Table of Contents

Table of Contents

Inmersión en Python

17 de septiembre de 2001

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.

Capítulo 1. Conozcamos Python

1.1. Inmersión

He aquí un programa completo y operativo en Python.

Probablemente no tenga ningún sentido para usted. No se preocupe por ello: vamos a diseccionarlo línea a línea. Pero antes échele un vistazo por si puede sacar algo de él.

Ejemplo 1.1. `odbchelper.py`

Si aún no lo ha hecho, puede descargar este y otros ejemplos de este libro.

```
def buildConnectionString(params):
    """Build a connection string from a dictionary of parameters.

    Returns string."""
    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 observe qué ocurre.

> **Sugerencia: Ejecutar un módulo (Windows)**
> En el entorno de programación Python en Windows, puede ejecutar un módulo con File−>Run... (**Control−R**). La salida se muestra en la ventana interactiva.
>
> **Sugerencia: Ejecutar un módulo (Mac OS)**
> En el entorno de programación Python en Mac OS, puede ejecutar un módulo con Python−>Run window... (**Cmd−R**), pero antes debe seleccionar una opción importante: Abra el módulo en el entorno de programación, muestre el menú de opciones de módulo pulsando el triángulo negro de la esquina superior derecha de la ventana, y asegúrese de que "Run as __main__" está seleccionado. Esta opción se guarda junto con el módulo, de modo que sólo debe hacer esto una vez por cada módulo.
>
> **Sugerencia: Ejecutar un módulo (UNIX)**
> En los sistemas compatibles con UNIX (incluido Mac OS X), puede ejecutar un módulo desde la línea de órdenes: **`python odbchelper.py`**

Ejemplo 1.2. Salida de `odbchelper.py`

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

1.2. Declaración de funciones

Python, como la mayoría de los lenguajes, tiene funciones, pero no utiliza ficheros de cabecera independientes como C++ o secciones `interface`/`implementation` como Pascal. Cuando necesite una función, simplemente declárela e incluya el código.

Ejemplo 1.3. Declaración de la función `buildConnectionString`

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

Aquí hay que indicar varias cosas. En primer lugar, la palabra clave `def` inicia la declaración de la función, seguida por el nombre de la función, seguido por los argumentos entre paréntesis. Si hay más de un argumento (aquí no ocurre) se separan con comas.

En segundo lugar, la función no define el tipo del valor de retorno. Las funciones de Python no especifican el tipo de su valor de retorno; ni siquiera especifican si devuelven o no un valor. De hecho, todas las funciones de Python devuelven un valor; si la función incluye una sentencia `return`, devolverá ese valor; en otro caso devolverá `None`, el valor nulo de Python.

> **Nota: Python frente a Visual Basic: valores de retorno**
> En Visual Basic, las funciones (que devuelven un valor) empiezan con `function`, y las subrutinas (que no devuelven un valor) empiezan con `sub`. En Python no hay subrutinas. Todo son funciones, todas las funciones devuelven un valor (incluso si este es `None`), y todas las funciones empiezan con `def`.

En tercer lugar, el argumento, `params`, no especifica el tipo de dato. En Python, las variables o tienen nunca un tipo explícito. Python supone de qué tipo es una variable y lo guarda internamente.

> **Nota: Python frente a Java: valores de retorno**
> En Java, C++ y otros lenguajes de tipos estáticos, se debe especificar el tipo de valor de retorno de una función y de cada argumento. En Python, nunca se especifica explícitamente el tipo de nada. Según el valor que se le asigne, Python almacena el tipo de dato internamente.

Añadido: Un erudito lector me envió esta exposición comparando Python con otros lenguajes de programación:

Lenguaje de tipos estáticos
: Un lenguaje en el que los tipos deben declararse. Java y C son lenguajes de tipos estáticos, porque cada declaración de variable debe incluir un tipo de dato.

Lenguaje de tipos dinámicos
: Un lenguaje en el que los tipos se descubren en el momento de ejecución; lo contrario de tipos estáticos. VBScript y Python son lenguajes de tipos dinámicos, porque suponen de qué tipo es una variable cuando se le asigna un valor por primera vez.

Lenguaje fuertemente tipado
: Un lenguaje en el que los tipos se mantienen siempre. Java y Python son fuertemente tipados. Si se tiene un entero, no se le puede tratar como una cadena sin convertirlo explícitamente (veremos más sobre como hacerlo en este capítulo).

Lenguaje débilmente tipado
: Un lenguaje en el que los tipos pueden ignorarse; lo contrario de fuertemente tipado. VBScript es débilmente tipado. En VBScript, se puede concatenar la cadena `'12'` con el entero `3` para obtener la cadena `'123'`, y después tratarla como el entero `123`, todo ello sin conversión explícita.

De modo que Python es un lenguaje de *tipos dinámicos* (ya que no utiliza declaraciones explícitas de tipos) y

fuertemente tipado (ya que cuando se ha dado un tipo a una variable, permanece).

1.3. Documentación de funciones

Se puede documentar una función de Python añadiéndole una cadena de documentación.

Ejemplo 1.4. Definición de la cadena de documentación de la función `buildConnectionString`

```
def buildConnectionString(params):
    """Build a connection string from a dictionary of parameters.

    Returns string."""
```

Las tres comillas indican que la cadena ocupa más de una línea. Todo lo que hay entre las comillas es parte de una única cadena, incluidos los retornos de carro y otras comillas. Se puede usar en cualquier punto, pero se ve con más frecuencia al definir una cadena de documentación

> **Nota: Python frente a Perl: entrecomillados**
> Las triples comillas son también una forma fácil de definir una cadena que incluya comillas simples y dobles, como `qq/.../` en Perl.

Todo lo que hay entre las comillas es la cadena de documentación de la función, que explica lo que hace ésta. Una cadena de documentación, si existe, debe ser lo primero que se define en la función (es decir, lo primero que aparece tras los dos puntos). No es técnicamente necesario incluir una cadena de documentación, pero es recomendable hacerlo siempre. Esto lo habrá oído usted en todas sus clases de programación, pero Python añade un incentivo: la cadena de documentación está disponible en tiempo de ejecución como atributo de la función.

> **Nota: Por qué son buenas las cadenas de documentación**
> Muchos entornos de programación de Python utilizan la cadena de documentación para proporcionar ayuda sensible al contexto, de modo que cuando se escribe el nombre de una función, su cadena de documentación se muestra como ayuda. Esto puede ser increíblemente útil, pero sólo es tan bueno como lo sean las cadenas de documentación que se escriban.

Lecturas complementarias

- La *Guía de estilo de Python* explica cómo escribir una buena cadena de documentación.
- El *Tutorial de Python* explica las convenciones de uso del espaciado en las cadenas de documentación.

1.4. Todo es un objeto

Por si no se ha dado cuenta, acabo de decir que las funciones en Python tiene atributos, y que esos atributos están disponibles en tiempo de ejecución.

Una función, como cualquier otra cosa en Python, es un objeto.

Ejemplo 1.5. Acceso a la cadena de documentación de `buildConnectionString`

```
>>> import odbchelper                                (1)
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa", "pwd":"secret"}
>>> print odbchelper.buildConnectionString(params) (2)
server=mpilgrim;uid=sa;database=master;pwd=secret
```

```
>>> print odbchelper.buildConnectionString.__doc__ (3)
Build a connection string from a dictionary

Returns string.
```

(1) La primera línea importa el programa `odbchelper` como módulo. Cuando se ha importado un módulo, se puede hacer referencia a cualquiera de sus funciones, clases o atributos públicos. Los módulos pueden hacer esto para acceder a la funcionalidad de otros módulos, y se puede hacer también en el entorno de programación. Esto es un concepto importante, y volveremos sobre ello más adelante.

(2) Cuando se quieren utilizar funciones definidas en los módulos importados, debe incluirse el nombre del módulo. No puede decirse sólo `buildConnectionString`, debe ser `odbchelper.buildConnectionString`. Si ha utilizado usted clases en Java, esto le sonará vagamente familiar.

(3) En lugar de llamar a la función como podría esperarse, pedimos uno de los atributos de la función, `__doc__`

> **Nota: Python frente a Perl: importar**
> `import` en Python es como `require` en Perl. Cuando se ha importado un módulo en Python, se puede acceder a sus funciones con *`módulo.función`*; cuando se ha requerido un módulo en Perl, se puede acceder a sus funciones con *`módulo::función`*.

Todo en Python es un objeto, y casi todo tiene atributos y métodos.[1] Todas las funciones tienen un atributo `__doc__` incorporado, que devuelve la cadena de documentación definida en el código fuente de la función.

Esto es tan importante que voy a repetirlo por si no se ha dado cuenta las otras veces: *todo en Python es un objeto*. Las cadenas son objetos. Las listas son objetos. Las funciones son objetos. Incluso los módulos son objetos, como veremos en seguida.

Lecturas complementarias

- La *Referencia del lenguaje Python* explica exactamente qué significa la afirmación de que todo en Python es un objeto.
- eff−bot resume los objetos de Python.

1.5. Sangrado del código

Las funciones de Python no incluyen explícitamente `begin` o `end`, ni llaves que marquen dónde comienza o dónde acaba la función. El único delimitador son los dos puntos (`:`) y el propio sangrado del código.

Ejemplo 1.6. Sangrado de la función `buildConnectionString`

```
def buildConnectionString(params):
    """Build a connection string from a dictionary of parameters.

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

Los bloques de código (funciones, sentencias `if`, bucles `for`, etc.) se definen por el sangrado. El sangrado comienza un bloque y el sangrado inverso lo termina. No hay llaves, corchetes ni palabras clave explícitas. Esto significa 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 cuatro espacios. No tienen por qué ser cuatro, simplemente debe ser consistente. La primera línea que no está sangrada está fuera de la función.

Tras algunas protestas iniciales y algunas comparaciones despectivas con Fortran, se reconciliará usted con este sistema y comenzará a ver sus beneficios. Uno de los principales es que todos los programas en Python tienen un aspecto similar, ya que el sangrado es un requisito del lenguaje y no un asunto de estilo. Esto hace más fácil leer y comprender el código en Python de otras personas.

> **Nota: Python frente a Java: separación de sentencias**
> Python usa el retorno de carro para separar sentencias y los dos puntos y el sangrado para separar bloques de código. C++ y Java usan el punto y coma para separar sentencias y las llaves para separar bloques de código.

Lecturas complementarias

- La *Referencia del lenguaje Python* plantea varias cuestiones sobre el sangrado en diferentes plataformas y muestra algunos errores de sangrado.
- La *Guía de estilo de Python* comenta el buen estilo de sangrado.

1.6. Prueba de módulos

Los módulos en Python son objetos y tienen varios atributos útiles. Estos pueden utilizarse para probar los módulos mientras se escriben.

Ejemplo 1.7. El truco `if __name__`

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

Unas observaciones breves antes de meternos en materia. En primer lugar, no son necesarios los paréntesis en la condición del `if`. En segundo lugar, la sentencia `if` termina con dos puntos, y le sigue código sangrado.

> **Nota: Python frente a C: comparación y asignación**
> Al igual que C, Python usa `==` para comparar y `=` para asignar. Pero a diferencia de C, Python no admite la asignación en línea, de modo que no hay posibilidad de asignar accidentalmente el valor con el que se piensa que se está comparando.

¿Por qué esta sentencia `if` en concreto es un truco? Los módulos son objetos, y todos los módulos tienen un atributo `__name__` incorporado. El atributo `__name__` de un módulo depende de cómo se utiliza el módulo. Si se importa con import, `__name__` es el nombre del fichero del módulo, sin la ruta al directorio ni la extensión. Pero también puede ejecutarse un módulo directamente como programa independiente, y en este caso `__name__` adquiere un valor especial, `__main__`.

Ejemplo 1.8. `__name__` en un módulo importado

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

Sabiendo esto, se puede diseñar una serie de pruebas para un módulo dentro del propio módulo, incluyéndola en esta sentencia `if`. Cuando se ejecuta el módulo directamente, `__name__` es `__main__`, por lo que la serie de pruebas se ejecuta. Cuando se importa el módulo, `__name__` es otra cosa, por lo que la serie de pruebas se ignora. Esto hace más fácil desarrollar y depurar nuevos módulos antes de integrarlos en un programa mayor.

> **Sugerencia: `if __name__` en Mac OS**
> En MacPython, hay que añadir un paso más para que funcione el truco `if __name__`. Mostrar el

menú de opciones del módulo pulsando en el triángulo negro de la esquina superior derecha de la ventana, y asegurarse de que Run as __main__ está seleccionado.

Lecturas complementarias

- La *Referencia del lenguaje Python* expone detalles de bajo nivel sobre la importación de módulos.

1.7. Todo sobre los diccionarios

A continuación una breve digresión para conocer los diccionarios, las tuplas y las listas (¡oh, no!). Si es usted un hacker de Perl, probablemente pueda saltarse la parte sobre diccionarios y listas, pero debería prestar atención a las tuplas.

Uno de los tipos de datos incorporados en Python es el diccionario, que define una relación uno a uno entre claves y valores.

> **Nota: Python frente a Perl: diccionarios**
> Un diccionario en Python es como un hash en Perl. En Perl, las variables que almacenan hashes comienzan siempre por el carácter %; en Python, las variables pueden tener cualquier nombre, y Python guarda internamente el tipo de dato.

> **Nota: Python frente a Java: diccionarios**
> Un diccionario en Python es como una instancia de la clase `Hashtable` en Java.

> **Nota: Python frente a Visual Basic: diccionarios**
> Un diccionario en Python es como una instancia del objeto `Scripting.Dictionary` en Visual Basic.

Ejemplo 1.9. Definición de un diccionario

```
>>> d = {"server":"mpilgrim", "database":"master"} (1)
>>> d
{'server': 'mpilgrim', 'database': 'master'}
>>> d["server"]                                    (2)
'mpilgrim'
>>> d["database"]                                  (3)
'master'
>>> d["mpilgrim"]                                  (4)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
KeyError: mpilgrim
```

(1) En primer lugar, creamos un diccionario nuevo con dos elementos y lo asignamos a la variable `d`. Cada elemento es un par clave−valor, y el conjunto total de elementos se encierra entre llaves.

(2) `server` es una clave, y su valor asociado, al que se hace referencia con `d["server"]`, es `mpilgrim`.

(3) `database` es una clave, y su valor asociado, al que se hace referencia con `d["database"]`, es `master`.

(4) Se pueden obtener los valores por su clave, pero no se pueden obtener las claves por el valor. Así, `d["server"]` es `mpilgrim`, pero `d["mpilgrim"]` lanza una excepción, porque `mpilgrim` no es una clave.

Ejemplo 1.10. Modificación de un diccionario

```
>>> d
{'server': 'mpilgrim', 'database': 'master'}
>>> d["database"] = "pubs" (1)
>>> d
{'server': 'mpilgrim', 'database': 'pubs'}
>>> d["uid"] = "sa"        (2)
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'pubs'}
```

(1) No puede haber claves duplicadas en un diccionario. La asignación de un valor a una clave existente borrará el valor anterior.

(2) Se pueden añadir nuevos pares clave−valor en cualquier momento. Esta sintaxis es idéntica a la de la modificación de valores existentes. (Sí, esto le molestará algún día, cuando pensando que está añadiendo nuevos valores esté realmente modificando el mismo valor una y otra vez porque la clave no cambia de la manera que usted piensa.)

Advierta que el nuevo elemento (clave `uid`, valor `sa`) aparece en medio. De hecho, es sólo una coincidencia que los elementos aparecieran en orden en el primer ejemplo; es también una coincidencia que aparezcan desordenados ahora.

> **Nota: Los diccionarios no tienen orden**
> Los diccionarios no tienen concepto alguno de orden entre sus elementos. Es incorrecto decir que los elementos están "desordenados"; simplemente no hay orden. Esta distinción es importante, y le estorbará cuando intente acceder a los elementos de un diccionario en un orden específico y repetible (como el orden alfabético de las claves). Hay formas de hacer esto, sólo que no forman parte de los diccionarios.

Ejemplo 1.11. Mezcla de tipos de datos en un diccionario

```
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'pubs'}
>>> d["retrycount"] = 3 (1)
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master', 'retrycount': 3}
>>> d[42] = "douglas"   (2)
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master', 42: 'douglas', 'retrycount': 3}
```

(1) Los diccionarios no sirven solamente para almacenar cadenas. Los valores de un diccionario pueden ser de cualquier tipo, incluidas cadenas, enteros, objetos o incluso otros diccionarios. Y en un mismo diccionario, los valores no tienen que ser todos del mismo tipo: se pueden mezclar y emparejar como sea necesario.

(2) Las claves de un diccionario están más restringidas, pero pueden ser cadenas, enteros, y algunos tipos más (lo veremos más adelante). También se pueden mezclar y emparejar distintos tipos de claves en un diccionario.

Ejemplo 1.12. Eliminación de elementos de un diccionario

```
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master', 42: 'douglas', 'retrycount': 3}
>>> del d[42] (1)
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master', 'retrycount': 3}
>>> d.clear() (2)
>>> d
{}
```

(1) `del` permite borrar elementos individuales de un diccionario por su clave.

(2) `clear` borra todos los elementos de un diccionario. Advierta que el conjunto de las dos llaves vacías indica un diccionario sin elementos.

Ejemplo 1.13. Las cadenas diferencian mayúsculas y minúsculas

```
>>> d = {}
>>> d["key"] = "value"
>>> d["key"] = "other value" (1)
>>> d
{'key': 'other value'}
>>> d["Key"] = "third value" (2)
>>> d
{'Key': 'third value', 'key': 'other value'}
```

(1) Al asignar un valor a una clave existente en un diccionario, simplemente se sustituye el valor antiguo con el nuevo.

(2) Aquí no se asigna un valor a una clave existente en el diccionario, porque en las cadenas de Python se diferencia entre mayúsculas y minúsculas, de modo que `'key'` no es lo mismo que `'Key'`. Aquí se crea un nuevo par clave−valor en el diccionario; puede parecer lo mismo, pero tal como lo ve Python, es completamente distinto.

Lecturas complementarias

- *How to Think Like a Computer Scientist* trata sobre diccionarios y muestra cómo usar diccionarios para modelar matrices dispersas.
- Python Knowledge Base tiene muchos ejemplos de código que utiliza diccionarios.
- Python Cookbook expone cómo ordenar los valores de un diccionario según su clave.
- La *Referencia de bibliotecas de Python* resume los métodos de los diccionarios.

1.8. Todo sobre las listas

Las listas son el burro de carga en Python. Si si única experiencia con listas son los arrays de Visual Basic o (Dios lo prohiba) los datastore en Powerbuilder, prepárese para ver las listas de Python.

> **Nota: Python frente a Perl: listas**
> Una lista en Python es como un array en Perl. En Perl, las variables que almacenan arrays comienzan siempre con el carácter `@`; en Python, las variables pueden llamarse de cualquier modo, y Python lleva el registro del tipo de datos internamente.

> **Nota: Python frente a Java: listas**
> Una lista en Python es muy parecida a un array en Java (aunque se puede usar de ese modo si eso es todo lo que se espera en la vida). Se puede comparar mejor con la clase `Vector`, que puede guardar objetos arbitrarios y expandirse dinámicamente al añadir nuevos elementos.

Ejemplo 1.14. Definición de una lista

```
>>> li = ["a", "b", "mpilgrim", "z", "example"] (1)
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[0]                                       (2)
'a'
>>> li[4]                                       (3)
```

```
'example'
```

(1) En primer lugar, definimos una lista de cinco elementos. Advierta que mantienen el orden original. Esto no es por azar. Una lista es un conjunto ordenado de elementos encerrados entre corchetes.

(2) Una lista puede utilizarse como un array con primer índice 0. El primer elemento de una lista no vacía es siempre `li[0]`.

(3) El último elemento en esta lista de cinco es `li[4]`, porque las listas siempre tienen como primer índice 0.

Ejemplo 1.15. Índices negativos en una lista

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[-1] (1)
'example'
>>> li[-3] (2)
'mpilgrim'
```

(1) Un índice negativo da acceso a los elementos del final de la lista, contando hacia atrás. El último elemento de cualquier lista no vacía es siempre `li[-1]`.

(2) Si los índices negativos le resultan confusos, piénselo de este modo: `li[n] == li[n - len(li)]`. De modo que en esta lista, `li[2] == li[2 - 5] == li[-3]`.

Ejemplo 1.16. Porciones de una lista

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[1:3]  (1)
['b', 'mpilgrim']
>>> li[1:-1] (2)
['b', 'mpilgrim', 'z']
>>> li[0:3]  (3)
['a', 'b', 'mpilgrim']
```

(1) Se puede obtener un subconjunto de una lista, llamado "porción", especificando dos índices. El valor de retorno es una nueva lista que contiene todos los elementos de la lista original, en orden, empezando con el primer índice de la porción (en este caso `li[1]`), hasta el segundo índice de la porción (en este caso `li[3]`), pero sin incluir éste.

(2) Las porciones funcionan si uno o ambos índices son negativos. Si le ayuda, el primer índice de la porción especifica el primer elemento que se desea obtener, y el segundo índice especifica el primer elemento que no se desea obtener. El valor de retorno es lo que hay entre medias.

(3) Las listas comienzan en el índice 0, de manera que `li[0:3]` devuelve los tres primeros elementos de la lista, desde `li[0]` hasta `li[3]`, sin incluir éste.

Ejemplo 1.17. Atajos para hacer porciones

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[:3] (1)
['a', 'b', 'mpilgrim']
>>> li[3:] (2)
['z', 'example']
>>> li[:]  (3)
['a', 'b', 'mpilgrim', 'z', 'example']
```

(1) Si uno de los índices es 0, puede omitirse, y se sobreentiende. De modo que `li[:3]` es lo mismo que li[0:3] en el ejemplo anterior.

(2) Advierta aquí la simetría. En esta lista de cinco elementos, `li[:3]` devuelve los tres primeros elementos, y `li[3:]` devuelve los dos últimos. De hecho, `li[:n]` devolverá siempre los `n` primeros elementos, y `li[n:]` devolverá el resto.

(3) Si ambos índices se omiten, se incluirán todos los elementos de la lista. Pero esta lista no es la original; es una nueva lista que resulta tener los mismos elementos. `li[:]` es una forma abreviada para hacer una copia completa de una lista.

Ejemplo 1.18. Adición de elementos a una lista

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li.append("new")               (1)
>>> li
['a', 'b', 'mpilgrim', 'z', 'example', 'new']
>>> li.insert(2, "new")            (2)
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new']
>>> li.extend(["two", "elements"]) (3)
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new', 'two', 'elements']
```

(1) `append` añade un elemento al final de la lista.

(2) `insert` inserta un elemento en una lista. El argumento numérico es el índice del primer elemento que será desplazado de su posición. Advierta que los elementos de la lista no tienen que ser únicos; ahora hay dos elementos separados con el mismo valor, `li[2]` y `li[6]`.

(3) `extend` concatena listas. Advierta que no se llama a `extend` con varios argumentos; debe llamarse con un único argumento: una lista. En este caso, la lista tiene dos elementos.

Ejemplo 1.19. Búsqueda en una lista

```
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new', 'two', 'elements']
>>> li.index("example") (1)
5
>>> li.index("new")     (2)
2
>>> li.index("c")       (3)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: list.index(x): x not in list
>>> "c" in li           (4)
0
```

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

(2) `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 en `li[6]`, pero `index` devuelve sólo el primer índice, `2`.

(3) Si el valor no se encuentra en la lista, Python lanza una excepción. Esto es notablemente distinto de la mayoría de los lenguajes, que devolverán un índice no válido. Aunque esto parezca molesto, es bueno, ya que significa que su programa terminará en el lugar donde se origina el problema, y no más adelante cuando se intente utilizar el índice no válido.

(4) Para comprobar si un valor está en la lista, utilice `in`, que devuelve `1` si se encuentra el valor y `0` si no.

Nota: ¿Qué es verdadero en Python?
No hay un tipo booleano en Python. En un contexto booleano (como una sentencia `if`), `0` es falso y el resto de los números son verdaderos. Esto se extiende también a otros tipos de datos. Una cadena vacía (`""`), una lista vacía (`[]`) y un diccionario vacío (`{}`) son todos falsos; el resto de cadenas, listas y diccionarios son verdaderos.

Ejemplo 1.20. Eliminación de elementos de una lista

```
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new', 'two', 'elements']
>>> li.remove("z")   (1)
>>> li
['a', 'b', 'new', 'mpilgrim', 'example', 'new', 'two', 'elements']
>>> li.remove("new") (2)
>>> li
['a', 'b', 'mpilgrim', 'example', 'new', 'two', 'elements']
>>> li.remove("c")   (3)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: list.remove(x): x not in list
>>> li.pop()         (4)
'elements'
>>> li
['a', 'b', 'mpilgrim', 'example', 'new', 'two']
```

(1) `remove` elimina la primera aparición de un valor en una lista.

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

(3) Si el valor no se encuentra en la lista, Python lanza una excepción. Esto imita el comportamiento del método `index`.

(4) `pop` es un animal curioso. Hace dos cosas: elimina el último elemento de la lista, y devuelve el valor que ha eliminado. Advierta que esto es diferente de `li[-1]`, que devuelve el valor sin modificar la lista, y de `li.remove(`*`valor`*`)`, que modifica la lista sin devolver ningún valor.

Ejemplo 1.21. Operadores de lista

```
>>> li = ['a', 'b', 'mpilgrim']
>>> li = li + ['example', 'new'] (1)
>>> li
['a', 'b', 'mpilgrim', 'example', 'new']
>>> li += ['two']               (2)
>>> li
['a', 'b', 'mpilgrim', 'example', 'new', 'two']
>>> li = [1, 2] * 3             (3)
>>> li
[1, 2, 1, 2, 1, 2]
```

(1) Las listas se pueden concatenar también con el operador `+`. *`lista = lista + otralista`* equivale a *`lista`*`.extend(`*`otralista`*`)` . Pero el operador `+` devuelve la nueva lista como valor, mientras que `extend` sólo modifica una lista existente.

(2) Python admite el operador `+=`. `li += ['two']` equivale a `li = li + ['two']`. El operador `+=` funciona con listas, cadenas y enteros, y puede sobrecargarse para funcionar con clases definidas por el usuario. (Veremos más sobre clases en el capítulo 3).

(3)

El operador * funciona con listas como repetidor. `li = [1, 2] * 3` equivale a `li = [1, 2] + [1, 2] + [1, 2]`, que concatena las tres listas en una.

Lecturas complementarias

- *How to Think Like a Computer Scientist* trata sobre listas y pone énfasis en el paso de listas como argumentos de función.
- El *Tutorial de Python* muestra cómo utilizar listas como pilas y colas.
- Python Knowledge Base responde algunas preguntas comunes sobre listas y tiene muchos ejemplos de código que utiliza listas..
- La *Referencia de bibliotecas de Python* resume todos los métodos de lista.

1.9. Todo sobre las tuplas

Una tupla es una lista inmutable. Una tupla no puede modificarse de ningún modo después de su creación.

Ejemplo 1.22. Defining a tuple

```
>>> t = ("a", "b", "mpilgrim", "z", "example") (1)
>>> t
('a', 'b', 'mpilgrim', 'z', 'example')
>>> t[0]                                       (2)
'a'
>>> t[-1]                                      (3)
'example'
>>> t[1:3]                                     (4)
('b', 'mpilgrim')
```

(1) Una tupla se define del mismo modo que una lista, salvo que el conjunto se encierra entre paréntesis en lugar de entre corchetes.

(2) Los elementos de una tupla tienen un orden definido, como los de una lista. Las tuplas tienen primer índice 0, como las listas, de modo que el primer elemento de una tupla no vacía es siempre `t[0]`,

(3) Los índices negativos cuentan desde el final de la tupla, como en las listas.

(4) Las porciones funcionan como en las listas. Advierta que al extraer una porción de una lista, se obtiene una lista nueva; al extraerla de una tupla, se obtiene una tupla nueva.

Ejemplo 1.23. Las tuplas no tienen métodos

```
>>> t
('a', 'b', 'mpilgrim', 'z', 'example')
>>> t.append("new")    (1)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'append'
>>> t.remove("z")      (2)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'remove'
>>> t.index("example") (3)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'index'
>>> "z" in t           (4)
1
```

(1) No pueden añadirse elementos a una tupla. Las tuplas no tienen los métodos `append` ni `extend`.

(2) No pueden eliminarse elementos de una tupla. Las tuplas no tienen los métodos `remove` ni `pop`.

(3) No pueden buscarse elementos en una tupla. Las tuplas no tienen el método `index`.

(4) Se puede, no obstante, usar `in` para ver si un elemento existe en la tupla.

Entonces, ¿para qué sirven las tuplas?

- Las tuplas son más rápidas que las listas. Si está usted definiendo un conjunto constante de valores y todo lo que va a hacer con él es recorrerla, utilice una tupla en lugar de una lista.
- ¿Recuerda que dije que las claves de un diccionario pueden ser enteros, cadenas y "algunos otros tipos"? Las tuplas son uno de estos tipos. Las tuplas pueden utilizarse como claves en un diccionario, pero las listas no.[2]
- Las tuplas se utilizan para formatear cadenas, como veremos en seguida.

> **Nota: De tuplas a listas y a tuplas**
> Las tuplas pueden convertirse 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 la práctica, `tuple` congela una lista, y `list` descongela una tupla.

Lecturas complementarias

- *How to Think Like a Computer Scientist* trata de tuplas y muestra cómo concatenar tuplas.
- Python Knowledge Base muestra cómo ordenar una tupla.
- El *Tutorial de Python* muestra cómodefinir una tupla con un solo elemento.

1.10. Definición de variables

Ahora que piensa que ya lo sabe todo sobre diccionarios, tuplas y listas (¡oh, no!), volvamos a nuestro programa de ejemplo, `odbchelper.py`.

Python tiene variables locales y globales como la mayoría de los lenguajes, pero no tiene declaraciones explícitas de variables. Las variables aparecen al asignarles un valor, y son automáticamente destruidas cuando salimos de su ámbito.

Ejemplo 1.24. Definición de la variable `myParams`

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

Aquí hay varios puntos de interés. En primer lugar, fíjese en el sangrado. Una sentencia `if` es un bloque de código y necesita estar sangrado, como una función.

En segundo lugar, la asignación de la variable es una orden partida en varias líneas, con una contrabarra ("\") como marca de continuación de la línea.

> **Nota: Órdenes de varias líneas**
> Cuando una orden se extiende a lo largo de varias líneas con la marca de continuación de línea ("\"), las líneas que siguen pueden sangrarse de cualquier modo; las severas normas de sangradeo de Python no se aplican. Si entorno de programación Python sangra automáticamente las líneas que continúan,

debe usted aceptar probablemente el valor por omisión a no ser que tenga una buena razón.

Nota: Órdenes de varias líneas implícitas
Para ser exactos, las expresiones entre paréntesis, corchetes o llaves (como la definición de un diccionario) pueden extenderse a varias líneas con la marca de continuación ("\") o sin ella. Yo prefiero incluir la contrabarra incluso cuando no es imprescindible, porque creo que hace el código más legible, pero esto es cuestión de estilo.

En tercer lugar, nunca se ha declarado la variable `myParams`, simplemente se le ha asignado un valor. Esto es como VBScript sin la opción `option explicit`. Afortunadamente, a diferencia de VBScript, Python no permite hacer referencia a una variable a la que no se ha asignado un valor; si se intenta, se lanzará una excepción.

Ejemplo 1.25. Referencia a una variable no asignada

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

Algún día agradecerá esto

Lecturas complementarias

- La *Referencia del lenguaje Python* muestra ejemplos de cuándo se puede omitir la marca de continuación de línea y cuándo debe usarse.

1.11. Asignación de múltiples valores de una vez

Uno de los mejores atajos de programación en Python es el uso de secuencias para asignar múltiples valores de una vez.

Ejemplo 1.26. Asignación de múltiples valores de una vez

```
>>> v = ('a', 'b', 'e')
>>> (x, y, z) = v (1)
>>> x
'a'
>>> y
'b'
>>> z
'e'
```

(1) `v` es una tupla con tres elementos, y `(x, y, z)` es una tupla con tres variables. Al asignar una a la otra se asigna cada uno de los valores de `v` a cada una de las variables, en orden.

Esto tiene varios usos. Con frecuencia, necesito asignar nombres a un rango de valores. En C, se utilizaría `enum` y se haría una lista a mano de cada constante y su valor asociado, lo cual resulta especialmente tedioso cuando los valores son consecutivos. En Python, puede utilizarse la función incorporada `range` con asignación a múltiples variables para asignar rápidamente valores consecutivos.

Ejemplo 1.27. Asignación de valores consecutivos

```
>>> range(7)                                                              (1)
[0, 1, 2, 3, 4, 5, 6]
>>> (MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY) = range(7) (2)
>>> MONDAY                                                                (3)
0
>>> TUESDAY
1
>>> SUNDAY
6
```

(1) La función incorporada `range` devuelve una lista de enteros. En su forma más simple, toma un límite superior y devuelve una lista que comienza en `0` y sigue hasta el límite superior, sin incluirlo. (Si le parece, puede pasar otros parámetros para especificar un origen distinto de `0` y un incremento distinto de `1`. Puede consultar `range.__doc__` para ver los detalles.)

(2) `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY` y `SUNDAY` son las variables que vamos a definir. (Este ejemplo proviene del módulo `calendar`, un pequeño y divertido módulo que imprime calendarios, como el programa `cal` de UNIX. El módulo `calendar` define constantes enteras para los días de la semana.)

(3) Ahora cada variable tiene su valor: `MONDAY` es `0`, `TUESDAY` es `1`, y así sucesivamente.

También puede utilizarse la asignación múltiple para construir funciones que devuelvan valores múltiples, simplemente devolviendo una tupla con todos los valores. Tras la llamada, el valor devuelto puede tratarse como una tupla, o asignar los valores a variables individuales. Muchas bibliotecas estándar de Python hacen esto, incluido el módulo `os`, que veremos en el capítulo 3.

Lecturas complementarias

- *How to Think Like a Computer Scientist* muestra cómo usar la asignación múltiple para intercambiar los valores de dos variables.

1.12. Formato de cadenas

Python acepta el formato de valores como cadenas. Aunque esto puede incluir expresiones muy complicadas, el uso más básico es la inserción de valores eun una cadena con la plantilla `%s`.

> **Nota: Python frente a C: formato de cadenas**
> El formato de cadenas en Python utiliza la misma sintaxis que la función `sprintf` en C.

Ejemplo 1.28. Presentación del formato de cadenas

```
>>> k = "uid"
>>> v = "sa"
>>> "%s=%s" % (k, v) (1)
'uid=sa'
```

(1) Toda la expresión se evalúa como una cadena. El primer `%s` se sustituye con el valor de `k`; el segundo `%s` con el valor de `v`. Todos los demás caracteres de la cadena (en este caso, el signo igual) permanecen sin cambios.

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

Puede pensar usted que esto es mucho trabajo sólo para una simple concatenación de cadenas, y quizá tenga razón, salvo en que el formato de cadenas no es sólo una concatenación. Ni siquiera es sólo formato. También es adaptación de tipos.

Ejemplo 1.29. Formato de cadenas frente a concatenación

```
>>> uid = "sa"
>>> pwd = "secret"
>>> print pwd + " is not a good password for " + uid      (1)
secret is not a good password for sa
>>> print "%s is not a good password for %s" % (pwd, uid) (2)
secret is not a good password for sa
>>> userCount = 6
>>> print "Users connected: %d" % (userCount, )           (3) (4)
Users connected: 6
>>> print "Users connected: " + userCount                 (5)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
TypeError: cannot add type "int" to string
```

(1) `+` es el operador de concatenación de cadenas.

(2) En este caso sencillo, el formato de cadenas produce el mismo resultado que la concatenación.

(3) `(userCount, )` es una tupla con un solo elemento. Sí, la sintaxis es algo extraña, pero hay una buena razón para ello: es, sin ambigüedad, una tupla. De hecho, siempre se puede incluir una coma tras el último elemento al definir una lista, tupla o diccionario; pero es imprescindible cuando se define una tupla con un solo elemento. Si no hiciera falta la coma, Python no sabría si `(userCount)` era una tupla con un elemento o sólo el valor de `userCount`.

(4) El formato de cadenas funciona con enteros especificando `%d` en lugar de `%s`.

(5) Si se intenta concatenar una cadena con algo diferente, se lanza una excepción. A diferencia del formato de cadenas, la concatenación de cadenas sólo funciona cuando todo son cadenas.

Lecturas complementarias

- La *Referencia de bibliotecas de Python* resume todos los caracteres de formato de cadenas.
- *Effective AWK Programming* expone todos los caracteres de formato y las técnicas avanzadas de formato de cadenas, como la indicación de ancho, la precisión y el relleno con ceros.

1.13. Correspondencia de listas

Una de las más potentes características de Python es la creación de listas por comprensión, que proporciona una forma compacta de relacionar una lista con otra aplicando una función a los elementos de la primera.

Ejemplo 1.30. Introducción a las listas por comprensión

```
>>> li = [1, 9, 8, 4]
>>> [elem*2 for elem in li]      (1)
[2, 18, 16, 8]
>>> li                           (2)
[1, 9, 8, 4]
>>> li = [elem*2 for elem in li] (3)
>>> li
[2, 18, 16, 8]
```

(1) Para hacerse una idea de esto, mírelo de derecha a izquierda. `li` es la lista a la que se aplica la relación. Python recorre esta lista elemento por elemento, asignando temporalmente el valor de cada uno de ellos a la variable `elem`. A continuación, Python aplica la función `elem*2` y añade el resultado a la lista que se devuelve.

(2) Advierta que al crear una lista por comprensión no se modifica la lista original.

(3) No hay riesgo en asignar el resultado de una lista creada por comprensión a la variable a que se aplica la relación. No hay nada de qué preocuparse: Python construye la nueva lista en memoria, y cuando la operación ha terminado, se asigna el resultado a la variable.

Ejemplo 1.31. Listas por comprensión en `buildConnectionString`

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

En primer lugar, advierta que se está llamando a la función `items` del diccionario `params`. Esta función devuelve una lista de tuplas con todos los datos del diccionario.

Ejemplo 1.32. `keys`, `values`, e `items`

```
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa", "pwd":"secret"}
>>> params.keys()   (1)
['server', 'uid', 'database', 'pwd']
>>> params.values() (2)
['mpilgrim', 'master', 'sa', 'secret']
>>> params.items()  (3)
[('server', 'mpilgrim'), ('uid', 'sa'), ('database', 'master'), ('pwd', 'secret')]
```

(1) 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, los elementos de un diccionario no guardan orden), pero es una lista.

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

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

Veamos ahora qué hace `buildConnectionString`. Toma una lista, `params.items()`, y la transforma en una nueva lista aplicando el formato de cadenas a cada elemento. La nueva lista tendrá el mismo número de elementos que `params.items()`, pero cada elemento de la nueva lista será una cadena que contiene tanto la clave como su valor asociado en el diccionario `params`.

Ejemplo 1.33. 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()]               (1)
['server', 'uid', 'database', 'pwd']
>>> [v for k, v in params.items()]               (2)
['mpilgrim', 'sa', 'master', 'secret']
>>> ["%s=%s" % (k, v) for k, v in params.items()] (3)
['server=mpilgrim', 'uid=sa', 'database=master', 'pwd=secret']
```

(1) Advierta que estamos usando dos variables para recorrer la lista `params.items()`. Este es otro uso de la asignación múltiple. El primer elemento de `params.items()` es `('server', 'mpilgrim')`, de modo que en el primer ciclo de la relación, `k` tomará el valor `'server'` y `v` el valor `'mpilgrim'`. En este caso

ignoramos el valor de `v` y sólo incluimos el de `k` en la lista devuelta, por lo que esta relación equivale finalmente a `params.keys()`. (No se utilizaría realmente una relación de listas como esta en el código real; es sólo un ejemplo demasiado simple para entender qué pasa aquí).

(2) Aquí hacemos lo mismo, pero ignoramos el valor de `k`, de manera que esta relación equivale finalmente a `params.values()`.

(3) Combinando los dos ejemplos anteriores con un simple formato de cadenas, obtenemos una lista de cadenas que incluyen tanto la clave como el valor de cada elemento del diccionario. Esto se parece sospechosamente a la salida del programa; todo lo que queda por hacer es unir los elementos de esta lista en una única cadena.

Lecturas complementarias

- El *Tutorial de Python* expone otra forma de relacionar listas usando la función incorporada `map`.
- El *Tutorial de Python* muestra cómoanidar relaciones de listas.

1.14. Unión y división de cadenas

Usted tiene una lista de pares clave−valor en la forma *`clave=valor`*, y quiere unirlos en una única cadena. Para unir cualquier lista de cadenas en una única cadena, utilice el método `join` de un objeto cadena.

Ejemplo 1.34. Unión de una lista en `buildConnectionString`

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

Una advertencia interesante antes de continuar. He repetido que las funciones son objetos, las cadenas son objetos, todo es un objeto. Podría usted pensar que quiero decir que las *variables* de cadena son objetos. Pero no, observe atentamente este ejemplo y verá que la cadena `";"` es un objeto, y que se está llamando a su método `join`.

En todo caso, el método `join` une los elementos de la lista para formar una única cadena, con cada elemento separado por un punto y coma. El delimitador no tiene que ser un punto y coma; no tiene que ser siquiere un único carácter. Puede ser cualquier cadena.

> **Importante: No se pueden unir más que cadenas**
> `join` sólo funciona con listas de cadenas; no hace ninguna conversión de tipos. Si se une una lista que tiene uno o más elementos que no sean cadenas, se lanzará una excepción.

Ejemplo 1.35. 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 es devuelta por la función `help` e impresa por el bloque que la llamó, lo que da la salida que le sorprendió cuando empezó usted a leer este capítulo.

Nota histórica. Cuando empecé a aprender Python, esperaba que `join` fuera un método de lista, que tomara el delimitador como argumento. Mucha gente piensa lo mismo, y hay toda una historia detrás del método `join`. Antes de la versión 1.6 de Python, las cadenas no tenían todos estos útiles métodos. Había un módulo separado, `string`, que contenía todas las funciones de cadena; cada función tomaba una cadena como primer argumento. Las funciones se consideraron lo suficientemente importantes como para añadirlas a las propias cadenas, lo cual tiene sentido en funciones como `lower`, `upper` y `split`. Pero muchos programadores de Python pusieron objeciones al nuevo

método `join`, argumentando que debería ser un método de lista y no de cadena, o incluso que debería seguir siendo parte del antiguo módulo `string` (que aún conserva material útil). Yo utilizo sólo el nuevo método `join`, pero puede verse código escrito en ambas formas, y si esto le resulta realmente molesto, puede usted utilizar la función `string.join` en su lugar.

Probablemente se preguntará usted si existe un método complementario para dividir una cadena en una lista. Por supuesto que existe, se llama `split`.

Ejemplo 1.36. División de 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(";")    (1)
['server=mpilgrim', 'uid=sa', 'database=master', 'pwd=secret']
>>> s.split(";", 1) (2)
['server=mpilgrim', 'uid=sa;database=master;pwd=secret']
```

(1) `split` realiza la operación inversa a `join`, dividiendo una cadena en una lista con varios elementos. Observe que el delimitador (`";"`) se elimina completamente; no aparece en ninguno de los elementos de la lista devuelta.

(2) `split` toma un segundo argumento opcional, que es el número de veces que se raliza la división. ("Oooooh, argumentos opcionales..." Aprenderá cómo hacer esto en sus propias funciones en el próximo capítulo.)

> **Nota: Buscar con `split`**
> *`cadena`*`.split(`*`delimitador`*`, 1)` es una técnica útil cuando se quiere buscar una subcadena en una cadena y después trabajar con lo que la precede (que queda en el primer elemento de la lista devuelta) y lo que la sigue (que queda en el segundo elemento).

Lecturas complementarias

- Python Knowledge Base responde preguntas comunes sobre cadenas y tiene muchos ejemplos de código que utiliza cadenas.
- La *Referencia de bibliotecas de Python* resumetodos los métodos de cadenas.
- La *Referencia de bibliotecas de Python* documenta el módulo `string`.
- *The Whole Python FAQ* explica por qué `join` es un método de cadena y no un método de lista.

1.15. Resumen

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

Ejemplo 1.37. `odbchelper.py`

```
def buildConnectionString(params):
    """Build a connection string from a dictionary of parameters.

    Returns string."""
    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)
```

Ejemplo 1.38. Salida de `odbchelper.py`

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

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

- Usar el entorno de programación Python para probar expresiones de forma interactiva
- Escribir módulos Python que puedan funcionar como programas independientes, al menos para realizar pruebas
- Importar módulos y llamar a sus funciones
- Deckarar funciones y usar sus cadenas de documentación, variables locales, y el sangrado adecuado
- Definir diccionarios, tuplas, y listas
- Acceder a los atributos y métodos de cualquier objeto, incluidas cadenas, listas, diccionarios, funciones y módulos.
- Concatenar valores con el formato de cadenas
- Crear listas a partir de otras usando relaciones
- Dividir cadenas formando listas y unir listas para formar cadenas

---

[1] Los distintos lenguajes de programación definen "objeto" de formas diferentes. En algunos, todos los objetos deben tener atributos y métodos; en otros, todos los objetos deben admitir subclases. En Python, la definición es más libre; algunos objetos no tienen ni atributos ni métodos, y no todos los objetos admiten subclases (lo veremos en el capítulo 3). Pero todo es un objeto en el sentido de que puede ser asignado a una variable o pasarse como argumento a una función (lo veremos en el capítulo 2).

[2] Realmente, es más complicado. Las claves de los diccionarios deben ser inmutables. Las tuplas son inmutables, pero si se trata de una tupla de listas, se considera mutable y no es seguro utilizarla como clave de diccionario. Sólo las tuplas de cadenas, números u otras tuplas seguras pueden utilizarse como claves en un diccionario.

Capítulo 2. El poder de la introspección

2.1. Inmersión

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 examina como objetos otros módulos y funciones en memoria, obtiene información sobre ellos y los maneja. De paso, definiremos funciones sin nombre, llamaremos a funciones con argumentos sin orden, y haremos referencia a funciones cuyos nombres desconocemos.

Aquí hay un programa Python completo y funcional. Debería usted comprenderlo sólo observándolo. Las líneas numeradas ilustran conceptos cubiertos en *Conozcamos Python*. No se preocupe si el resto del código le parece inquietante; aprenderá todo sobre él en este capítulo.

Ejemplo 2.1. `apihelper.py`

Si aún no lo ha hecho, puede descargar este y otros ejemplos de este libro.

```
def help(object, spacing=10, collapse=1): (1) (2) (3)
    """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__":               (4) (5)
    print help.__doc__
```

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

(2) La función `help` 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.

(3) El código de la función está sangrado.

(4) El truco `if __name__` permite a este programa hacer algo útil cuando se ejecuta aislado, sin que esto interfiera con su uso como módulo por otros programas. En este caso, el programa simplemente imprime la cadena de documentación de la función `help`.

(5) Las sentencias `if` usan `==` para la comparación, y no son necesarios los paréntesis.

La función `help` está diseñada para ser utilizada por usted, el programador, mientras trabaja en el entorno de programación 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 2.2. Ejemplo de uso de `apihelper.py`

```
>>> from apihelper import help
>>> li = []
>>> help(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 diez caracteres, se puede especificar un valor mayor en el argumento *`spacing`* para hacer más legible la salida.

Ejemplo 2.3. Uso avanzado de `apihelper.py`

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

    Returns string.
```

2.2. Argumentos opcionales y con nombre

Python permite que los argumentos de funciones tengan valores por omisión; si la función es llamada sin el argumento, éste toma su valor por omisión. Además, los argumentos pueden especificarse en cualquier orden si se les da 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.

Ejemplo 2.4. `help`, una función con dos argumentos opcionales

```
def help(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 `help` sólo con un argumento, `spacing` valdrá `10` y `collapse` valdrá `1`. Si se llama a `help` 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 no tendría tanta suerte, pues debería llamar a la función con los tres argumentos. Pero en Python, los argumentos pueden indicarse por su nombre en cualquier orden.

Ejemplo 2.5. Llamadas válidas a `help`

```
help(odbchelper)                    (1)
help(odbchelper, 12)                (2)
help(odbchelper, collapse=0)        (3)
help(spacing=15, object=odbchelper) (4)
```

(1)

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

(2) Con dos argumentos, `collapse` toma su valor por omisión de `1`.

(3) Aquí se nombra explícitamente el argumento `collapse` y se le da un valor. `spacing` toma su valor por omisión de `10`.

(4) 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 llamar 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.

> **Nota: Llamar a funciones es flexible**
> Lo único que debe hacerse para llamar 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

- 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.

2.3. `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).

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 2.6. Presentación de `type`

```
>>> type(1)           (1)
<type 'int'>
>>> li = []
>>> type(li)          (2)
<type 'list'>
>>> import odbchelper
>>> type(odbchelper)  (3)
<type 'module'>
>>> import types      (4)
>>> type(odbchelper) == types.ModuleType
1
```

(1) `type` toma cualquier cosa y devuelve su tipo. Y quiero decir cualquier cosa: enteros, cadenas, listas, diccionarios, tuplas, funciones, clases, módulos, incluso tipos.

(2) `type` puede tomar una variable y devolver su tipo.

(3) `type` funciona también con módulos.

(4) Pueden utilizarse las constantes del módulo `types` para comparar tipos de objetos. Esto es lo que hace la

función `help`, como veremos en seguida.

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

Ejemplo 2.7. Presentación de `str`

```
>>> str(1)          (1)
'1'
>>> horsemen = ['war', 'pestilence', 'famine']
>>> horsemen.append('Powerbuilder')
>>> str(horsemen)   (2)
"['war', 'pestilence', 'famine', 'Powerbuilder']"
>>> str(odbchelper) (3)
"<module 'odbchelper' from 'c:\\docbook\\dip\\py\\odbchelper.py'>"
>>> str(None)       (4)
'None'
```

(1) 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.

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

(3) `str` funciona también con módulos. Advierta que la representación como cadena del módulo incluye la ruta del módulo en el disco, por lo que lo que usted obtenga será diferente.

(4) 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 `help`, como veremos en seguida.

En el corazón de nuestra función `help` 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 2.8. Presentación de `dir`

```
>>> li = []
>>> dir(li)           (1)
['append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
>>> d = {}
>>> dir(d)            (2)
['clear', 'copy', 'get', 'has_key', 'items', 'keys', 'setdefault', 'update', 'values']
>>> import odbchelper
>>> dir(odbchelper)   (3)
['__builtins__', '__doc__', '__file__', '__name__', 'buildConnectionString']
```

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

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

(3) Aquí es donde empieza lo interesante. `odbchelper` es un módulo, por lo que `dir(odbchelper)` devuelve una lista con todo lo definido en el módulo, incluidos atributos incorporados, como `__name__` y `__doc__`, y cualesquiera otros atributos y métodos que se hayan definido. En este caso, `odbchelper` tiene un solo método definido, la función `buildConnectionString` que estudiamos en *Conozcamos Python*.

Finalmente, la función `callable` toma cualquier objeto y devuelve `1` si se puede llamar al objeto, o `0` en caso contrario. Los objetos que pueden ser llamados son funciones, métodos de clase o incluso las propias clases. (Más sobre clases en el capítulo 3).

Ejemplo 2.9. Presentación de `callable`

```
>>> import string
>>> string.punctuation           (1)
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
>>> string.join                  (2)
<function join at 00C55A7C>
>>> callable(string.punctuation) (3)
0
>>> callable(string.join)        (4)
1
>>> print string.join.__doc__    (5)
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)
```

(1) 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.

(2) `string.join` es una función que une una lista de cadenas.

(3) `string.punctuation` no puede llamarse; es una cadena. (Una cadena tiene métodos que pueden llamarse, pero la propia cadena no).

(4) `string.join` puede ser llamada; es una función que toma dos argumentos.

(5) Cualquier objeto que pueda llamarse 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.

`type`, `str`, `dir` y el resto de funciones incorporadas de Python se agrupan en un módulo especial llamado `__builtins__`. (Con dos subrayados antes y después). Por si sirve de ayuda, puede interpretarse que Python ejecuta automáticamente `from __builtins__ import *` al iniciarse, 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 como un grupo, obteniendo información sobre el módulo `__builtins__`. E imagíneselo, tenemos una función para ello: se llama `help`. 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 2.10. Atributos y funciones incorporados

```
>>> from apihelper import help
>>> help(__builtins__, 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...]
```

Nota: Python se autodocumenta
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, o, Dios le socorra, MSDN) para recordar cómo se usan estos módulos, Python está ampliamente autodocumentado.

Lecturas complementarias

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

2.4. Obtención de referencias a objetos con `getattr`

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 2.11. Presentación de `getattr`

```
>>> li = ["Larry", "Curly"]
>>> li.pop                      (1)
<built-in method pop of list object at 010DF884>
>>> getattr(li, "pop")          (2)
<built-in method pop of list object at 010DF884>
>>> getattr(li, "append")("Moe") (3)
>>> li
["Larry", "Curly", "Moe"]
>>> getattr({}, "clear")        (4)
<built-in method clear of dictionary object at 00F113D4>
>>> getattr((), "pop")          (5)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'pop'
```

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

(2) 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`.

(3) 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 llamado 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.

(4) `getattr` también funciona con diccionarios.

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

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

Ejemplo 2.12. `getattr` en `apihelper.py`

```
>>> import odbchelper
>>> odbchelper.buildConnectionString            (1)
```

```
<function buildConnectionString at 00D18DD4>
>>> getattr(odbchelper, "buildConnectionString") (2)
<function buildConnectionString at 00D18DD4>
>>> object = odbchelper
>>> method = "buildConnectionString"
>>> getattr(object, method)                      (3)
<function buildConnectionString at 00D18DD4>
>>> type(getattr(object, method))                (4)
<type 'function'>
>>> import types
>>> type(getattr(object, method)) == types.FunctionType
1
>>> callable(getattr(object, method))            (5)
1
```

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

(2) 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.

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

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

(5) Como `method` es una función, se puede llamar.

2.5. Filtrado de listas

Como ya sabe, Python tiene una potente capacidad para convertir una lista en otra por medio de las relaciones de listas. Esto puede combinarse con un mecanismo de filtrado en el que algunos elementos de la lista se utilicen mientras otros se pasen por alto.

Ejemplo 2.13. Sintaxis del filtrado de listas

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

Esto es una extensión de la relación de listas que usted conoce y tanto le gusta. 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 pasan por el proceso de relación y no se incluyen en la lista final.

Ejemplo 2.14. Presentación del filtrado de listas

```
>>> li = ["a", "mpilgrim", "foo", "b", "c", "b", "d", "d"]
>>> [elem for elem in li if len(elem) > 1]       (1)
['mpilgrim', 'foo']
>>> [elem for elem in li if elem != "b"]         (2)
['a', 'mpilgrim', 'foo', 'c', 'd', 'd']
>>> [elem for elem in li if li.count(elem) == 1] (3)
['a', 'mpilgrim', 'foo', 'c']
```

(1) 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.

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

(3) `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.

Ejemplo 2.15. Filtrado de una lista en `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; devuelve el valor de cada elemento. `dir(object)` devuelve la lista de los atributos y métodos de `object`; esa es la lista a 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, y 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 llamado, como lo 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

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

2.6. 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 reales que están comparando.

Ejemplo 2.16. Presentación de `and`

```
>>> 'a' and 'b'          (1)
'b'
>>> '' and 'b'           (2)
''
>>> 'a' and 'b' and 'c' (3)
'c'
```

(1) 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.[3] Si todos los valores son verdaderos en un contexto booleano, `and` devuelve el último valor. En este caso, `and` evalúa `'a'`, que es verdadera, después `'b'`, que es verdadera, y devuelve `'b'`.

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

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

Ejemplo 2.17. Presentación de `or`

```
>>> 'a' or 'b'          (1)
'a'
>>> '' or 'b'           (2)
'b'
>>> '' or [] or {}      (3)
{}
>>> def sidefx():
...     print "in sidefx()"
...     return 1
>>> 'a' or sidefx()     (4)
'a'
```

(1) 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.

(2) `or` evalúa `''`, que es falsa, después `'b'`, que es verdadera, y devuelve `'b'`.

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

(4) 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, la función `sidefx` no se llama nunca, porque `or` evalúa `'a'`, que es verdadera, y devuelve `'a'` inmediatamente.

Si es usted un hacker de C, le será familiar la expresión *`bool ? 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.

Ejemplo 2.18. Presentación del truco `and-or`

```
>>> a = "first"
>>> b = "second"
>>> 1 and a or b (1)
'first'
>>> 0 and a or b (2)
'second'
```

(1) Esta sintaxis resulta similar a la de la expresión *`bool ? 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'`.

(2) `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 *`bool ? a : b`* en C. Si el valor de `a` es falso, la expresión no funcionará como sería de esperar. (¿Puedes decir que he estado

obsesionado con esto? ¿Más de una vez?)

Ejemplo 2.19. Cuando falla el truco `and-or`

```
>>> a = ""
>>> b = "second"
>>> 1 and a or b (1)
'second'
```

(1) 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.

Importante: Uso eficaz de `and-or`
El truco `and-or`, *`bool` `and a or b`*, no funcionará como la expresión *`bool ? 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 2.20. Utilización segura del truco `and-or`

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

(1) Como `[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

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

2.7. 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 2.21. Presentación de las funciones `lambda`

```
>>> def f(x):
...     return x*2
...
>>> f(3)
6
```

```
>>> g = lambda x: x*2  (1)
>>> g(3)
6
>>> (lambda x: x*2)(3) (2)
6
```

(1) Esta 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.

(2) 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 larga como quiera.

> **Nota: Las funciones `lambda` son opcionales**
> Las funciones `lambda` son una cuestión de estilo. Su uso nunca es necesario. En cualquier lugar 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.

Ejemplo 2.22. Las funciones `lambda` en `apihelper.py`

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

Hay varias cosas que advertir de paso. En primer lugar, estamos utilizando la forma simple del truco `and-or`, lo cual es válido, porque una función `lambda` siempre es verdadera en contexto booleano. (Esto no significa que una función `lambda` no pueda devolver un valor falso. La función siempre es verdadera; su valor de retorno puede ser cualquier cosa).

En segundo lugar, estamos utilizando la función `split` sin argumentos. Ya se ha visto su uso con uno o dos argumentos, pero sin argumentos divide en los espacios en blanco.

Ejemplo 2.23. `split` sin argumentos

```
>>> s = "this   is\na\ttest"  (1)
>>> print s
this   is
a       test
>>> print s.split()           (2)
['this', 'is', 'a', 'test']
>>> print " ".join(s.split()) (3)
'this is a test'
```

(1) Esta es una cadena de varias líneas, definida por los caracteres de escape en lugar detriples comillas. `\n` es el retorno de carro; `\t` es el carácter de tabulación.

(2) `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.

(3) 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 `help` para unificar las cadenas de documentación de varias líneas en una sola línea.

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

Ejemplo 2.24. Asignación de una función a una variable

```
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`* y utilizara 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

- 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`.

2.8. 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 ya está dispuesto de la manera en que lo necesitamos. Las fichas del dominó están en su sitio; lo que queda es golpear la primera.

Ejemplo 2.25. 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 contrabarra? 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 en `object`. De modo que estamos recorriendo esta lista con la variable `method`.

Ejemplo 2.26. Obtención de una cadena de documentación de forma dinámica

```
>>> import odbchelper
>>> object = odbchelper                    (1)
>>> method = 'buildConnectionString'       (2)
>>> getattr(object, method)                (3)
<function buildConnectionString at 010D6D74>
>>> print getattr(object, method).__doc__ (4)
Build a connection string from a dictionary of parameters.

    Returns string.
```

(1) En la función `help`, `object` es el objeto sobre el que pedimos ayuda, pasado como argumento.

(2) Cuando recorremos `methodList`, `method` es el nombre del método actual.

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

(4) 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 2.27. ¿Por qué usar `str` con una cadena de documentación?

```
>>> {}.keys.__doc__          (1)
>>> {}.keys.__doc__ == None (2)
1
>>> str({}.keys.__doc__)    (3)
'None'
```

(1) La función `keys` de un diccionario no tiene cadena de documentación, luego su atributo `__doc__` es `None`. Para más confusión, si se evalúa directamente el atributo `__doc__`, el entorno de programación Python no muestra nada en absoluto, lo cual tiene sentido si se piensa bien, pero es poco práctico.

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

(3) Con el uso de la función `str` se toma el valor nulo y se devuelve su representación como cadena, `'None'`.

> **Nota: Python frente a SQL: comparación de valores nulos**
> En sql, se utiliza `IS NULL` en vez de `= NULL` para comparar un valor nulo. En Python no hay una sintaxis especial; se usa `== None` como en cualquier otra comparación.

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, vemos que estamos utilizando el formato de cadenas de nuevo para concatenar el valor de retorno de `processFunc` con el valor de retorno del método `ljust` de `method`. Este es un nuevo método de cadena que no hemos visto antes.

Ejemplo 2.28. Presentación del método `ljust`

```
>>> s = 'buildConnectionString'
>>> s.ljust(30) (1)
'buildConnectionString         '
>>> s.ljust(20) (2)
```

```
'buildConnectionString'
```

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

(2) 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 2.29. Muestra de una lista

```
>>> li = ['a', 'b', 'c']
>>> print "\n".join(li) (1)
a
b
c
```

(1) Este es también un útil truco de depuración cuando se trabaja con listas. Y en Python, siempre se trabaja con listas.

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

Ejemplo 2.30. Otra vez el meollo de `apihelper.py`

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

2.9. Resumen

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

Ejemplo 2.31. `apihelper.py`

```
def help(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 help.__doc__
```

Ejemplo 2.32. Salida de `apihelper.py`

```
>>> from apihelper import help
>>> li = []
>>> help(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 sumergirse en el siguiente capítulo, asegúrese de que se siente cómodo haciendo todo esto:

- Definir y llamar funciones con argumentos opcionales y con 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.

---

[3] 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 el capítulo 3.

Apéndice A. Lecturas complementarias

Capítulo 1. Conozcamos Python

- 1.3. Documentación de funciones
  - La *Guía de estilo de Python* explica cómo escribir una buena cadena de documentación.
  - El *Tutorial de Python* explica las convenciones de uso del espaciado en las cadenas de documentación.
- 1.4. Todo es un objeto
  - La *Referencia del lenguaje Python* explica exactamente qué significa la afirmación de que todo en Python es un objeto.
  - eff−bot resume los objetos de Python.
- 1.5. Sangrado del código
  - La *Referencia del lenguaje Python* plantea varias cuestiones sobre el sangrado en diferentes plataformas y muestra algunos errores de sangrado.
  - La *Guía de estilo de Python* comenta el buen estilo de sangrado.
- 1.6. Prueba de módulos
  - La *Referencia del lenguaje Python* expone detalles de bajo nivel sobre la importación de módulos.
- 1.7. Todo sobre los diccionarios
  - *How to Think Like a Computer Scientist* trata sobre diccionarios y muestra cómo usar diccionarios para modelar matrices dispersas.
  - Python Knowledge Base tiene muchos ejemplos de código que utiliza diccionarios.
  - Python Cookbook expone cómo ordenar los valores de un diccionario según su clave.
  - La *Referencia de bibliotecas de Python* resume los métodos de los diccionarios.
- 1.8. Todo sobre las listas
  - *How to Think Like a Computer Scientist* trata sobre listas y pone énfasis en el paso de listas como argumentos de función.
  - El *Tutorial de Python* muestra cómo utilizar listas como pilas y colas.
  - Python Knowledge Base responde algunas preguntas comunes sobre listas y tiene muchos ejemplos de código que utiliza listas..
  - La *Referencia de bibliotecas de Python* resume todos los métodos de lista.
- 1.9. Todo sobre las tuplas
  - *How to Think Like a Computer Scientist* trata de tuplas y muestra cómo concatenar tuplas.
  - Python Knowledge Base muestra cómo ordenar una tupla.
  - El *Tutorial de Python* muestra cómodefinir una tupla con un solo elemento.
- 1.10. Definición de variables

- La *Referencia del lenguaje Python* muestra ejemplos de cuándo se puede omitir la marca de continuación de línea y cuándo debe usarse.

- 1.11. Asignación de múltiples valores de una vez
  - *How to Think Like a Computer Scientist* muestra cómo usar la asignación múltiple para intercambiar los valores de dos variables.

- 1.12. Formato de cadenas
  - La *Referencia de bibliotecas de Python* resume todos los caracteres de formato de cadenas.
  - *Effective AWK Programming* expone todos los caracteres de formato y las técnicas avanzadas de formato de cadenas, como la indicación de ancho, la precisión y el relleno con ceros.

- 1.13. Correspondencia de listas
  - El *Tutorial de Python* expone otra forma de relacionar listas usando la función incorporada `map`.
  - El *Tutorial de Python* muestra cómoanidar relaciones de listas.

- 1.14. Unión y división de cadenas
  - Python Knowledge Base responde preguntas comunes sobre cadenas y tiene muchos ejemplos de código que utiliza cadenas.
  - La *Referencia de bibliotecas de Python* resumetodos los métodos de cadenas.
  - La *Referencia de bibliotecas de Python* documenta el módulo `string`.
  - *The Whole Python FAQ* explica por qué `join` es un método de cadena y no un método de lista.

Capítulo 2. El poder de la introspección

- 2.2. 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.

- 2.3. `type`, `str`, `dir`, y otras funciones incorporadas
  - La *Referencia de bibliotecas de Python* documenta todas las funciones incorporadas y todas las excepciones incorporadas.

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

- 2.6. La peculiar naturaleza de `and` y `or`
  - Python Cookbook expone alternativas al truco `and-or`.

- 2.7. Utilización de las funciones `lambda`
  - 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`.

Apéndice B. Repaso en cinco minutos

- 2.7. 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.
- 2.8. 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 ya está dispuesto de la manera en que lo necesitamos. Las fichas del dominó están en su sitio; lo que queda es golpear la primera.
- 2.9. Resumen

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

Apéndice C. Trucos y consejos

Capítulo 1. Conozcamos Python

- 1.1. Inmersión

Sugerencia: Ejecutar un módulo (Windows)
En el entorno de programación Python en Windows, puede ejecutar un módulo con File−>Run... (**Control−R**). La salida se muestra en la ventana interactiva.

Sugerencia: Ejecutar un módulo (Mac OS)
En el entorno de programación Python en Mac OS, puede ejecutar un módulo con Python−>Run window... (**Cmd−R**), pero antes debe seleccionar una opción importante: Abra el módulo en el entorno de programación, muestre el menú de opciones de módulo pulsando el triángulo negro de la esquina superior derecha de la ventana, y asegúrese de que "Run as __main__" está seleccionado. Esta opción se guarda junto con el módulo, de modo que sólo debe hacer esto una vez por cada módulo.

Sugerencia: Ejecutar un módulo (UNIX)
En los sistemas compatibles con UNIX (incluido Mac OS X), puede ejecutar un módulo desde la línea de órdenes: `python odbchelper.py`

- 1.2. Declaración de funciones

Nota: Python frente a Visual Basic: valores de retorno
En Visual Basic, las funciones (que devuelven un valor) empiezan con `function`, y las subrutinas (que no devuelven un valor) empiezan con `sub`. En Python no hay subrutinas. Todo son funciones, todas las funciones devuelven un valor (incluso si este es `None`), y todas las funciones empiezan con `def`.

Nota: Python frente a Java: valores de retorno
En Java, C++ y otros lenguajes de tipos estáticos, se debe especificar el tipo de valor de retorno de una función y de cada argumento. En Python, nunca se especifica explícitamente el tipo de nada. Según el valor que se le asigne, Python almacena el tipo de dato internamente.

- 1.3. Documentación de funciones

Nota: Python frente a Perl: entrecomillados
Las triples comillas son también una forma fácil de definir una cadena que incluya comillas simples y dobles, como `qq/.../` en Perl.

Nota: Por qué son buenas las cadenas de documentación
Muchos entornos de programación de Python utilizan la cadena de documentación para proporcionar ayuda sensible al contexto, de modo que cuando se escribe el nombre de una función, su cadena de documentación se muestra como ayuda. Esto puede ser increíblemente útil, pero sólo es tan bueno como lo sean las cadenas de documentación que se escriban.

- 1.4. Todo es un objeto

Nota: Python frente a Perl: importar
`import` en Python es como `require` en Perl. Cuando se ha importado un módulo en Python, se puede acceder a sus funciones con *`módulo.función`*; cuando se ha requerido

un módulo en Perl, se puede acceder a sus funciones con *`módulo::función`*.

- 1.5. Sangrado del código

  Nota: Python frente a Java: separación de sentencias
  Python usa el retorno de carro para separar sentencias y los dos puntos y el sangrado para separar bloques de código. C++ y Java usan el punto y coma para separar sentencias y las llaves para separar bloques de código.

- 1.6. Prueba de módulos

  Nota: Python frente a C: comparación y asignación
  Al igual que C, Python usa `==` para comparar y `=` para asignar. Pero a diferencia de C, Python no admite la asignación en línea, de modo que no hay posibilidad de asignar accidentalmente el valor con el que se piensa que se está comparando.

  Sugerencia: `if __name__` en Mac OS
  En MacPython, hay que añadir un paso más para que funcione el truco `if __name__`. Mostrar el menú de opciones del módulo pulsando en el triángulo negro de la esquina superior derecha de la ventana, y asegurarse de que Run as __main__ está seleccionado.

- 1.7. Todo sobre los diccionarios

  Nota: Python frente a Perl: diccionarios
  Un diccionario en Python es como un hash en Perl. En Perl, las variables que almacenan hashes comienzan siempre por el carácter `%`; en Python, las variables pueden tener cualquier nombre, y Python guarda internamente el tipo de dato.

  Nota: Python frente a Java: diccionarios
  Un diccionario en Python es como una instancia de la clase `Hashtable` en Java.

  Nota: Python frente a Visual Basic: diccionarios
  Un diccionario en Python es como una instancia del objeto `Scripting.Dictionary` en Visual Basic.

  Nota: Los diccionarios no tienen orden
  Los diccionarios no tienen concepto alguno de orden entre sus elementos. Es incorrecto decir que los elementos están "desordenados"; simplemente no hay orden. Esta distinción es importante, y le estorbará cuando intente acceder a los elementos de un diccionario en un orden específico y repetible (como el orden alfabético de las claves). Hay formas de hacer esto, sólo que no forman parte de los diccionarios.

- 1.8. Todo sobre las listas

  Nota: Python frente a Perl: listas
  Una lista en Python es como un array en Perl. En Perl, las variables que almacenan arrays comienzan siempre con el carácter `@`; en Python, las variables pueden llamarse de cualquier modo, y Python lleva el registro del tipo de datos internamente.

  Nota: Python frente a Java: listas
  Una lista en Python es muy parecida a un array en Java (aunque se puede usar de ese modo si eso es todo lo que se espera en la vida). Se puede comparar mejor con la clase `Vector`, que

puede guardar objetos arbitrarios y expandirse dinámicamente al añadir nuevos elementos.

Nota: ¿Qué es verdadero en Python?
No hay un tipo booleano en Python. En un contexto booleano (como una sentencia `if`), `0` es falso y el resto de los números son verdaderos. Esto se extiende también a otros tipos de datos. Una cadena vacía (`""`), una lista vacía (`[]`) y un diccionario vacío (`{}`) son todos falsos; el resto de cadenas, listas y diccionarios son verdaderos.

- 1.9. Todo sobre las tuplas

Nota: De tuplas a listas y a tuplas
Las tuplas pueden convertirse 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 la práctica, `tuple` congela una lista, y `list` descongela una tupla.

- 1.10. Definición de variables

Nota: Órdenes de varias líneas
Cuando una orden se extiende a lo largo de varias líneas con la marca de continuación de línea ("\"), las líneas que siguen pueden sangrarse de cualquier modo; las severas normas de sangradeo de Python no se aplican. Si entorno de programación Python sangra automáticamente las líneas que continúan, debe usted aceptar probablemente el valor por omisión a no ser que tenga una buena razón.

Nota: Órdenes de varias líneas implícitas
Para ser exactos, las expresiones entre paréntesis, corchetes o llaves (como la definición de un diccionario) pueden extenderse a varias líneas con la marca de continuación ("\") o sin ella. Yo prefiero incluir la contrabarra incluso cuando no es imprescindible, porque creo que hace el código más legible, pero esto es cuestión de estilo.

- 1.12. Formato de cadenas

Nota: Python frente a C: formato de cadenas
El formato de cadenas en Python utiliza la misma sintaxis que la función `sprintf` en C.

- 1.14. Unión y división de cadenas

Importante: No se pueden unir más que cadenas
`join` sólo funciona con listas de cadenas; no hace ninguna conversión de tipos. Si se une una lista que tiene uno o más elementos que no sean cadenas, se lanzará una excepción.

Nota: Buscar con `split`
`cadena``.split(`*`delimitador`*`, 1)` es una técnica útil cuando se quiere buscar una subcadena en una cadena y después trabajar con lo que la precede (que queda en el primer elemento de la lista devuelta) y lo que la sigue (que queda en el segundo elemento).

Capítulo 2. El poder de la introspección

- 2.2. Argumentos opcionales y con nombre

Nota: Llamar a funciones es flexible
Lo único que debe hacerse para llamar 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.

- 2.3. `type`, `str`, `dir`, y otras funciones incorporadas

  Nota: Python se autodocumenta
  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, o, Dios le socorra, MSDN) para recordar cómo se usan estos módulos, Python está ampliamente autodocumentado.

- 2.6. La peculiar naturaleza de `and` y `or`

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

- 2.7. Utilización de las funciones `lambda`

  Nota: Las funciones `lambda` son opcionales
  Las funciones `lambda` son una cuestión de estilo. Su uso nunca es necesario. En cualquier lugar 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.

- 2.8. Todo junto

  Nota: Python frente a SQL: comparación de valores nulos
  En sql, se utiliza `IS NULL` en vez de `= NULL` para comparar un valor nulo. En Python no hay una sintaxis especial; se usa `== None` como en cualquier otra comparación.

Apéndice D. Lista de ejemplos

Apéndice E. Historial de revisiones

Historial de revisiones	
Revisión 1.0	30 de septiembre de 2001
• Versión inicial: capítulos 1 y 2 y material complementario.	

Apéndice F. Sobre este libro

Este libro se escribió en DocBook XML 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, que incluyen también las hojas de estilo XSL adaptadas utilizadas 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.

Apéndice G. GNU Free Documentation License

Version 1.1, March 2000

> Copyright (C) 2000 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111−1307 USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

G.0. Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

G.1. Applicability and definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front−matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front−Cover Texts or Back−Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine−readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic

text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard−conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine−generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

G.2. Verbatim copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

G.3. Copying in quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front−Cover Texts on the front cover, and Back−Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine−readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly−accessible computer−network location containing a complete Transparent copy of the Document, free of added material, which the general network−using public has access to download anonymously at no charge using public−standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

G.4. Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front−matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties−−for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front−Cover Text, and a passage of up to 25 words as a Back−Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front−Cover Text and one of Back−Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

G.5. Combining documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

G.6. Collections of documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

G.7. Aggregation with independent works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self−contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

G.8. Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

G.9. Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

G.10. Future revisions of this license

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

G.11. How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front−Cover Texts being LIST, and with the Back−Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front−Cover Texts, write "no Front−Cover Texts" instead of "Front−Cover Texts being LIST"; likewise for Back−Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Apéndice H. Python 2.1.1 license

H.A. History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI) in the Netherlands as a successor of a language called ABC. Guido is Python's principal author, although it includes many contributions from others. The last version released from CWI was Python 1.2. In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI) in Reston, Virginia where he released several versions of the software. Python 1.6 was the last of the versions released by CNRI. In 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. Python 2.0 was the first and only release from BeOpen.com.

Following the release of Python 1.6, and after Guido van Rossum left CNRI to work with commercial software developers, it became clear that the ability to use Python with software available under the GNU Public License (GPL) was very desirable. CNRI and the Free Software Foundation (FSF) interacted to develop enabling wording changes to the Python license. Python 1.6.1 is essentially the same as Python 1.6, with a few minor bug fixes, and with a different license that enables later versions to be GPL−compatible. Python 2.1 is a derivative work of Python 1.6.1, as well as of Python 2.0.

After Python 2.0 was released by BeOpen.com, Guido van Rossum and the other PythonLabs developers joined Digital Creations. All intellectual property added from this point on, starting with Python 2.1 and its alpha and beta releases, is owned by the Python Software Foundation (PSF), a non−profit modeled after the Apache Software Foundation. See http://www.python.org/psf/ for more information about the PSF.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.

H.B. Terms and conditions for accessing or otherwise using Python

H.B.1. PSF license agreement

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 2.1.1 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty−free, world−wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 2.1.1 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright (c) 2001 Python Software Foundation; All Rights Reserved" are retained in Python 2.1.1 alone or in any derivative version prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.1.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 2.1.1.
4. PSF is making Python 2.1.1 available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 2.1.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.1.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.1.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By copying, installing or otherwise using Python 2.1.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

H.B.2. BeOpen Python open source license agreement version 1

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").
2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non−exclusive, royalty−free, world−wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.
3. BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the "BeOpen Python" logos available at http://www.pythonlabs.com/logos.html may be used according to the permissions granted on that web page.
7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

H.B.3. CNRI open source GPL−compatible license agreement

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 ("CNRI"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty−free, world−wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided,

however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., "Copyright (c) 1995−2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): "Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: http://hdl.handle.net/1895.22/1013".

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.
4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non−separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By clicking on the "ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

H.B.4. CWI permissions statement and disclaimer

Copyright (c) 1991 − 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS

SOFTWARE.