está probando.

- 13.5. Prueba de fallo

  No es suficiente probar que las funciones tienen éxito cuando se les pasa valores correctos; también debe probar que fallarán si se les da una entrada incorrecta. Y no sólo cualquier tipo de fallo; deben fallar de la manera esperada.

- 13.6. Pruebas de cordura

  A menudo se encontrará conque un código unitario contiene un conjunto de funciones recíprocas normalmente

en forma de funciones de conversión en que una convierte de A a B y la otra de B a A. En estos casos es útil crear "pruebas de cordura" (*sanity checks* para asegurarse de que puede convertir de A a B y de vuelta a A sin perder precisión, incurrir en errores de redondeo o encontrar ningún otro tipo de fallo.

- 15.3. Refactorización

  Lo mejor de las pruebas unitarias exhaustivas no es la sensación que le queda cuando todos los casos de prueba terminan por pasar, o incluso la que le llega cuando alguien le acusa de romper su código y usted puede *probar* realmente que no lo hizo. Lo mejor de las pruebas unitarias es que le da la libertad de refactorizar sin piedad.

- 15.4. Epílogo

  El lector inteligente leería la sección anterior y la llevaría al siguiente nivel. El mayor dolor de cabeza (y lastre al rendimiento) en el programa tal como está escrito es la expresión regular, que precisamos porque no tenemos otra manera de hacer disección de un número romano. Pero sólo hay 5000; ¿por qué no podemos generar una tabla de búsqueda una vez y luego limitarnos a leerla? La idea se vuelve aún mejor cuando nos damos cuenta de que no necesitamos usar expresiones regulares para nada. Mientras creamos la tabla para convertir enteros a números romanos podemos construir la tabla inversa que convierta romanos a enteros.

- 15.5. Resumen

  Las pruebas unitarias son un concepto podente que, si se implementa adecuadamente, puede tanto reducir el coste de mantenimiento como incrementar la flexibilidad en cualquier proyecto a largo plazo. Es importante también entender que las pruebas unitarias no son la panacea, un Resolutor Mágico de Problemas o una bala de plata. Escribir buenos casos de prueba es duro, y mantenerlos actualizados necesita disciplina (especialmente cuando los

clientes están gritando que quieren arreglos para fallos críticos). Las pruebas unitarias no sustituyen otras formas de comprobación, incluyendo las pruebas funcionales, las de integración y las de aceptación del cliente. Pero son factibles y funcionan, y una vez que las ha visto trabajando uno se pregunta cómo ha podido pasar sin ellas hasta ahora.

Capítulo 16. Programación Funcional

- 16.1. Inmersión

  En Capítulo 13, *Pruebas unitarias (Unit Testing)* aprendimos la filosofía de las pruebas unitarias. En Capítulo 14, *Programación Test-First* implementamos paso a paso una prueba unitaria en Python. En Capítulo 15, *Refactorización* vimos cómo las pruebas unitarias hacen más sencilla la refactorización a gran escala. Este capítulo se apoyará en esos programas de ejemplo, pero aquí nos centraremos en técnicas más avanzadas específicas de Python, en lugar de en las pruebas unitarias en sí.

- 16.2. Encontrar la ruta

  A veces es útil, cuando ejecutamos *scripts* de Python desde la línea de órdenes, saber dónde está situado en el disco el *scripts* que está en marcha.

- 16.3. Revisión del filtrado de listas

  Ya está familiarizado con el uso de listas por comprensión para filtrar listas. Hay otra manera de conseguir lo mismo que algunas personas consideran más expresiva.

- 16.4. Revisión de la relación de listas

Quiero hablar sobre los sustantivos en plural. También sobre funciones que devuelven otras funciones, expresiones regulares avanzadas y generadores. Los generadores son nuevos en Python 2.3. Pero primero hablemos sobre cómo hacer nombres en plural[20].

- 17.2. plural.py, fase 1

  Así que estamos mirando las palabras, que al menos en inglés son cadenas de caracteres. Y tenemos reglas que dicen que debe encontrar diferentes combinaciones de caracteres y luego hacer cosas distintas con ellos. Esto suena a trabajo para las expresiones regulares.

- 17.3. plural.py, fase 2

  Ahora vamos a añadir un nivel de abstracción. Empezamos definiendo una lista de reglas: si pasa esto, haz lo aquello, si no pasa la siguiente regla. Compliquemos temporalmente parte del programa para poder simplificar otra.

- 17.4. plural.py, fase 3

  No es realmente necesario definir una función aparte para cada regla de coincidencia y modificación. Nunca las invocamos directamente; las definimos en la lista `rules` y las llamamos a través suya. Vamos a reducir el perfil de la definición de las reglas haciéndolas anónimas.

- 17.5. plural.py, fase 4

  Eliminemos la duplicación del código para que sea más sencillo definir nuevas reglas.

Lo primero que comprueba la función Soundex es si la entrada es una cadena de letras que no esté vacía. ¿Cual es la mejor manera de hacerlo?

- 18.4. Optimización de búsquedas en diccionarios

  El segundo paso del algoritmo Soundex es convertir los caracteres en dígitos según un patrón específico. ¿Cual es la mejor manera de hacer esto?

- 18.5. Optimización de operaciones con listas

  El tercer paso en el algoritmo Soundex es eliminar dígitos consecutivos duplicados. ¿Cual es la mejor manera de hacerlo?

- 18.6. Optimización de manipulación de cadenas

  El último paso del algoritmo Soundex es rellenar los resultados cortos con ceros y truncar los largos. ¿Cual es la mejor manera de hacerlo?

- 18.7. Resumen

  Este capítulo ha ilustrado varios aspectos importantes del ajuste de rendimiento en Python y en general.

Apéndice C. Trucos y consejos

Capítulo 1. Instalación de Python

Capítulo 2. Su primer programa en Python

- 2.1. Inmersión

En el IDE ActivePython para Windows puede ejecutar el programa de Python que esté editando escogiendo File->Run... (**Ctrl-R**). La salida se muestra en la pantalla interactiva.

En el IDE de Python de Mac OS puede ejecutar un programa de Python con Python->Run window... (**Cmd-R**), pero hay una opción importante que debe activar antes. Abra el fichero `.py` en el IDE, y muestre el menú de opciones pulsando en el triángulo negro en la esquina superior derecha de la ventana, asegurándose de que está marcada la opción Run as __main__. Esta preferencia está asociada a cada fichero por separado, pero sólo tendrá que marcarla una vez por cada uno.

En sistemas compatibles con UNIX (incluido Mac OS X), puede ejecutar un programa de Python desde la línea de órdenes: **`python odbchelper.py`**

- 2.2. Declaración de funciones

En Visual Basic las funciones (devuelven un valor) comienzan con `function`, y las subrutinas (no devuelven un valor) lo hacen con `sub`. En Python no tenemos subrutinas. Todo son funciones, todas las funciones devuelven un valor (incluso si es `None`) y todas las funciones comienzan por `def`.

En Java, C++ y otros lenguajes de tipo estático debe especificar el tipo de dato del valor de retorno de la función y de cada uno de sus argumentos. En Python nunca especificará de forma explícita el tipo de dato de nada. Python lleva un registro interno del tipo de dato basándose en el valor asignado.

- 2.3. Documentación de funciones

Las comillas triples también son una manera sencilla de definir una cadena que contenga comillas tanto simples como dobles, como `qq/.../` en Perl.

Muchos IDE de Python utilizan la `cadena de documentación` para proporcionar una ayuda sensible al contexto, de manera que cuando escriba el nombre de una función aparezca su `cadena de documentación` como ayuda. Esto puede ser increíblemente útil, pero lo será tanto como buenas las `cadenas de documentación` que usted escriba.

- 2.4. Todo es un objeto

`import` en Python es como `require` en Perl. Una vez que hace `import` sobre un módulo de Python, puede acceder a sus funciones con *`módulo.función`*; una vez que hace `require` sobre un módulo de Perl, puede acceder a sus funciones con *`módulo::función`*.

- 2.5. Sangrado (indentado) de código

Python utiliza retornos de carro para separar sentencias y los dos puntos y el sangrado para reconocer bloques de código. C++ y Java usan puntos y coma para separar sentencias, y llaves para indicar

bloques de código.

- 2.6. Prueba de módulos

  

  Al igual que C, Python utiliza == para la comparación y = para la asignación. Al contrario que C, Python no permite la asignación embebida, de manera que no existe la posibilidad de asignar un valor accidentalmente donde deseaba hacer una comparación.

  

  En MacPython, hay que dar un paso adicional para hacer que funcione el truco `if __name__`. Muestre el menú de opciones pulsando el triángulo negro en la esquina superior derecha de la ventana, y asegúrese de que está marcado Run as __main__.

Capítulo 3. Tipos de dato nativos

- 3.1. Presentación de los diccionarios

  

  Un diccionario en Python es como un hash en Perl. En Perl, las variables que almacenan hashes siempre empiezan con un carácter `%`. En Python las variables se pueden llamar de cualquier manera, y Python sabe su tipo internamente.

  

  Un diccionario en Python es como una instancia de la clase `Hashtable` de Java.

  

  Un diccionario en Python es como una instancia del objeto `Scripting.Dictionary` de Visual Basic.

- 3.1.2. Modificar diccionarios

  

  Los diccionarios no tienen concepto de orden entre sus elementos. Es

incorrecto decir que los elementos están “desordenados”, ya que simplemente no tienen orden. Esto es una distinción importante que le irritará cuando intente acceder a los elementos de un diccionario en un orden específico y repetible (por ejemplo, en orden alfabético por clave). Hay maneras de hacer esto, pero no vienen de serie en el diccionario.

- 3.2. Presentación de las listas

Una lista de Python es como un array en Perl. En Perl, las variables que almacenan arrays siempre empiezan con el carácter `@`; en Python, las variables se pueden llamar de cualquier manera, y Python se ocupa de saber el tipo que tienen.

Una lista en Python es mucho más que un array en Java (aunque puede usarse como uno si es realmente eso todo lo que quiere en esta vida). Una mejor analogía podría ser la clase `ArrayList`, que puede contener objetos arbitrarios y expandirse de forma dinámica según se añaden otros nuevos.

- 3.2.3. Buscar en listas

Antes de la versión 2.2.1, Python no tenía un tipo booleano. Para compensarlo, Python aceptaba casi cualquier cosa en un contexto booleano (como una sentencia `if`), de acuerdo a las siguientes reglas:

  - `0` es falso; el resto de los números son verdaderos.
  - Una cadena vacía (`""`) es falso, cualquier otra cadena es verdadera.
  - Una lista vacía (`[]`) es falso; el resto de las listas son verdaderas.
  - Una tupla vacía (`()`) es falso; el resto de las tuplas son

verdaderas.

- Un diccionario vacío (`{}`) es falso; todos los otros diccionarios son verdaderos.

Estas reglas siguen aplicándose en Python 2.2.1 y siguientes, pero ahora además puedes usar un verdadero booleano, que tiene el valor de `True` o `False`. Tenga en cuenta las mayúsculas; estos valores, como todo lo demás en Python, las distinguen.

- 3.3. Presentación de las tuplas

Las tuplas se pueden convertir en listas, y viceversa. La función incorporada `tuple` toma una lista y devuelve una tupla con los mismos elementos, y la función `list` toma una tupla y devuelve una lista. En efecto, `tuple` "congela" una lista, y `list` "descongela" una `tuple`.

- 3.4. Declaración de variables

Cuando una orden se divide en varias líneas con la marca de continuación de línea (“\”), las siguientes líneas se pueden sangrar de cualquier manera; la habitual sangrado astringente de Python no se aplica aquí. Si su IDE de Python autosangra la línea a continuación, probablemente debería aceptar este comportamiento a menos que tenga una imperiosa razón para no hacerlo.

- 3.5. Formato de cadenas

El formato de cadenas en Python usa la misma sintaxis que la función `sprintf` en C.

- 3.7. Unir listas y dividir cadenas

`join` funciona sólo sobre listas de cadenas; no hace ningún tipo de conversión de tipos. Juntar una lista que tenga uno o más elementos que no sean cadenas provocará una excepción.

unacadena`.split(`*delimitador*`, 1)` es una forma útil de buscar una subcadena dentro de una cadena, y trabajar después con todo lo que hay antes de esa subcadena (que es el primer elemento de la lista devuelta) y todo lo que hay detrás (el segundo elemento).

Capítulo 4. El poder de la introspección

- 4.2. Argumentos opcionales y con nombre

Lo único que necesita para invocar a una función es especificar un valor (del modo que sea) para cada argumento obligatorio; el modo y el orden en que se haga esto depende de usted.

- 4.3.3. Funciones incorporadas

Python se acompaña de excelentes manuales de referencia, que debería usted leer detenidamente para aprender todos los módulos que Python ofrece. Pero mientras en la mayoría de lenguajes debe usted volver continuamente sobre los manuales o las páginas de manual para recordar cómo se usan estos módulos, Python está autodocumentado en su mayoría.

- 4.7. Utilización de las funciones lambda

Las funciones `lambda` son una cuestión de estilo. Su uso nunca es necesario. En cualquier sitio en que puedan utilizarse, se puede definir una función normal separada y utilizarla en su lugar. Yo las utilizo en lugares donde deseo encapsulación, código no reutilizable

que no ensucie mi propio código con un montón de pequeñas funciones de una sola línea.

- 4.8. Todo junto

En SQL, se utiliza `IS NULL` en vez de `= NULL` para comparar un valor nulo. En Python puede usar tanto `== None` como `is None`, pero `is None` es más rápido.

Capítulo 5. Objetos y orientación a objetos

- 5.2. Importar módulos usando from módulo import

`from` *`módulo`* `import *` en Python es como `use` *`módulo`* en Perl; `import` *`módulo`* en Python es como `require` *`módulo`* en Perl.

`from` *`módulo`* `import *` en Python es como `import` *`módulo`*`.*` en Java; `import` *`módulo`* en Python es como `import` *`módulo`* en Java.

Utilice `from module import *` lo menos posible, porque hace difícil determinar de dónde vino una función o atributo en particular, y complica más la depuración y el refactorizado.

- 5.3. Definición de clases

La sentencia `pass` de Python es como unas llaves vacías (`{}`) en Java o C.

En Python, el ancestro de una clase se lista entre paréntesis inmediatamente tras el nombre de la clase. No hay palabras reservadas especiales como `extends` en Java.

- 5.3.1. Inicialización y programación de clases

Por convención, el primer argumento de cualquier clase de Python (la referencia a la instancia) se denomina `self`. Este argumento cumple el papel de la palabra reservada `this` en C++ o Java, pero `self` no es una palabra reservada en Python, sino una mera convención. De todas maneras, por favor no use otro nombre sino `self`; es una convención muy extendida.

- 5.3.2. Saber cuándo usar self e __init__

Los métodos `__init__` son opcionales, pero cuando define uno, debe recordar llamar explícitamente al método `__init__` del ancestro (si define uno). Suele ocurrir que siempre que un descendiente quiera extender el comportamiento de un ancestro, el método descendiente deba llamar al del ancestro en el momento adecuado, con los argumentos adecuados.

- 5.4. Instanciación de clases

En Python, simplemente invocamos a una clase como si fuese una función para crear una nueva instancia de la clase. No hay operador `new` explícito como en C++ o Java.

- 5.5. Exploración de UserDict: Una clase cápsula

En el IDE ActivePython en Windows, puede abrir rápidamente cualquier módulo de su ruta de bibliotecas mediante File->Locate... (**Ctrl-L**).

Java y Powerbuilder admiten la sobrecarga de funciones por lista de argumentos, es decir una clase puede tener varios métodos con el mismo nombre, pero con argumentos en distinta cantidad, o de

distinto tipo. Otros lenguajes (notablemente PL/SQL) incluso admiten sobrecarga de funciones por nombre de argumento; es decir una clase puede tener varios métodos con el mismo nombre y número de argumentos de incluso el mismo tipo, pero con diferentes nombres de argumento. Python no admite ninguno de estos casos; no hay forma de sobrecarga de funciones. Los métodos se definen sólo por su nombre, y hay un único método por clase con un nombre dado. De manera que si una clase sucesora tiene un método `__init__`, *siempre* sustituye al método `__init__` de su ancestro, incluso si éste lo define con una lista de argumentos diferentes. Y se aplica lo mismo a cualquier otro método.

Guido, el autor original de Python, explica el reemplazo de métodos así: "Las clases derivadas pueden reemplazar los métodos de sus clases base. Dado que los métodos no tienen privilegios especiales para llamar a otros métodos del mismo objeto, un método de una clase base que llama a otro método definido en la misma clase base, puede en realidad estar llamando a un método de una clase derivada que la reemplaza (para programadores de C++: todos los métodos de Python son virtuales a los efectos)". Si esto no tiene sentido para usted (a mí me confunde sobremanera), ignórelo. Sólo pensaba que debía comentarlo.

Asigne siempre un valor inicial a todos los atributos de datos de una instancia en el método `__init__`. Le quitará horas de depuración más adelante, en busca de excepciones `AttributeError` debido a que está haciendo referencia a atributos sin inicializar (y por tanto inexistentes).

En las versiones de Python previas a la 2.20, no podía derivar directamente tipos de datos internos como cadenas, listas y

diccionarios. Para compensarlo, Python proporcionaba clases encapsulantes que imitaban el comportamiento de estos tipos: `UserString`, `UserList`, y `UserDict`. Usando una combinación de métodos normales y especiales, la clase `UserDict` hace una excelente imitación de un diccionario. En Python 2.2 y posteriores, puede hacer que una clase herede directamente de tipos incorporados como `dict`. Muestro esto en los ejemplos que acompañan al libro, en `fileinfo_fromdict.py`.

- 5.6.1. Consultar y modificar elementos

Cuando se accede a atributos de datos dentro de una clase, necesitamos calificar el nombre del atributo: `self.`*`atributo`*. Cuando llamamos a otros métodos dentro de una clase, necesitamos calificar el nombre del método: `self.`*`método`*.

- 5.7. Métodos especiales avanzados

En Java, determinamos si dos variables de cadena referencian la misma posición física de memoria usando `str1 == str2`. A esto se le denomina *identidad de objetos*, y en Python se escribe así: `str1 is str2`. Para comparar valores de cadenas en Java, usaríamos`str1.equals(str2)`; en Python, usaríamos `str1 == str2`. Los programadores de Java a los que se les haya enseñado a creer que el mundo es un lugar mejor porque == en Java compara la identidad en lugar del valor pueden tener dificultades ajustándose a la falta de este tipo de "*gotchas*" en Python.

Mientras que otros lenguajes orientados a objeto sólo le permitirán definir el modelo físico de un objeto ("este objeto tiene un método `GetLength`"), los métodos especiales de Python como `__len__` le permiten definir el modelo lógico de un objeto ("este objeto tiene

una longitud").

- 5.8. Presentación de los atributos de clase

  

  En Java, tanto las variables estáticas (llamadas atributos de clase en Python) como las variables de instancia (llamadas atributos de datos en Python) se declaran inmediatamente en la definición de la clase (unas con la palabra clave `static`, otras sin ella). En Python, sólo se pueden definir aquí los atributos de clase; los atributos de datos se definen en el método `__init__`.

  

  No hay constantes en Python. Todo puede cambiar si lo intenta con ahínco. Esto se ajusta a uno de los principios básicos de Python: los comportamientos inadecuados sólo deben desaconsejarse, no prohibirse. Si en realidad quiere cambiar el valor de `None`, puede hacerlo, pero no venga luego llorando si es imposible depurar su código.

- 5.9. Funciones privadas

  

  En Python, todos los métodos especiales (como `__setitem__`) y atributos incorporados (como `__doc__`) siguen una convención estándar: empiezan y terminan con dos guiones bajos. No ponga a sus propios métodos ni atributos nombres así, porque sólo le confudirán a usted (y otros) más adelante.

Capítulo 6. Excepciones y gestión de ficheros

- 6.1. Gestión de excepciones

  

  Python utiliza `try...except` para gestionar las excepciones y `raise` para generarlas. Java y C++ usan `try...catch` para gestionarlas, y `throw` para generarlas.

- 6.5. Trabajo con directorios

  

  Siempre que sea posible, debería usar las funciones de `os` y `os.path` para manipulaciones sobre ficheros, directorios y rutas. Estos módulos encapsulan los específicos de cada plataforma, de manera que funciones como `os.path.split` funcionen en UNIX, Windows, Mac OS y cualquier otra plataforma en que funcione Python.

Capítulo 7. Expresiones regulares

- 7.4. Uso de la sintaxis {n,m}

  

  No hay manera de determinar programáticamente si dos expresiones regulares son equivalentes. Lo mejor que puede hacer es escribir varios casos de prueba para asegurarse de que se comporta correctamente con todos los tipos de entrada relevantes. Hablaremos más adelante en este mismo libro sobre la escritura de casos de prueba.

Capítulo 8. Procesamiento de HTML

- 8.2. Presentación de sgmllib.py

  

  Python 2.0 sufría un fallo debido al que `SGMLParser` no podía reconocer declaraciones (no se llamaba nunca a `handle_decl`), lo que quiere decir que se ignoraban los `DOCTYPE` sin advertirlo. Esto quedó corregido en Python 2.1.

  

  En el IDE ActivePython para Windows puede especificar argumentos en la línea de órdenes desde el cuadro de diálogo “Run script”. Si incluye varios argumentos sepárelos con espacios.

- 8.4. Presentación de BaseHTMLProcessor.py

La especificación de HTML precisa que todo lo que no sea HTML (como JavaScript para el cliente) debe estar encerrado dentro de comentarios de HTML, pero no todas las páginas web lo hacen correctamente (y todos los navegadores modernos hacen la vista gorda en ese caso). `BaseHTMLProcessor` no es tan permisivo; si un *script* no está adecuadamente embebido, será analizado como si fuera HTML. Por ejemplo, si el script contiene símbolos "igual" o "menor que", `SGMLParser` puede entender de incorrectamente que ha encontrado etiquetas y atributos. `SGMLParser` siempre convierte los nombres de etiquetas y atributos a minúsculas, lo que puede inutilizar el *script*, y `BaseHTMLProcessor` siempre encierra los valores de atributos dentro de comillas dobles (incluso si el documento original utiliza comillas simples o ningún tipo de comillas), lo que hará inútil el *script* con certeza. Proteja siempre sus *script* dentro de comentarios de HTML.

- 8.5. locals y globals

Python 2.2 introdujo un cambio sutil pero importante que afecta al orden de búsqueda en los espacios de nombre: los ámbitos anidados. En las versiones de Python anteriores a la 2.2, cuando hace referencia a una variable dentro de una función anidda o función `lambda`, Python buscará esa variable en el espacio de la función actual (anidada o `lambda`) y después en el espacio de nombres del módulo. Python 2.2 buscará la variable en el espacio de nombres de la función actual (anidada o `lambda`), *después en el espacio de su función madre*, y luego en el espacio del módulo. Python 2.1 puede funcionar de ambas maneras; por omisión lo hace como Python 2.0, pero puede añadir la siguiente línea al código al principio de su módulo para hacer que éste funcione como Python 2.2:

```
from __future__ import nested_scopes
```

Puede obtener dinámicamente el valor de variables arbitrarias usando las funciones `locals` y `globals`, proporcionando el nombre de la variable en una cadena. Esto imita la funcionalidad de la función `getattr`, que le permite acceder a funciones arbitrarias de forma dinámica proporcionando el nombre de la función en una cadena.

- 8.6. Cadenas de formato basadas en diccionarios

Usar cadenas de formato basadas en diccionarios con `locals` es una manera conveniente de hacer más legibles expresiones de cadenas de formato, pero tiene un precio. Llamar a locals tiene una ligera influencia en el rendimiento, ya que `locals` construye una copia del espacio de nombres local.

Capítulo 9. Procesamiento de XML

- 9.2. Paquetes

Un paquete es un directorio que contiene el fichero especial `__init__.py`. El fichero `__init__.py` define los atributos y métodos del paquete. No tiene por qué definir nada; puede ser un fichero vacío, pero ha de existir. Pero si no existe `__init__.py` el directorio es sólo eso, un directorio, no un paquete, y no puede ser importado o contener módulos u otros paquetes.

- 9.6. Acceso a atributos de elementos

Esta sección puede ser un poco confusa, debido a que la terminología se solapa. Los elementos de documentos XML tienen atributos, y los objetos de Python también. Cuando analizamos un

documento XML obtenemos un montón de objetos de Python que representan todas las partes del documento XML, y algunos de estos objetos de Python representan atributos de los elementos de XML. Pero los objetos (de Python) que representan los atributos (de XML) también tienen atributos (de Python), que se usan para acceder a varias partes del atributo (de XML) que representa el objeto. Le dije que era confuso. Estoy abierto a sugerencias para distinguirlos con más claridad.

Al igual que un diccionario, los atributos de un elemento XML no tienen orden. *Puede que* los atributos estén listados en un cierto orden en el documento XML original, y *Puede que* los objetos `Attr` estén listados en un cierto orden cuando se convierta el documento XML en objetos de Python, pero estos órdenes son arbitrarios y no deberían tener un significado especial. Siempre debería acceder a los atributos por su nombre, como claves de un diccionario.

Capítulo 10. Scripts y flujos

Capítulo 11. Servicios Web HTTP

- 11.6. Tratamiento de Last-Modified y ETag

En estos ejemplos el servidor HTTP ha respondido tanto a la cabecera `Last-Modified` como a `ETag`, pero no todos los servidores lo hacen. Como cliente de un servicio web debería estar preparado para usar ambos, pero debe programar de forma defensiva por si se da el caso de que el servidor sólo trabaje con uno de ellos, o con ninguno.

Capítulo 12. Servicios web SOAP

Capítulo 13. Pruebas unitarias (Unit Testing)

- 13.2. Inmersión

`unittest` está incluido en Python 2.1 y posteriores. Los usuarios de Python 2.0 pueden descargarlo de `pyunit.sourceforge.net`.

Capítulo 14. Programación Test-First

- 14.3. roman.py, fase 3

La cosa más importante que le puede decir una prueba unitaria exhaustiva es cuándo debe dejar de programar. Cuando una función pase todas sus pruebas unitarias, deje de programarla. Cuando un módulo pase todas sus pruebas unitarias, deje de programar en él.

- 14.5. roman.py, fase 5

Cuando hayan pasado todas sus pruebas, deje de programar.

Capítulo 15. Refactorización

- 15.3. Refactorización

Siempre que vaya a usar una expresión regular más de una vez, debería compilarla para obtener un objeto patrón y luego llamar directamente a los métodos del patrón.

Capítulo 16. Programación Funcional

- 16.2. Encontrar la ruta

No hace falta que existan los nombres de rutas y ficheros que se le pasan a `os.path.abspath`.

`os.path.abspath` no se limita a construir nombres completos de rutas, también los normaliza. Eso significa que si está en el directorio

`/usr/`, `os.path.abspath('bin/../local/bin')` devolverá `/usr/local/bin`. Normaliza la ruta haciéndola lo más sencilla posible. Si sólo quiere normalizar una ruta así, sin convertirla en completa, use `os.path.normpath`.

`os.path.abspath` es multiplataforma igual que las otras funciones del os módulos `os` y `os.path`. Los resultados parecerán ligeramente diferentes que en mis ejemplos si está usted usando Windows (que usa barras invertidas como separador de ruta) o Mac OS (que usa dos puntos - :), pero seguirá funcionando. Ésa es la idea principal del módulo `os`.

Capítulo 17. Funciones dinámicas

Capítulo 18. Ajustes de rendimiento

- 18.2. Uso del módulo timeit

Podemos usar el módulo `timeit` desde la línea de órdenes para probar un programa de Python que ya exista, sin modificar el código. Vea http://docs.python.org/lib/node396.html si desea documentación sobre las opciones para línea de órdenes.

El módulo `timeit` sólo sirve de algo si ya sabe qué parte de su código quiere optimizar. Si tiene un programa grande escrito en Python y no sabe dónde tiene los problemas de rendimiento, pruebe el módulo `hotshot`.

Apéndice D. Lista de ejemplos

Apéndice E. Historial de revisiones

26 de enero de 2005 (5.4-es.11)

- Correcciones varias. Completa la revisión de los dos primeros capítulos

24 de enero de 2005 (5.4-es.10)

- Completa la traducción del capítulo 18

23 de enero de 2005 (5.4-es.9)

- Completa la traducción del capítulo 17

21 de enero de 2005 (5.4-es.8)

- Completa la traducción del capítulo 16

17 de enero de 2005 (5.4-es.7)

- Completa la traducción de los capítulos 13, 14 y 15

16 de enero de 2005 (5.4-es.6)

- Completa la traducción de los capítulos 11 y 12

9 de enero de 2005 (5.4-es.5)

- Completa la traducción del capítulo 10

7 de enero de 2005 (5.4-es.4)

- Completa la traducción del capítulo 9
- Correcciones

6 de enero de 2005 (5.4-es.3)

- Varias correcciones tipográficas y de estilo, señaladas por mis atentos lectores. Muchas gracias a todos por sus desvelos
- Completa la traducción del capítulo 8

5 de enero de 2005 (5.4-es.2)

- Pequeñas correcciones tipográficas
- Completa la traducción del capítulo 7

4 de enero de 2005 (5.4-es.1)

- Revisión del material traducido
- Retomada la traducción en la versión 5.4 del original
- Completada la traducción hasta el capítulo 6, inclusive

30 de septiembre de 2001 (1.0)

- Versión inicial: capítulos 1 y 2 y material complementario.

Apéndice F. Sobre este libro

Este libro se escribió en DocBook XML usando Emacs, y se transformó en HTML utilizando el procesador XSLT SAXON de Michael Kay con una versión adaptada de las hojas de estilo XSL de Norman Walsh. A partir de ahí, se realizó la conversión a PDF con HTMLDoc, y a texto llano con w3m. Los listados de programas y los ejemplos se colorearon con una versión actualizada de `pyfontify.py`, de Just van Rossum, que se incluye entre los *scripts* de ejemplo.

Si está usted interesado en aprender más sobre DocBook para escribir textos técnicos, puede descargar los ficheros fuente XML, y los *scripts* de compilación, que incluyen también las hojas de estilo XSL modificadas que se usan para crear los distintos formatos. Debería leer también el libro canónico, *DocBook: The Definitive Guide*. Si desea hacer algo serio con DocBook, le recomiendo que se suscriba a las listas de correo de DocBook.