Guía de aprendizaje de Python

Contenido

Release 2.4.1a0, documentation updated on septiembre 11, 2005.
Consultar en Acerca de este documento... información para sugerir cambios.

Letra pequeña

Resumen:

Python es un lenguaje de programación fácil de aprender y potente. Dispone de eficaces estructuras de datos de alto nivel y una solución de programación orientada a objetos simple pero eficaz. La elegante sintaxis de Python, su gestión de tipos dinámica y su naturaleza interpretada hacen de él el lenguaje ideal para guiones (scripts) y desarrollo rápido de aplicaciones en muchas áreas y en la mayoría de las plataformas.

El intérprete de Python y su extensa biblioteca estándar están disponibles libremente, en forma de fuentes o ejecutables, para las plataformas más importantes, en la sede web de Python, http://www.python.org, y se pueden distribuir libremente. La misma sede contiene también distribuciones y direcciones de muchos módulos, programas y herramientas Python de terceras partes, además de documentación adicional.

Es fácil ampliar el intérprete de Python con nuevas funciones y tipos de datos implementados en C o C++ (u otros lenguajes a los que se pueda acceder desde C). Python es también adecuado como lenguaje de extensión para aplicaciones adaptables al usuario.

Esta guía presenta informalmente al lector los conceptos y características básicos del lenguaje y sistema Python. Es conveniente tener a mano un intérprete para hacer experimentos, pero todos los ejemplos son autosuficientes, así que la guía se puede leer sin estar conectado.

Para obtener descripciones de módulos u objetos estándar, consulta el documento *Referencia de las bibliotecas*. El *Manual de Referencia de Python* ofrece una definición más formal del lenguaje. Para escribir extensiones en C o C++, lee los manuales de *Extensión y empotrado* y *Referencia de la API Python/C*. Existen también diversos libros que tratan Python con detalle.

Esta guía no intenta ser exhaustiva ni agotar cada posibilidad de Python, ni siquiera las más comúnmente utilizadas. En lugar de esto, introduce muchas de las capacidades que caracterizan a Python y proporciona una idea clara del estilo y sabor del lenguaje. Tras su lectura, el lector será capaz de leer y escribir módulos y programas en Python y estará preparado para aprender más sobre los diversos módulos de biblioteca de Python descritos en la *Referencia de las bibliotecas*.

Esta traducción al castellano fue realizada por Marcos Sánchez Provencio (rapto@arrakis.es), con correcciones de Sonia Rubio Hernando.

Release 2.4.1a0, documentation updated on septiembre 11, 2005.
Consultar en Acerca de este documento... información para sugerir cambios.

1. Abriendo el apetito

Si en alguna ocasión has escrito un guion para un intérprete de órdenes (o *shell script*) de UNIX[1.1] largo, puede que conozcas esta sensación: Te encantaría añadir una característica más, pero ya es tan lento, tan grande, tan complicado...O la característica involucra una llamada al sistema u otra función accesible sólo desde C. El problema en sí no suele ser tan complejo como para transformar el guion en un programa en C. Igual el programa requiere cadenas de longitud variable u otros tipos de datos (como listas ordenadas de nombres de fichero) fáciles en sh, pero tediosas en C. O quizá no tienes tanta soltura con C.

Otra situación: Quizá tengas que trabajar con bibliotecas de C diversas y el ciclo normal en C de escribir-compilar-probar-recompilar es demasiado lento. Necesitas desarrollar software con más velocidad. Posiblemente has escrito un programa al que vendría bien un lenguaje de extensión y no quieres diseñar un lenguaje, escribir y depurar el intérprete y adosarlo a la aplicación.

En tales casos, Python puede ser el lenguaje que necesitas. Python es simple, pero es un lenguaje de programación real. Ofrece más apoyo e infraestructura para programas grandes que el intérprete de órdenes. Por otra parte, también ofrece mucha más comprobación de errores que C y, al ser un *lenguaje de muy alto nivel*, tiene incluidos tipos de datos de alto nivel, como matrices flexibles y diccionarios, que llevarían días de programación en C. Dados sus tipos de datos más generales, se puede aplicar a un rango de problemas más amplio que *Awk* o incluso *Perl*, pero muchas cosas son, al menos, igual de fáciles en Python que en esos lenguajes.

Python te permite dividir su programa en módulos reutilizables desde otros programas en Python. Viene con una gran colección de módulos estándar que puedes utilizar como base de tus programas (o como ejemplos para empezar a aprender Python). También hay módulos incluidos que proporcionan E/S de ficheros, llamadas al sistema, ``sockets" y hasta interfaces gráficas con el usuario, como Tk.

Python es un lenguaje interpretado, lo que ahorra un tiempo considerable en el desarrollo del programa, pues no es necesario compilar ni enlazar. El intérprete se puede utilizar de modo interactivo, lo que facilita experimentar con características del lenguaje, escribir programas desechables o probar funciones durante el desarrollo del programa de la base hacia arriba. También es una calculadora muy útil.

Python permite escribir programas muy compactos y legibles. Los programas escritos en Python son normalmente mucho más cortos que sus equivalentes en C o C++, por varios motivos:

- Los tipos de datos de alto nivel permiten expresar operaciones complejas en una sola sentencia.
- El agrupamiento de sentencias se realiza mediante sangrado (indentación) en lugar de begin/end o llaves.
- No es necesario declarar los argumentos ni las variables.

Python es *ampliable*: si ya sabes programar en C, es fácil añadir una nueva función o módulo al intérprete, para realizar operaciones críticas a la máxima velocidad o para enlazar programas en Python con bibliotecas que sólo están disponibles en forma binaria (como bibliotecas de gráficos específicas del fabricante). Una vez enganchado, puedes enlazar el intérprete de Python a una

aplicación escrita en C y utilizarlo como lenguaje de macros para dicha aplicación.

Por cierto, el nombre del lenguaje viene del espectáculo de la BBC ``Monty Python's Flying Circus'' (el circo ambulante de Monty Python) y no tiene nada que ver con desagradables reptiles. Hacer referencias a escenas de Monty Python no sólo se permite: ¡es recomendable!

Ahora que estás emocionado con el Python, querrás examinarlo con más detalle. Como la mejor manera de aprender un lenguaje es utilizarlo, desde aquí te invitamos a hacerlo con esta guía.

En el siguiente capítulo se explica la mecánica de uso del intérprete. Es información un tanto aburrida, pero esencial para probar los ejemplos posteriores.

El resto de la guía presenta varias características del sistema y del lenguaje Python mediante ejemplos, empezando por las expresiones, sentencias y tipos de datos sencillos, pasando por las funciones y los módulos, para finalizar con una primera visión de conceptos avanzados, como excepciones y clases definidas por el usuario.

Notas al pie

...[1.1]

En Windows, el intérprete de órdenes se corresponde con la ventana MS-DOS y los guiones con los archivos .bat, aunque la potencia de los guiones UNIX es mucho mayor.

---

2. Utilización del intérprete de Python

2.1 Llamar al intérprete

En UNIX, el intérprete de Python se suele instalar como /usr/local/bin/python en aquellas máquinas donde esté disponible. En Windows, se instala en el directorio Archivos de programa. Poner este directorio en la ruta de ejecutables hace posible arrancarlo tecleando en el intérprete de órdenes la orden:

```
python
```

Como la elección del directorio donde reside el intérprete es una opción de instalación, es posible que se halle en otros lugares. Consulta con tu guru de Python local o tu administrador de sistemas (por ejemplo, /usr/local/python es una alternativa frecuente).

Teclear un carácter fin de fichero (`Control-D` en UNIX, `Control-Z` en DOS o Windows) en el intérprete causa la salida del intérprete con un estado cero. Si eso no funciona, se puede salir del intérprete tecleando las siguientes órdenes: "`import sys; sys.exit()`".

Las opciones de edición de la línea de órdenes no son muy destacables. En UNIX, es posible que quien instalara el intérprete en tu sistema incluyera soporte para la biblioteca de GNU `readline', que permite una edición de línea más elaborada y la recuperación de órdenes anteriores. El modo más rápido de ver si hay soporte de edición de líneas es teclear Control-P en cuanto aparece el intérprete. Si pita, la edición de líneas está disponible (en el Apéndice A hay una introducción a las teclas de edición). Si no sale nada o sale `P`, no está disponible la edición de líneas y sólo se puede utilizar la tecla de borrado para borrar el último carácter tecleado.

El intérprete funciona como el intérprete de órdenes de UNIX: cuando se lo llama con la entrada estándar conectada a un dispositivo tty, lee y ejecuta las órdenes interactivamente; cuando se le da un nombre de fichero como argumento o se le da un fichero como entrada estándar, lee y ejecuta un guion desde ese fichero.

Otro modo de arrancar el intérprete es "**`python -c`** *orden* `[argumento] ...`", que ejecuta las sentencias de *orden*, de forma análoga a la opción **`-c`** de la línea de órdenes. Como las sentencias de Python suelen contener espacios u otros caracteres que la línea de órdenes considera especiales, lo mejor es encerrar *orden* entre dobles comillas por completo.

Hay módulos de Python que son útiles como programas independientes. Se los puede llamar mediante "**`python -m`** *módulo* `[arg] ...`", que ejecuta el fichero de código fuente de *module* como si se hubiera dado el nombre completo en la línea de órdenes.

Observa que hay una diferencia entre "**`python`** `fichero`" y "**`python`** `<fichero`". En el caso de la redirección, las solicitudes de entrada del programa, tales como llamadas a `input()` y `raw_input()`, se satisfacen desde *fichero*. Como este fichero ya se ha leído hasta el final antes de empezar la

ejecución del programa, el programa se encuentra el fin de fichero inmediatamente. En el caso del nombre de fichero como argumento, las solicitudes de entrada son satisfechas desde lo que esté conectado a la entrada estándar (esto suele ser lo deseado).

Cuando se utiliza un fichero de guion, a veces es útil ejecutar el guion y entrar en modo interactivo inmediatamente después. Esto se consigue pasando **-i** como argumento, antes del nombre del guion (esto no funciona si el guion se lee desde la entrada estándar, por la misma razón indicada en el párrafo anterior).

2.1.1 Traspaso de argumentos

El intérprete pasa el nombre del guion y los argumentos, si los conoce, mediante la variable `sys.argv`, que es una lista de cadenas. Su longitud es al menos uno (cuando no hay guion y no hay argumentos, `sys.argv[0]` es una cadena vacía). Cuando se usa **-m** *módulo*, se le da a `sys.argv[0]` el nombre completo del módulo adecuado. Cuando el guion es `'-'` (es decir, la entrada estándar), `sys.argv[0]` vale `'-'`. Cuando se utiliza **-c** *orden*, `sys.argv[0]` vale `'-c'`. Las opciones tras **-c** *orden* o **-m** *módulo* no las utiliza el intérprete Python, sino que quedan en `sys.argv` para uso de la orden o del módulo.

2.1.2 Modo interactivo

Cuando se leen órdenes desde una tty, se dice que el intérprete está en *modo interactivo*. En este modo, espera a la siguiente orden con el *indicador principal*, que suele se tres signos `mayor' (">>>" ). Para las líneas adicionales, se utiliza el *indicador secundario*, por omisión tres puntos ("`...` ").

El intérprete muestra un mensaje de bienvenida con su número de versión e información de derechos de copia, antes de mostrar el primer indicador:

```
python
Python 1.5.2b2 (#1, Feb 28 1999, 00:02:06)  [GCC 2.8.1] on sunos5
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>>
```

Las líneas de continuación son necesarias al introducir construcciones milti-línea. Por ejemplo, echa un vistazo a esta sentencia `if`:

```
>>> la_tierra_es_plana = 1
>>> if la_tierra_es_plana:
...     print "¡Cuidado, que te caes!"
...
¡Cuidado, que te caes!
```

2.2 El intérprete y su entorno

2.2.1 Gestión de errores

Cuando ocurre un error, el intérprete muestra un mensaje de error y una traza de la pila. En el modo interactivo, después vuelve al indicador principal. Si la entrada venía de un fichero, sale con un resultado distinto de cero tras mostrar la traza de la pila (las excepciones gestionadas con una sentencia `except` en una construcción `try` no son errores en este contexto). Existen errores no capturables que hacen que se cierre el intérprete con un resultado distinto de cero. Por ejemplo, esto ocurre con las inconsistencias internas y, en algunos casos, al quedarse sin memoria. Todos los mensajes de error se escriben en la salida de error estándar (la pantalla, si no se redirige a un fichero u otra cosa). La salida del programa se escribe en la salida estándar (que también es la pantalla, salvo en el caso mencionado antes).

Si se teclea el carácter de interrupción (suele ser Control-C o DEL) en el indicador principal o secundario se cancela la entrada y se hace volver el indicador primario[2.1]. Si se intenta interrumpir mientras se ejecuta una orden, se activa la excepción `KeyboardInterrupt`, que puede ser gestionada por una construcción `try`.

2.2.2 Guiones Python ejecutables

En sistemas UNIX tipo BSD, los guiones Python se pueden hacer ejecutables directamente, como guiones de línea de órdenes, poniendo la línea

```
#! /usr/bin/env python
```

(suponiendo que el intérprete está en el PATH del usuario) al principio del guion y dándole al guion permisos de ejecución. "`#!`" deben ser los primeros caracteres del fichero. En algunas plataformas, esta primera línea debe terminar con un fin de línea tipo UNIX ("`\n`"), no tipo Mac OS ("`\r`") ni Windows ("`\r\n`"). Observa que la almohadilla, "#", se utiliza para iniciar un comentario en Python.

Se le puede dar al guion un modo o permiso ejecutable, con la orden **chmod**:

```
$ chmod +x miguion.py
```

2.2.3 Codificación del código fuente

Se puede utilizar codificaciones distintas de ASCII en los archivos de código fuente Python. El mejor modo es poner una línea de comentario justo después de la línea de `#!` (si no la hay, ponerla la primera) para definir la codificación del código fuente:

```
# -*- coding: codificación -*-
```

Con esta declaración, se indica que todos los caracteres del código fuente tienen la codificación y será posible escribir directamente literales de cadena Unicode en la codificación seleccionada. La lista de codificaciones posibles se encuentra en la *Referencia de las bibliotecas de Python*, en la sección sobre `codecs`.

Por ejemplo, para escribir literales Unicode que incluyan el símbolo de la moneda Euro, se puede utilizar la codificación ISO-8859-15, en la que el símbolo del Euro tiene el ordinal 164. Este guion mostrará el valor 8364 (el ``codepoint'' Unicode correspondiente al símbolo del Euro) y terminará:

```
# -*- coding: iso-8859-15 -*-

moneda = u"€"
print ord(moneda)
```

Si tu editor tiene la posibilidad de guardar ficheros como `UTF-8` con una *marca de orden de bytes* UTF-8 (también conocida como BOM), también se podrá usar esto en lugar de una declaración de codificación. IDLE tiene esta capacidad si se establece `Opciones/General/Codificación de la fuente predeterminada/UTF-8`. Obsérvese que esta firma no la entienden las antiguas versiones de Python (2.2 y anteriores), ni tampoco los sistemas operativos en el caso de guiones con líneas `#!` (sólo para los sistemas tipo UNIX).

Mediante el uso de UTF-8 (bien por la firma o por la declaración de codificación), se pueden utilizar caracteres de la mayoría de los lenguajes tanto en los literales como en los comentarios. No se puede usar caracteres no ASCII en los identificadores. Para mostrar dichos caracteres adecuadamente, tu editor debe reconocer que el fichero es UTF-8, además de usar un tipo de letra que contenga los caracteres hallados en el fichero.

2.2.4 El fichero de arranque interactivo

Al usar Python interactivamente, suele ser útil que se ejecuten algunas órdenes estándar cada vez que se arranca el intérprete. Esto se puede lograr poniendo en la variable de entorno PYTHONSTARTUP el nombre del fichero que contiene las órdenes de arranque. Es similar a la característica .profile de la línea de órdenes de UNIX o al fichero autoexec.bat de MS-DOS.

Este fichero sólo se lee en sesiones interactivas, no cuando Python lee órdenes de un guion, ni cuando se utiliza /dev/tty como fuente explícita de órdenes (lo que hace que se comporte como una sesión interactiva salvo en este detalle). Estas órdenes se ejecutan en el mismo espacio nominal que las órdenes interactivas, para que los objetos definidos o módulos importados se puedan usar sin necesidad de prefijos en la sesión interactiva. También se puede cambiar los indicadores principal y secundario (`sys.ps1` y `sys.ps2`) usando este fichero.

Si deseas leer un archivo de arranque adicional del directorio actual puedes programarlo así en el fichero de arranque global, de este modo: "`if os.path.isfile('.pythonrc.py'): execfile('.pythonrc.py')`". Si deseas utilizar el fichero de arranque en un guion, debes hacerlo explícitamente dentro del guion:

```
import os
nombreFich = os.environ.get('PYTHONSTARTUP')
if nombreFich and os.path.isfile(nombreFich):
    execfile(nombreFich)
```

Notas al pie

... primario[2.1]
Puede haber problemas con el paquete GNU readline que impidan esto.

Release 2.4.1a0, documentation updated on septiembre 11, 2005.
Consultar en Acerca de este documento... información para sugerir cambios.

3. Introducción informal a Python

En los siguientes ejemplos, la entrada y la salida se distinguen por la presencia o ausencia de indicadores (">>> " y ". . . "). Para repetir el ejemplo debe teclear todo lo que sigue al indicador, cuando aparezca éste. Las líneas que no empiezan por un indicador son la salida del intérprete. Observa que un indicador secundario solo en una línea indica que debe teclear una línea en blanco. Esto se utiliza para finalizar una orden multi-línea.

Muchos de los ejemplos de este manual, incluidos los que se escriben interactivamente, incluyen comentarios. Los comentarios en Python empiezan por el carácter almohadilla, "#", y se extienden hasta el final de la línea física. Se puede iniciar un comentario al principio de una línea o tras espacio en blanco o código, pero no dentro de una constante literal. Una almohadilla dentro de una cadena es, simplemente, una almohadilla.

Ejemplos:

```
# éste es el primer comentario
fiambre = 1                 # y éste
                            # ... ¡y un tercero!
cadena = "# Esto no es un comentario."
```

3.1 Python como calculadora

Vamos a probar algunas órdenes simples de Python. Arranca el intérprete y espera a que aparezca el indicador principal, ">>>" (no debería tardar mucho).

3.1.1 Números

El intérprete funciona como una simple calculadora: Tú tecleas una expresión y él muestra el resultado. La sintaxis de las expresiones es bastante intuitiva: Los operadores +, -, * y / funcionan como en otros lenguajes (p. ej. Pascal o C). Se puede usar paréntesis para agrupar operaciones. Por ejemplo:

```
>>> 2+2
4
>>> # Esto es un comentario
... 2+2
4
>>> 2+2  # un comentario junto al código
4
>>> (50-5*6)/4
5
>>> # La división entera redondea hacia abajo:
... 7/3
```

```
2
>>> 7/-3
-3
```

Se usa el signo de igualdad "=" para asignar un valor a una variable. Tras la asignación, no se presenta nada antes de presentar el siguiente indicador:

```
>>> ancho = 20
>>> alto = 5*9
>>> ancho * alto
900
```

Se puede asignar un valor simultáneamente a varias variables:

```
>>> x = y = z = 0  # Poner a cero 'x', 'y' y 'z'
>>> x
0
>>> y
0
>>> z
0
```

La coma flotante funciona de la manera esperada. Los operadores con tipos mixtos convierten el operando entero a coma flotante:

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

También funcionan de la manera esperada los números complejos: Los números imaginarios se escriben con el sufijo "`j`" o "`J`". Los números complejos con una parte real distinta de cero se escriben "`(`*real*`+`*imag*`j)`", y se pueden crear con la función "`complex(`*real*`, `*imag*`)`".

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

Los números complejos siempre se representan como dos números de coma flotante, la parte real y la imaginaria. Para extraer una de las partes de un número complejo *z*, usa `z.real` o `z.imag`.

```
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
```

Las funciones de conversión a coma flotante y a entero (`float()`, `int()` y `long()`) no funcionan con números complejos, pues no hay un modo único y correcto de convertir un complejo a real. Usa

`abs(z)` para sacar su módulo (como flotante) o `z.real` para sacar su parte real.

```
>>> a=3.0+4.0j
>>> float(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use e.g. abs(z)
>>> a.real
3.0
>>> a.imag
4.0
>>> abs(a)  # sqrt(a.real**2 + a.imag**2)
5.0
>>>
```

En modo interactivo, la última expresión impresa se asigna a la variable _. Esto significa que, cuando se usa Python como calculadora, se facilita continuar los cálculos, por ejemplo:

```
>>> iva = 12.5 / 100
>>> precio = 100.50
>>> precio * iva
12.5625
>>> precio + _
113.0625
>>> round(_, 2)
113.06
>>>
```

Sólo debe leer esta variable. No le asignes un valor explícitamente, ya que crearías una variable local del mismo nombre y enmascararías la variable interna que proporciona la funcionalidad especial.

3.1.2 Cadenas

Además de los números, Python también sabe manipular cadenas, que se pueden expresar de diversas maneras. Se pueden encerrar entre comillas simples o dobles:

```
>>> 'fiambre huevos'
'fiambre huevos'
>>> 'L\'Hospitalet'
"L'Hospitalet"
>>> "L'Hospitalet"
"L'Hospitalet"
>>> '"Sí," dijo.'
'"Sí," dijo.'
>>> "\"Sí,\" dijo."
'"Sí," dijo.'
>>> '"En L\'Hospitalet," dijo.'
'"En L\'Hospitalet," dijo.'
```

Las cadenas pueden ocupar varias líneas de diferentes maneras. Se puede impedir que el final de línea física se interprete como final de línea lógica usando una barra invertida al final de cada línea parcial:

```
hola = "Esto es un texto bastante largo que contiene\n\
varias líneas de texto, como si fuera C.\n\
```

```
    Observa que el espacio en blanco al principio de la línea es\
 significativo."
print hola
```

observa que los saltos de línea se han de incrustar en la cadena con `\n`, pues el salto de línea físico se desecha. El ejemplo mostraría lo siguiente:

```
Esto es un texto bastante largo que contiene
varias líneas de texto, como si fuera C.
    Observa que el espacio en blanco al principio de la línea es signif
```

Sin embargo, si hacemos de la cadena una cadena ``en bruto", las secuencias `\n` no se convierten en saltos de línea, sino que se incluyen tanto la barra invertida y el carácter de salto de línea del código fuente en la cadena como datos. Así, el ejemplo:

```
hola = r"Ésta es una cadena bastante larga que contiene\n\
varias líneas de texto de manera parecida a como se haría en C."

print hola
```

mostraría:

```
Ésta es una cadena bastante larga que contiene\n\
varias líneas de texto de manera parecida a como se haría en C.
```

O se pueden encerrar las cadenas entre comillas triples emparejadas: `"""` o `'''`. No es necesario poner barra invertida en los avances de línea cuando se utilizan comillas triples; se incluyen en la cadena.

```
print """
Uso: cosilla [OPCIONES]
     -h                        Mostrar este mensaje de uso
     -H NombreServidor         Nombre del servidor al que conectarse
"""
```

presenta:

```
Uso: cosilla [OPCIONES]
     -h                        Mostrar este mensaje de uso
     -H NombreServidor         Nombre del servidor al que conectarse
```

El intérprete muestra los resultados de las operaciones con cadenas como se escriben a la entrada: Entre comillas y con las comillas y otros caracteres raros marcados por barras invertidas, para mostrar el valor exacto. La cadena se encierra entre comillas dobles si contiene una comilla simple y no contiene comillas dobles, si no, se encierra entre comillas simples (se puede utilizar `print` para escribir cadenas sin comillas ni secuencias de escape).

Se puede concatenar cadenas (pegarlas) con el operador `+` y repetirlas con `*`:

```
>>> palabra = 'Ayuda' + 'Z'
>>> palabra
'AyudaZ'
>>> '<' + palabra*5 + '>'
'<AyudaZAyudaZAyudaZAyudaZAyudaZ>'
```

Dos literales juntos se concatenan automáticamente. La primera línea de arriba se podría haber escrito "`palabra = 'Ayuda' 'Z'`". Esto sólo funciona con literales, no con expresiones de cadena arbitrarias.

```
>>> import string
>>> 'cad' 'ena'                  #  <-  Esto vale
'cadena'
>>> 'cad'.strip() + 'ena'   #  <-  Esto vale
'cadena'
>>> 'cad'.strip() 'ena'     #  <-  Esto no vale
  File "<stdin>", line 1, in ?
    string.strip('cad') 'ena'
                        ^
SyntaxError: invalid syntax
```

Se puede indexar una cadena. Como en C, el primer carácter de una cadena tiene el índice 0. No hay un tipo carácter diferente; un carácter es una cadena de longitud uno. Como en Icon, las subcadenas se especifican mediante la *notación de corte*: dos índices separados por dos puntos.

```
>>> palabra[4]
'a'
>>> palabra[0:2]
'Ay'
>>> palabra[2:4]
'ud'
```

Los índices de corte tienen valores por omisión muy prácticos; si se omite el primer índice se supone cero y si se omite el segundo se supone el tamaño de la cadena sometida al corte.

```
>>> palabra[:2]    # Los primeros dos caracteres
'Ay'
>>> palabra[2:]    # Todos menos los primeros dos caracteres
'udaZ'
```

He aquí un comportamiento útil en las operaciones de corte: `s[:i] + s[i:]` equivale a `s`.

```
>>> palabra[:2] + palabra[2:]
'AyudaZ'
>>> palabra[:3] + palabra[3:]
'AyudaZ'
```

A diferencia de las cadenas en C, las cadenas de Python no se pueden cambiar. Si se intenta asignar a una posicion indexada dentro de la cadena se produce un error:

```
>>> palabra[0] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
>>> palabra[:1] = 'Choof'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support slice assignment
```

Sin embargo crear una nueva cadena con el contenido combinado es fácil y eficaz:

```
>>> 'x' + palabra[1:]
'xyudaZ'
```

```
>>> 'Choof' + word[4]
'ChoofZ'
```

Los índices degenerados se tratan con elegancia: un índice demasiado grande se reemplaza por el tamaño de la cadena, un índice superior menor que el inferior devuelve una cadena vacía.

```
>>> palabra[1:100]
'yudaZ'
>>> palabra[10:]
''
>>> palabra[2:1]
''
```

Los índices pueden ser negativos, para hacer que la cuenta comience por el final. Por ejemplo:

```
>>> palabra[-1]     # El último carácter
'Z'
>>> palabra[-2]     # El penúltimo carácter
'a'
>>> palabra[-2:]    # Los dos últimos caracteres
'aZ'
>>> palabra[:-2]    # Todos menos los dos últimos
'Ayud'
```

Pero date cuenta de que -0 es 0, así que ¡no cuenta desde el final!

```
>>> palabra[-0]     # (porque -0 es igual a 0)
'A'
```

Los índices de corte negativos fuera de rango se truncan, pero no ocurre así con los índices simples (los que no son de corte):

```
>>> palabra[-100:]
'AyudaZ'
>>> palabra[-10]    # error
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
```

El mejor modo de recordar cómo funcionan los índices es pensar que apuntan *al espacio entre* los caracteres, estando el borde izquierdo del primer carácter numerado 0. El borde derecho del último carácter de una cadena de *n* caracteres tiene el índice *n*, por ejemplo:

```
 +---+---+---+---+---+---+
 | A | y | u | d | a | Z |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

La primera fila de números da la posición de los índices 0..5 de la cadena; la segunda da los índices negativos correspondientes. El corte desde *i* hasta *j* consta de todos los caracteres entre los bordes etiquetados *i* y *j*, respectivamente.

Para los índices no negativos, la longitud del corte es la diferencia entre los índices, si los dos están entre límites. Por ejemplo, la longitud de `palabra[1:3]` es 2.

La función interna `len()` devuelve la longitud de una cadena:

```
>>> s = 'supercalifragilisticoexpialidoso'
>>> len(s)
32
```

Ver también:

Tipos secuenciales
: Las cadenas normales y las cadenas Unicode descritas en la siguiente sección son ejemplos de *tipos secuenciales* y dan soporte a las operaciones comunes de dicho tipo.

Métodos de cadena
: Tanto las cadenas como las cadenas Unicode dan soporte a una gran cantidad de métodos de transformaciones básicas y búsquedas.

Operaciones de formato de cadenas
: Aquí se describen con más detalle las operaciones de formato invocadas cuando las cadenas y las cadenas Unicode son el operador izquierdo del operador `%`.

3.1.3 Cadenas Unicode

A partir de Python 2.0, el programador dispone de un nuevo tipo de datos para almacenar datos de texto: el objeto Unicode. Se puede usar para almacenar y manipular datos Unicode (consultar http://www.unicode.org/) y se integra bien con los objetos de cadena existentes, proporcionando conversiones automáticas si se da el caso.

La codificación Unicode tiene la ventaja de proporcionar un ordinal para cada sistema de escritura utilizado en textos antiguos y modernos. Anteriormente, había sólo 256 ordinales para los caracteres escritos y se solía asociar los textos a una página de códigos, que hacía corresponder los ordinales con los caracteres escritos. Esto llevaba a una gran confusión, especialmente en lo relativo a la internacionalización (comúnmente escrito "`i18n`" -- "`i`" + 18 caracteres + "`n`") del software. Unicode resuelve estos problemas definiendo una página de códigos única para todos los sistemas de escritura.

Crear cadenas Unicode en Python es tan simple como crear cadenas normales:

```
>>> u'Muy buenas'
u'Muy buenas'
```

La "`u`" minúscula frente a la comilla indica que se ha de crear una cadena Unicode. Si deseas incluir caracteres especiales dentro de la cadena, lo puedes hacer mediante la codificación *Unicode-Escape* de Python. El siguiente ejemplo muestra cómo:

```
>>> u'Muy\u0020buenas'
u'Muy buenas'
```

La secuencia de escape `\u0020` indica que se ha de insertar el carácter Unicode con ordinal hexadecimal 0x0020 (el espacio) en la posición indicada.

El resto de los caracteres se interpretan utilizando sus ordinales respectivos directamente como ordinales Unicode. Si se tienen cadenas en la codificación estándar Latin-1 utilizada en muchos países occidentales[3.1], resulta bastante práctico que los primeros 256 caracteres de Unicode sean los mismos que los de Latin-1.

Para los expertos, existe también un modo en bruto, como para las cadenas normales. Se debe prefijar la cadena con 'ur' para que Python utilice la codificación *En bruto-Unicode-Escape*. Sólo aplicará la conversión citada `\uXXXX` si hay un número impar de barras invertidas frente a la "u".

```
>>> ur'Muy\u0020buenas'
u'Muy buenas'
>>> ur'Muy\\u0020buenas'
u'Muy\\\\u0020buenas'
```

El modo en bruto es útil cuando hay que meter gran cantidad de barras invertidas, como suele resultar necesario en las expresiones regulares.

Además de estas codificaciones estándar, Python proporciona un conjunto completo de modos de crear cadenas Unicode basándose en una codificación conocida.

La función interna `unicode()` proporciona acceso a todos los codes (codificadores/descodificadores) Unicode registrados. Algunas de las codificaciones más conocidas a las que pueden convertir estos codes son *Latin-1*, *ASCII*, *UTF-8* y *UTF-16*. Los últimos dos son codificaciones de longitud variable que almacenan cada carácter Unicode como uno o más bytes. La codificación predeterminada suele ser ASCII, que admite los caracteres del 0 al 127 y rechaza el resto de los caracteres. Cuando se presenta una cadena, se escribe en un fichero o se convierte con `str()`, se lleva a cabo la conversión con esta codificación predeterminada.

```
\begin{verbatim}
>>> u"abc"
u'abc'
>>> str(u"abc")
'abc'
>>> u"äöü"
u'\xe4\xf6\xfc'
>>> str(u"äöü")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
UnicodeEncodeError: 'ascii' codec can't encode characters in position 0·
```

Para convertir una cadena Unicode a una cadena de caracteres de 8 bit usando una codificación específica, los objetos Unicode proporcionan un método `encode()` que toma un argumento, el nombre de la codificación. Son preferibles los nombres en minúsculas.

```
>>> u"äöü".encode('utf-8')
'\xc3\xa4\xc3\xb6\xc3\xbc'
```

Si dispones de datos en una codificación concreta y quieres obtener su cadena Unicode correspondiente, usa la función `unicode()` con el nombre de la codificación como segundo argumento.

```
>>> unicode('\xc3\xa4\xc3\xb6\xc3\xbc', 'utf-8')
u'\xe4\xf6\xfc'
```

3.1.4 Listas

Python utiliza varios tipos de datos *compuestos*, que se utilizan para agrupar otros valores. El más versátil es la *lista*, que se puede escribir como una lista de valores (elementos) separada por comas entre corchetes. Los elementos de una lista no tienen que ser todos del mismo tipo.

```
>>> a = ['fiambre', 'huevos', 100, 1234]
>>> a
['fiambre', 'huevos', 100, 1234]
```

Como los índices de las cadenas, los índices de una lista empiezan en cero. Las listas también se pueden cortar, concatenar, etc.:

```
>>> a[0]
'fiambre'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['huevos', 100]
>>> a[:2] + ['panceta', 2*2]
['fiambre', 'huevos', 'panceta', 4]
>>> 3*a[:3] + ['¡Hey!']
['fiambre', 'huevos', 100, 'fiambre', 'huevos', 100, 'fiambre', 'huevos
```

A diferencia de las cadenas, que son *inmutables*, es posible cambiar los elementos de una lista:

```
>>> a
['fiambre', 'huevos', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['fiambre', 'huevos', 123, 1234]
```

Se puede asignar a un corte, lo que puede hasta cambiar el tamaño de la lista:

```
>>> # Reemplazar elementos:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Quitar elementos:
... a[0:2] = []
>>> a
[123, 1234]
>>> # Insertar cosas:
... a[1:1] = ['puaj', 'xyzzy']
>>> a
[123, 'puaj', 'xyzzy', 1234]
>>> a[:0] = a     # Insertarse (una copia) al principio de ella misma
>>> a
[123, 'puaj', 'xyzzy', 1234, 123, 'puaj', 'xyzzy', 1234]
```

La función interna `len()` se aplica también a las listas:

```
>>> len(a)
```

```
8
```

Es posible anidar listas (crear listas que contienen otras listas), por ejemplo:

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
>>> p[1].append('xtra')     # Consulte la sección 5.1
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

Observa que, en el último ejemplo, `¡p[1]` y `q` se refieren en realidad al mismo objeto! Volveremos al tema de la *semántica de objetos* más tarde.

3.2 Primeros pasos programando

Por supuesto, se puede usar Python para tareas más complejas que sumar dos y dos. Por ejemplo, podemos escribir una secuencia parcial de la serie de *Fibonacci*[3.2] de este modo:

```
>>> # Serie de Fibonacci:
... # La suma de dos elementos nos da el siguiente
... a, b = 0, 1
>>> while b < 10:
...       print b
...       a, b = b, a+b
...
1
1
2
3
5
8
```

Este ejemplo introduce varias características nuevas.

- La primera línea contiene una *asignación múltiple*: a las variables `a` y `b` se les asignan a la vez los nuevos valores 0 y 1. En la última línea se utiliza esto de nuevo, demostrando que las expresiones del lado derecho se evalúan antes que la primera de las asignaciones. Las expresiones del lado derecho se evalúan de izquierda a derecha.

- El bucle `while` se ejecuta mientras la condición (en este caso: `b < 10`) sea cierta. En Python, como en C, cualquier valor entero distinto de cero es verdadero y 0 es falso. La condición también puede ser una lista o cualquier secuencia, cualquier cosa con longitud distinta de cero es verdadero, las secuencias vacías son falso. La comprobación en este caso es simple. Los operadores de comparación estándar se escriben igual que en C: < (menor que), > (mayor que), == (igual a), <= (menor o igual a), >= (mayor o igual a) y `!=` (distinto de).

- El *cuerpo* del bucle está *sangrado* (o indentado): Éste es el modo en que Python agrupa las sentencias. Python (todavía) no ofrece un servicio de edición de líneas sangradas, así que hay que teclear a mano un tabulador o espacio(s) para cada línea sangrada. En la práctica, los programas más complejos se realizan con un editor de texto y la mayoría ofrece un servicio de sangrado automático. Cuando se introduce una sentencia compuesta interactivamente, se debe dejar una línea en blanco para indicar el final (porque el analizador de sintaxis no puede adivinar cuándo has acabado) Observa que cada línea de un bloque básico debe estar sangrada exactamente al mismo nivel.

- La sentencia `print` escribe el valor de la expresión o expresiones dadas. Se diferencia de escribir simplemente la expresión (como hemos hecho antes en los ejemplos de la calculadora) en el modo en que gestiona las expresiones múltiples y las cadenas. Las cadenas se imprimen sin comillas y se inserta un espacio entre los elementos para queden las cosas colocadas:

```
>>> i = 256*256
>>> print 'El valor de i es', i
El valor de i es 65536
```

  Si se pone una coma al final se evita el retorno de carro tras la salida:

```
>>> a, b = 0, 1
>>> while b < 1000:
...     print b,
...     a, b = b, a+b
...
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

  Observa que el intérprete inserta una nueva línea antes de presentar el indicador si la última línea quedó incompleta.

Notas al pie

... occidentales[3.1]

En España, Windows utiliza WinANSI (cp1252), que es muy parecido a Latin-1. MS-DOS y Windows en consola utilizan una codificación propia, denominada OEM a veces, en la que no coinciden algunos caracteres, en concreto las letras acentuadas. Supongo que esto coincide con otros países en los que se habla castellano. Las distribuciones de Linux actuales (2000) utilizan Latin-1 siempre. Además, ahora se empieza a utilizar una codificación alterada que incluye el carácter Euro.

...Fibonacci[3.2]

La serie de Fibonacci (matemático que vivió en Pisa de 1170 a 1250) se caracteriza porque cada elemento es la suma de los dos anteriores, excepto los dos primeros, que son 0 y 1

Release 2.4.1a0, documentation updated on septiembre 11, 2005.
Consultar en Acerca de este documento... información para sugerir cambios.

4. Más herramientas de control de flujo

Además de la sentencia `while`, recién presentada, Python conoce las sentencias de control de flujo comunes de otros lenguajes, aunque con sabor propio.

4.1 Construcciones `if`

Quizá la mejor conocida de las construcciones es `if` (si). Por ejemplo:

```
>>> x = int(raw_input("Introduce un número entero: "))
>>> if x < 0:
...      x = 0
...      print 'Negativo cambiado a cero'
... elif x == 0:
...      print 'Cero'
... elif x == 1:
...      print 'Uno'
... else:
...      print 'Más'
...
```

Puede haber cero o más partes `elif` y la parte `else` (si no) es opcional. La palabra clave `elif' es una abreviatura de `else if' y evita el sagrado excesivo. Una secuencia `if` ... `elif` ... `elif` ... es la sustituta de las sentencias `switch` o `case` de otros lenguajes.

4.2 Sentencias `for`

La construcción `for` (para) es un poco diferente a lo acostumbrado en C o Pascal. En lugar de recorrer siempre una progresión aritmética (como en Pascal) o dejar al programador total libertad de elección de inicialización, comprobación y salto de paso (como en C), el `for`de Python recorre los elementos de una secuencia (una lista o cadena), en el orden en que aparecen en la secuencia. Por ejemplo:

```
>>> # Medir algunas cadenas:
... a = ['gato', 'ventana', 'defenestrar']
>>> for x in a:
...     print x, len(x)
...
gato 4
ventana 7
defenestrar 11
```

No es aconsejable modificar la secuencia que se está recorriendo (lo que sólo puede ocurrir en

secuencias mutables, como las listas). Si se necesita modificar la lista recorrida, por ejemplo, para duplicar los elementos, hay que recorrer una copia. La notación de corte hace esto muy cómodo.

```
>>> for x in a[:]: # hacer una copia por corte de la lista entera
...     if len(x) > 7: a.insert(0, x)
...
>>> a
['defenestrar', 'gato', 'ventana', 'defenestrar']
```

4.3 La función `range()`

Si lo que necesitas es recorrer una secuencia de números, la función interna `range()` viene de perlas. Genera listas con progresiones aritméticas:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

El punto final indicado nunca es parte de la lista generada, así que `range(10)` genera una lista de 10 valores, justo los índices legales de los elementos de una secuencia de longitud 10. Es posible hacer que el rango arranque en otro número o especificar un incremento diferente (incluso negativo). Este incremento se llama paso (step):

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

Para recorrer los índices de una secuencia, combina `range()` y `len()` de este modo:

```
>>> a = ['Cinco', 'lobitos', 'tiene', 'la', 'loba']
>>> for i in range(len(a)):
...     print i, a[i]
...
0 Cinco
1 lobitos
2 tiene
3 la
4 loba
```

4.4 Construcciones con `break`, `continue` y `else` en bucles

La sentencia `break` (romper), como en C, salta del bucle `for` o `while` en curso más interno.

La sentencia `continue` (continuar), también un préstamo de C, hace que siga la siguiente iteración del bucle.

Las construcciones de bucle pueden tener una cláusula `else`. Ésta se ejecuta, si existe, cuando se termina el bucle por agotamiento de la lista (con `for`) o cuando la condición se hace falsa (con `while`), pero no cuando se termina el bucle con `break`. Para aclarar esto último, valga un ejemplo, que busca números primos:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...            print n, '=', x, '*', n/x
...            break
...     else:
...          #Se terminó el bucle sin encontrar ningún factor
...          print n, 'es primo'
...
2 es primo
3 es primo
4 = 2 * 2
5 es primo
6 = 2 * 3
7 es primo
8 = 2 * 4
9 = 3 * 3
```

4.5 Construcciones con pass

La sentencia `pass` no hace nada. Se puede utilizar cuando hace falta una sentencia sintácticamente pero no hace falta hacer nada. Por ejemplo:

```
>>> while True:
...       pass # Espera activamente una interrupción de teclado
...
```

4.6 Definición de funciones

Se puede crear una función que escriba la serie de Fibonacci hasta un límite superior arbitrario:

```
>>> def fib(n):    # escribir la serie Fibonacci hasta n
...     """Escribir la serie Fibonacci hasta n."""
...     a, b = 0, 1
...     while b < n:
...         print b,
...         a, b = b, a+b
...
>>> # Y ahora llamamos a la función recién definida:
... fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

La palabra clave `def` introduce una *definición* de función. Debe ir seguida del nombre de la función y la lista entre paréntesis de los parámetros formales. Las sentencias que forman el cuerpo de la función empiezan en la siguiente línea y deben ir sangradas. La primera sentencia del cuerpo de la función puede ser una constante de cadena: esta cadena es la documentación de la función o

docstring.

Existen herramientas para producir automáticamente documentación impresa/electrónica o permitir al usuario navegar por el código interactivamente. Es una buena práctica incluir documentación en el código que escribas, así que intenta hacer de ello un hábito.

La *ejecución* de una función introduce una tabla de símbolos nueva para las variables locales de Python. En concreto, todas las asignaciones de variables de una función almacenan el valor en la tabla de símbolos local; por lo que las referencias a variables primero miran en la tabla de símbolos local, luego en la tabla de símbolos global y, por último, en la tabla de nombres internos. Por ello no se puede asignar un valor a una variable global dentro de una función (salvo que esté mencionada en una sentencia `global`), pero se puede hacer referencia a ellas.

Los parámetros reales (argumentos) de una llamada a una función se introducen en la tabla de símbolos local de la función aludida al llamarla: los argumentos se pasan *por valor* (en donde el *valor* siempre es una *referencia* a un objeto, no el valor del objeto[4.1]. Cuando una función llama a otra función, se crea una tabla de símbolos locales nueva para esa llamada.

Una definición de función introduce el nombre de la función en la tabla de símbolos vigente. El valor del nombre de la función tiene un tipo reconocido por el intérprete como función definida por el usuario. Se puede asignar este valor a otro nombre y usar éste, a su vez, como función que es. Esto sirve de mecanismo de renombrado genérico:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
1 1 2 3 5 8 13 21 34 55 89
```

Se puede objetar que `fib` no es una función, sino un procedimiento. En Python, como en C, los procedimientos son simplemente funciones que no devuelven ningún valor. De hecho, hablando técnicamente, los procedimientos sí devuelven un valor, sólo que bastante aburrido. Este valor se llama `None` (es un nombre interno). El intérprete suele omitir la escritura del valor de `None`, si es el único valor que se fuera a escribir. Se puede ver si realmente lo deseas:

```
>>> print fib(0)
None
```

Resulta simple escribir una función que devuelva una lista de los números de la serie de Fibonacci, en lugar de mostrarla:

```
>>> def fib2(n): # Devolver la serie de Fibonacci hasta n
...     """Devolver una lista con los números de la serie de Fibonacci h
...     resultado = []
...     a, b = 0, 1
...     while b < n:
...         resultado.append(b)    # ver más abajo
...         a, b = b, a+b
...     return resultado
...
>>> f100 = fib2(100)    # llamarlo
>>> f100                # escribir el resultado
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Este ejemplo, como es habitual, demuestra algunas características nuevas de Python:

- La sentencia `return` devuelve la ejecución al que llamó a la función, devolviendo un valor. Si se utiliza `return` sin argumento se devuelve `None`. Si se acaba el código de la función, también se devuelve `None`.

- La sentencia `resultado.append(b)` llama a un *método* del objeto lista `resultado`. Un método es una función que `pertenece' a un objeto y se llama `obj.nombreMétodo`, donde `obj` es un objeto (que puede resultar de una expresión) y `nombreMétodo` es el nombre del método definido por el tipo del objeto. Los métodos de diferentes tipos pueden tener el mismo nombre sin ambigüedad. Es posible definir tus propios tipos de objetos y métodos, utilizando *clases*, según se discute más adelante en esta guía. El método `append()` (empalmar), mostrado en el ejemplo, está definido para objetos lista: Añade un elemento nuevo al final de la lista. En este ejemplo es equivalente a "`resultado = resultado + [b]`", pero más eficaz.

4.7 Más sobre la definición de funciones

También es posible definir funciones con un número variable de argumentos. Existen tres formas, que se pueden combinar.

4.7.1 Valores por omisión en los argumentos

Es la forma más útil de especificar un valor por omisión para uno o más de los argumentos. Esto crea una función a la que se puede llamar con menos argumentos de los que tiene definidos:

```
def confirmar(indicador, intentos=4, queja='¡O sí o no!'):
    while True:
        respuesta = raw_input(indicador)
        if respuesta in ('s', 'si', 'sí'): return True
        if respuesta in ('n', 'no', 'nanay', 'nasti'): return False
        intentos = intentos - 1
        if intentos < 0: raise IOError, 'Usuario rechazado'
        print queja
```

Se puede llamar a esta función así: `confirmar('¿Quiere salir?')` o así: `confirmar('¿Desea borrar el fichero?', 2)`.

Este ejemplo también presenta la palabra clave `in`. Ésta comprueba si una secuencia contiene un valor dado.

Los valores por omisión se evalúan en el instante de definición de la función en el ámbito *de definición*, así:

```
i = 5
def f(arg=i):
    print arg
i = 6
f()
```

mostrará `5`.

Aviso importante: El argumento por omisión se evalúa una sola vez. Esto lleva a diferentes

resultados cuando el valor por omisión es un objeto mutable, tal como una lista, diccionario o una instancia de la mayoría de las clases. Por ejemplo, la siguiente función acumula los argumentos que se le pasan en sucesivas llamadas:

```
def f(a, L = []):
    L.append(a)
    return L
print f(1)
print f(2)
print f(3)
```

Esto presenta:

```
[1]
[1, 2]
[1, 2, 3]
```

Si no se desea que el valor por omisión sea compartido por las sucesivas llamadas, se puede escribir la función de este modo:

```
def f(a, L = None):
    if L is None:
        L = []
    L.append(a)
    return L
```

4.7.2 Argumentos por clave

También se puede llamar a una función utilizando la forma "*clave* = *valor*". Por ejemplo, la siguiente función:

```
def loro(tension, estado='tieso', accion='voom', tipo='Azul noruego'):
    print "-- Este loro no podría", accion,
    print "aunque le aplicara", tension, "voltios."
    print "-- Bello plumaje, el", tipo
    print "-- ¡Está", estado, "!"
```

puede invocarse de estas maneras:

```
loro(1000)
loro(accion = 'VOOOOOM', tension = 1000000)
loro('mil', estado = 'criando malvas')
loro('un millón de', 'desprovisto de vida', 'saltar')
```

pero las siguientes llamadas serían todas incorrectas:

```
loro()                      # falta un argumento obligatorio
loro(tension=5.0, 'muerto')  # argumento clave seguido por argumento no·
loro(110, tension=220)       # valor de argumento duplicado
loro(actor='John Cleese')    # clave desconocida
```

En general, una lista de argumentos debe tener argumentos posicionales seguidos por argumentos clave, donde las claves se seleccionan de los nombres de los parámetros formales. No importa si un parámetro formal tiene valor por omisión o no. Ningún argumento debe recibir valor más de una vez

(los nombres de parámetros formales correspondientes a argumentos posicionales no se pueden usar como claves en la misma llamada). He aquí un ejemplo que falla por culpa de esta restricción:

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: function() got multiple values for keyword argument 'a'
```

Cuando el último parámetro formal tiene la forma ***nombre* recibe un diccionario que contiene todos los argumentos clave salvo los que corresponden con parámetros formales. Esto se puede combinar con un parámetro formal de la forma **nombre* (descrito en la siguiente subsección) que recibe una tupla que contiene los argumentos posicionales que exceden la lista de parámetros formales (**nombre* debe aparecer antes de ***nombre*). Por ejemplo, si definimos una función como ésta:

```
def queseria(clase, *argumentos, **palabrasclave):
    print "-- ¿Tiene", clase, '?'
    print "-- Lo siento, no nos queda", clase
    for arg in argumentos: print arg
    print '-'*40
    claves = palabrasclave.keys()
    claves.sort()
    for kw in claves:
        print kw, ':', palabrasclave[kw]
```

se puede invocar de estas maneras:

```
queseria('Limburger', "Chorrea mucho, señor.",
          "Chorrea mucho, muchísimo.",
          cliente='John Cleese',
          tendero='Michael Palin',
          escena='Escena de la quesería')
```

Y mostraría:

```
-- ¿Tiene Limburger ?
-- Lo siento, no nos queda Limburger
Chorrea mucho, señor.
Chorrea mucho, muchísimo.
----------------------------------------
cliente : John Cleese
tendero : Michael Palin
escena : Escena de la quesería
```

Obsérvese que se llama al método `sort()` de la lista de los nombres de los argumentos clave antes de mostrar el contenido del diccionario `palabrasclave`; si no se hace esto no está determinado el orden en que se presentan los argumentos.

4.7.3 Listas de argumentos arbitrarias

Finalmente, la opción menos frecuente es especificar que una función puede llamarse con un número arbitrario de argumentos. Estos argumentos se agruparán en una tupla. Antes del número variable de

argumentos puede haber cero o más argumentos normales.

```
def fprintf(file, formato, *args):
    file.write(formato % args)
```

4.7.4 Desempaquetado de listas de argumentos

La situación inversa se da cuando los argumentos ya están en una lista o tupla pero se han de desempaquetar para llamar a una función que requiere argumentos posicionales separados. Por ejemplo, la función interna `range()` espera recibir los argumentos separados *inicio* y *fin*. Si no están disponibles por separado, escribe la llamada a la función con el operador `*` para desempaquetar los argumentos de una lista o tupla:

```
>>> range(3, 6)             # llamada normal con argumentos separados
[3, 4, 5]
>>> args = [3, 6]
>>> range(*args)            # llamada con los argumentos desempaquetados
[3, 4, 5]
```

4.7.5 Formas lambda

A petición popular, se han añadido a Python algunas características comúnmente halladas en los lenguajes de programación funcional y Lisp. Con la palabra clave `lambda` es posible crear pequeñas funciones anónimas. Ésta es una función que devuelve la suma de sus dos argumentos: "`lambda a, b: a+b`". Las formas lambda se pueden utilizar siempre que se necesite un objeto función. Están sintácticamente restringidas a una expresión simple. Semánticamente son un caramelo sintáctico para una definición de función normal. Al igual que las definiciones de funciones anidadas, las formas lambda pueden hacer referencia a las variables del ámbito que las contiene:

```
>>> def montar_incrementador(n):
    return lambda x: x + n
>>> f = montar_incrementador(42)
>>> f(0)
42
>>> f(1)
43
```

4.7.6 Cadenas de documentación

Hay convenciones crecientes sobre el contenido y formato de las cadenas de documentación.

La primera línea debe ser siempre un corto y conciso resumen de lo que debe hacer el objeto. En aras de la brevedad, no debes hacer constar el nombre y tipo del objeto, pues éstos están disponibles mediante otros modos (excepto si el nombre es un verbo que describe el funcionamiento de la función). Esta línea debe empezar por mayúscula y terminar en punto.

Si hay más líneas en la cadena de documentación, la segunda línea debe ir en blanco, separando visualmente el resumen del resto de la descripción. Las siguientes líneas deben ser párrafos que describan las convenciones de llamada de los objetos, sus efectos secundarios, etc.

El analizador de Python no elimina el sangrado de los literales multilínea, así que las herramientas que procesen documentación tienen que eliminar el sangrado si se desea. Esto se realiza del siguiente modo. La primera línea que no está en blanco *tras* la primera línea de la documentación determina el grado de sangrado de la cadena de documentación entera (no se puede utilizar la primera línea, porque suele estar pegada a las comillas de apertura y su sangrado no es evidente dentro del literal). Se elimina el espacio en blanco ``equivalente'' a este sangrado del principio de todas las líneas de la cadena. No debería haber líneas menos sangradas, pero si las hay se debe eliminar su espacio en blanco inicial. La equivalencia del espacio en blanco se debe realizar tras la expansión de los tabuladores (a 8 espacios, normalmente).

He aquí un ejemplo de una cadena de documentación multilínea:

```
>>> def mi_funcion():
...     """Esta función no hace nada, pero está muy bien documentada.
...
...     Que no, que no hace nada.
...     """
...     pass
...
>>> print mi_funcion.__doc__
Esta función no hace nada, pero está muy bien documentada.

    Que no, que no hace nada.
```

Notas al pie

... objeto[4.1]

En realidad, *por referencia al objeto* resultaría más correcto, ya que si se pasa un objeto mutable, el que llama verá los cambios realizados por el llamado (por ejemplo, si inserta elementos en una lista).

Release 2.4.1a0, documentation updated on septiembre 11, 2005.
Consultar en Acerca de este documento... información para sugerir cambios.

5. Estructuras de datos

Este capítulo describe con más detalle algunas cosas que ya has visto y añade algunas cosas nuevas.

5.1 Más sobre las listas

El tipo de datos ``lista'' tiene algunos métodos más. Éstos son todos los métodos de los objetos lista:

append(*x*)
Añade un elemento al final de una lista; es equivalente a `a[len(a):] = [x]`.

extend(*L*)
Extiende la lista concatenándole todos los elementos de la lista indicada; es equivalente a `a[len(a):] = L`.

insert(*i*, *x*)
Inserta un elemento en un posición dada. El primer argumento es el índice del elemento antes del que se inserta, por lo que `a.insert(0, x)` inserta al principio de la lista y `a.insert(len(a), x)` equivale a `a.append(x)`.

remove(*x*)
Elimina el primer elemento de la lista cuyo valor es `x`. Provoca un error si no existe tal elemento.

pop([*i*])
Elimina el elemento de la posición dada de la lista y lo devuelve. Si no se especifica un índice, `a.pop()` devuelve el último elemento de la lista y también lo elimina. Los corchetes que rodean la *i* en la signatura del método indican que el parámetro es opcional, no que haya que teclear los corchetes en dicha posición. Esta notación es frecuente en la *Referencia de las bibliotecas de Python*.

index(*x*)
Devuelve el índice del primer elemento de la lista cuyo valor sea `x`. Provoca un error si no existe tal elemento.

count(*x*)
Devuelve el número de veces que aparece `x` en la lista.

sort()
Ordena ascendentemente los elementos de la propia lista (la lista queda cambiada).

reverse()
Invierte la propia lista (la lista queda cambiada).

Un ejemplo que utiliza varios métodos de la lista:

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.25), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.6, 333, 333, 1234.5]
```

5.1.1 Cómo usar las listas como pilas

Los métodos de las listas facilitan mucho usar una lista como una pila, donde el último elemento añadido es el primer elemento recuperado. (``last-in, first-out'', ``último en llegar, primero en salir''). Para apilar un elemento, usa `append()`. Para recuperar el elemento superior de la pila, usa `pop()` sin un índice explícito. Por ejemplo:

```
>>> pila = [3, 4, 5]
>>> pila.append(6)
>>> pila.append(7)
>>> pila
[3, 4, 5, 6, 7]
>>> pila.pop()
7
>>> pila
[3, 4, 5, 6]
>>> pila.pop()
6
>>> pila.pop()
5
>>> pila
[3, 4]
```

5.1.2 Cómo usar las listas como colas

También es muy práctico usar una lista como cola, donde el primer elemento que se añade a la cola es el primero en salir (``first-in, first-out'', ``primero en llegar, último en salir''). Para añadir un elemento al final de una cola, usa `append()`. Para recuperar el primer elemento de la cola, usa `pop()` con `0` de índice. Por ejemplo:

```
>>> cola = ["Eric", "John", "Michael"]
>>> cola.append("Terry")           # llega Terry
```

```
>>> cola.append("Graham")          # llega Graham
>>> cola.pop(0)
'Eric'
>>> cola.pop(0)
'John'
>>> cola
['Michael', 'Terry', 'Graham']
```

5.1.3 Herramientas de programación funcional

Hay tres funciones internas que son muy útiles al tratar con listas: `filter()`, `map()` y `reduce()`.

"`filter(`*función, secuencia*`)`", filtrar, devuelve una secuencia (del mismo tipo, si es posible) que contiene aquellos elementos de la secuencia de entrada para los que *función*`(`*elemento*`)` resulta verdadero. Por ejemplo, para calcular algunos primos:

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
...
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```

"`map(`*función, secuencia*`)`", transformar, llama a *función*`(`*elemento*`)` para cada uno de los elementos de la secuencia y devuelve una lista compuesta por los valores resultantes. Por ejemplo, para calcular algunos cubos:

```
>>> def cubo(x): return x*x*x
...
>>> map(cubo, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

Se puede pasar más de una secuencia como parámetro. La función debe tener tantos argumentos como secuencias se le pasan y se llama a la función con el valor correspondiente de cada secuencia de entrada (o `None` si una secuencia es más corta que otra). Por ejemplo:

```
>>> secuencia = range(8)
>>> def suma(x,y): return x+y
...
>>> map(add, seq, seq)
[0, 2, 4, 6, 8, 10, 12, 14]
```

"`reduce(`*func, secuencia*`)`",reducir, devuelve un valor simple que se construye llamando a la función binaria *func* con los dos primeros elementos de la secuencia, luego con el resultado y el siguiente elemento y así sucesivamente. Por ejemplo, para calcular la suma de los números de 1 a 10:

```
>>> def suma(x,y): return x+y
...
>>> reduce(suma, range(1, 11))
55
```

Si sólo hay un elemento en la secuencia, se devuelve su valor; si la secuencia está vacía, se lanza una excepción.

Se puede pasar un tercer argumento para indicar el valor inicial. En este caso, se devuelve este valor inicial para la secuencia vacía y la función se aplica al primer elemento, luego al segundo y así

sucesivamente. Por ejemplo,

```
>>> def sum(secuencia):
...     def suma(x,y): return x+y
...     return reduce(suma, secuencia, 0)
...
>>> sum(range(1, 11))
55
>>> sum([])
0
```

No uses la función `sum()` de este ejemplo: como sumar números es una tarea común, se proporciona una función interna `sum(`*secuencia*`)` que hace esto exactamente. New in version 2.3.

5.1.4 Listas autodefinidas

Las listas autodefinidas proporcionan un modo conciso de crear listas sin recurrir al uso de `map()`, `filter()` ni `lambda`. La definición de lista resultante tiende a ser más clara que las listas construidas con los métodos citados. Cada LC consta de una expresión seguida de una cláusula `for` y cero o más cláusulas `for` o `if`. La lista resultante se obtiene evaluando la expresión en el contexto de las cláusulas `for` e `if` que la siguen. Si la expresión debe dar como resultado una tupla, hay que encerrarla entre paréntesis.

```
>>> frutafresca = ['  plátano', '  mora ', 'fruta de la pasión  ']
>>> [arma.strip() for arma in frutafresca]
['plátano', 'mora', 'fruta de la pasión']
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
>>> [[x,x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
>>> [x, x**2 for x in vec]  # error - se necesita un paréntesis en las t
  File "<stdin>", line 1, in ?
    [x, x**2 for x in vec]
               ^
SyntaxError: invalid syntax
>>> [(x, x**2) for x in vec]
[(2, 4), (4, 16), (6, 36)]
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
[8, 12, -54]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
[8, 12, -54]
```

Las listas autodefinidas son mucho más lexibles que `map()` y se pueden aplicar a funciones con más de un argumento y a funciones anidadas:

```
>>> [str(round(355/113.0, i)) for i in range(1,6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

5.2 La sentencia `del`

Hay un modo de eliminar un elemento de una lista dado su índice en lugar de su valor: la sentencia `del`. También se puede utilizar para eliminar cortes de una lista (lo que hacíamos antes asignando una lista vacía al corte). Por ejemplo:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
```

`del` se puede utilizar para eliminar variable completas:

```
>>> del a
```

Hacer referencia al nombre `a` a partir de aquí provoca un error (al menos hasta que se asigne otro valor al nombre). Veremos otros usos de `del` más adelante.

5.3 Tuplas y secuencias

Hemos visto que las listas y las cadenas tienen muchas propiedades en común, por ejemplo, el indexado y el corte. Son dos ejemplos de tipos de datos *secuenciales*. Como Python es un lenguaje en evolución, se pueden añadir otros tipos de datos de secuencia. Hay otro tipo de dato secuencial estándar: la *tupla*.

Una tupla consta de cierta cantidad de valores separada por comas, por ejemplo:

```
>>> t = 12345, 54321, '¡hola!'
>>> t[0]
12345
>>> t
(12345, 54321, '¡hola!')
>>> # Se pueden anidar tuplas:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, '¡hola!'), (1, 2, 3, 4, 5))
```

Como puedes observar, en la salida se encierran las tuplas entre paréntesis, para que las tuplas anidadas se interpreten correctamente. En la entrada los paréntesis son opcionales, aunque a menudo son necesarios (si la tupla es parte de una expresión más compleja).

Las tuplas son muy útiles: Por ejemplo, pares de coordenadas (x,y), registros de empleados de una base de datos, etc. Las tuplas, como las cadenas, son inmutables: No es posible asignar un valor a los elementos individuales de una tupla (sin embargo, se puede simular el mismo efecto mediante corte

y concatenación). También es posible crear tuplas que contengan objetos mutables, por ejemplo, listas.

Un problema especial es la construcción de tuplas de 0 ó 1 elementos: La sintaxis tiene trucos para resolver esto. Las tuplas vacías se construyen mediante un par de paréntesis vacío y las tuplas de un solo elemento se construyen mediante el valor del elemento seguido de coma (no vale con encerrar el valor entre paréntesis). Es feo, pero funciona. Por ejemplo:

```
>>> vacio = ()
>>> singleton = 'hola',    # <-- Observa la coma final
>>> len(vacio)
0
>>> len(singleton)
1
>>> singleton
('hola',)
```

La sentencia `t = 12345, 54321, '¡hola!'` es un ejemplo de *empaquetado de tuplas*: los valores `12345`, `54321` y `'¡hola!'` se empaquetan en una tupla. También se puede realizar la operación inversa:

```
>>> x, y, z = t
```

Esto se llama, por supuesto, *desempaquetado de secuencias*. El desempaquetado de secuencias requiere que el número de variables sea igual al número de elementos de la secuencia. Observa que la asignación múltiple sólo es un efecto combinado del empaquetado de tuplas y desempaquetado de secuencias.

Esto resulta un poco asimétrico, ya que el empaquetado de varios valores siempre resulta en una tupla, aunque el desempaquetado funciona para cualquier secuencia.

5.4 Conjuntos

Python también incluye un tipo de dato para los *conjuntos*. Un conjunto es una colección desordenada sin elementos duplicados. Los usos básicos incluyen la comprobación de pertenencia y la eliminación de elementos duplicados. Los objetos conjunto disponen también de operaciones matemáticas como la unión, intersección, diferencia y diferencia simétrica.

He aquí una breve demostración:

```
>>> cesta = ['manzana', 'naranja', 'manzana', 'pera', 'naranja', 'banana
>>> frutas = set(cesta)               # crea un conjunto sin duplicados
>>> frutas
set(['naranja', 'pera', 'manzana', 'banana'])
>>> 'naranja' in frutas                  # comprobación rápida de pertene
True
>>> 'ortigas' in frutas
False

>>> # Demostración de las operaciones de conjuntos sobre las letras de c

>>> a = set('abracadabra')
>>> b = set('alacazam')
```

```
>>> a                                # letras diferentes de a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b                            # letras de a que no están en b
set(['r', 'd', 'b'])
>>> a | b                            # letras que están en a o b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b                            # letras que están en a y también
set(['a', 'c'])
>>> a ^ b                            # letras que están en a y b pero
set(['r', 'd', 'b', 'm', 'z', 'l'])
```

5.5 Diccionarios

Otro tipo de dato interno de Python que resulta útil es el *diccionario*. Los diccionarios aparecen a veces en otros lenguajes como ``memorias asociativas'' o ``matrices asociativas''. A diferencia de las secuencias, que se indexan mediante un rango de números, los diccionarios se indexan mediante *claves*, que pueden ser de cualquier tipo inmutable. Siempre se puede utilizar cadenas y números como claves. Las tuplas pueden usarse de claves si sólo contienen cadenas, números o tuplas. Si una tupla contiene cualquier objeto mutable directa o indirectamente, no se puede usar como clave. No se pueden utilizar las listas como claves, ya que las listas se pueden modificar, por ejemplo, mediante el método `append()` y su método `extend()`, además de las asignaciones de corte y asignaciones aumentadas.

Lo mejor es pensar en un diccionario como en un conjunto desordenado de parejas *clave: valor*, con el requisito de que las claves sean únicas (dentro del mismo diccionario). Una pareja de llaves crea un diccionario vacío: `{}`. Si se coloca una lista de parejas *clave*`:` *valor* entre las llaves se añaden parejas *clave*`:` *valor* iniciales al diccionario. Así es como se presentan los diccionarios en la salida (hay ejemplos dentro de poco).

Las operaciones principales sobre un diccionario son la de almacenar una valor con una clave dada y la de extraer el valor partiendo de la clave. También se puede eliminar una pareja *clave*`:` *valor* con `del`. Si se introduce una clave que ya existe, el valor anterior se olvida. Intentar extraer un valor utilizando una clave no existente provoca un error.

El método `keys()` de un objeto de tipo diccionario devuelve todas las claves utilizadas en el diccionario, en orden arbitrario (si deseas ordenarlas, aplica el método `sort()` a la lista de claves). Para comprobar si una clave existe en el diccionario, utiliza el método `has_key()` del diccionario.

He aquí un pequeño ejemplo que utiliza un diccionario:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> tel.has_key('guido')
```

```
True
```

El constructor `dict()` construye diccionarios directamente a partir de listas de parejas clave-valor guardadas como tuplas. Cuando las parejas cumplen un patrón, se puede especificar de manera compacta las lista de parejas clave-valor.

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
>>> dict([(x, x**2) for x in (2, 4, 6)])     # uso de lista autodefinida
{2: 4, 4: 16, 6: 36}
```

Más adelante en esta guía, conoceremos las expresiones generadoras, que son aún más adecuadas para la tarea de proporcionar parejas clave-valor al constructor `dict()`.

5.6 Técnicas para hacer bucles

Al recorrer diccionarios, es posible recuperar la clave y su valor correspondiente a la vez, utilizando el método `iteritems()`.

```
>>> caballeros = {'gallahad': 'el casto', 'robin': 'el valeroso'}
>>> for k, v in caballeros.iteritems():
...     print k, v
...
gallahad el casto
robin el valeroso
```

Al recorrer una secuencia, se pueden recuperar a la vez el índice de posición y su valor correspondiente usando la función `enumerate()`.

```
>>> for i, v in enumerate(['pim', 'pam', 'pum']):
...     print i, v
...
0 pim
1 pam
2 pum
```

Para recorrer dos o más secuencias en paralelo, se pueden emparejar los valores con la función `zip()`.

```
>>> preguntas = ['nombre', 'misión', 'color favorito']
>>> respuestas = ['lanzarote', 'el santo grial', 'azul']
>>> for p, r in zip(preguntas, respuestas):
...     print '¿Cuál es tu %s? %s.' % (p, r)
...
¿cuál es tu nombre? lanzarote.
¿cuál es tu misión? el santo grial.
¿cuál es tu color favorito? azul.
```

<

Para recorrer una secuencia en orden inverso, hay que especificar primero la secuencia en el orden original y llamar a la función `reversed()`.

```
>>> for i in reversed(xrange(1,10,2)):
...     print i
...
9
7
5
3
1
```

Para recorrer una secuencia en orden, usa la función `sorted()`, que devuelve una lista ordenada nueva, dejando intacta la secuencia original.

```
>>> cesta = ['manzana', 'naranja', 'manzana', 'naranja', 'pera', 'banana
>>> for f in sorted(set(cesta)):
...     print f
...
banana
manzana
naranja
pera
```

5.7 Más sobre las condiciones

Las condiciones utilizadas en construcciones `while` e `if` descritas anteriormente pueden contener cualquier operador, no sólo comparaciones.

Los operadores de comparación `in` (dentro de) y `not in` (no dentro de) comprueban si un valor está incluido (o no) en una secuencia. Los operadores `is` (es) y `is not` (no es) comprueban si dos objetos son en realidad el mismo. Esto sólo tiene importancia en los objetos mutables, como las listas. Todos los operadores de comparación tienen la misma prioridad, que es menor que la de los operadores numéricos.

Se pueden encadenar las comparaciones: Por ejemplo, `a < b == c` comprueba si `a` es menor que `b` y además si `b` es igual a `c`.

Las comparaciones se pueden combinar mediante los operadores lógicos `and` (y) y `or` (o) y la salida de una comparación (o cualquier otra expresión lógica) se puede negar mediante `not` (no). Todos éstos tienen menor prioridad que los operadores de comparación. Entre ellos, `not` tiene la prioridad más elevada y `or` la más baja, por lo que `A and not B or C` equivale a `(A and (not B)) or C`. Como siempre, se pueden utilizar paréntesis para expresar un orden de operación concreto.

Los operadores lógicos `and` y `or` se denominan operadores de *cortocircuito*: Sus argumentos se evalúan de izquierda a derecha y la evaluación se interrumpe tan pronto como queda determinado el valor de salida. Por ejemplo, si `A` tiene un valor de verdadero pero `B` es falso, `A and B and C` no evalúa el valor de la expresión C. En general, el valor devuelto por un operador de atajo, cuando se utiliza como valor en general y no como valor lógico, es el último argumento evaluado.

Es posible asignar el resultado de una comparación u otra expresión lógica a una variable. Por ejemplo:

```
>>> cadena1, cadena2, cadena3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = cadena1 or cadena2 or cadena3
```

```
>>> non_null
'Trondheim'
```

Observa que en Python, al contrario que en C, no puede haber asignación dentro de una expresión. Los programadores en C pueden quejarse de esto, pero evita una causa común problemas hallados en los programas en C: teclear = en una expresión en la que se quería decir ==.

5.8 Comparación entre secuencias y otros tipos

Los objetos de secuencia se pueden comparar con otros objetos del mismo tipo de secuencia. La comparación utiliza ordenación *lexicográfica*: Primero se comparan los dos primeros elementos, si estos difieren ya está determinado el valor de la comparación, si no, se comparan los dos elementos siguientes de cada secuencia y, así sucesivamente, hasta que se agota alguna de las dos secuencias. Si alguno de los elementos que se compara es él mismo una secuencia, se lleva a cabo una comparación lexicográfica anidada. Si todos los elementos son iguales, se considera que las secuencias son iguales. Si una de las secuencias es igual a la otra truncada a partir de cierto elemento, la secuencia más corta de las dos es la menor. La ordenación lexicográfica para las cadenas utiliza el orden de los códigos ASCII de sus caracteres. He aquí ejemplos de comparaciones entre secuencias del mismo tipo:

```
(1, 2, 3)              < (1, 2, 4)
[1, 2, 3]              < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
(1, 2, 3)             == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

Observa que es legal comparar objetos de tipos diferentes. El resultado es determinístico pero arbitrario: los tipos se ordenan por su nombre. De este modo, una lista siempre es menor que una cadena, una cadena siempre es menor que una tupla, etc.[5.1] Los valores numéricos de tipos diferentes se comparan por su valor numérico, por lo que 0 es igual a 0.0, etc.

Notas al pie

... etc.[5.1]

Las reglas de comparación entre objetos de tipos diferentes no son fiables. Pueden cambiar en versiones futuras del lenguaje.

---

6. Módulos

Si sales del intérprete de Python y vuelves a entrar, las definiciones que hayas hecho (funciones y variables) se pierden. Por ello, si quieres escribir un programa algo más largo, será mejor que utilices un editor de texto para preparar la entrada del intérprete y ejecutarlo con ese fichero como entrada. Esto se llama crear un guion. Según vayan creciendo los programas, puede que quieras dividirlos en varios ficheros para facilitar el mantenimiento. Puede que también quieras utilizar una función que has escrito en varios programas sin tener que copiar su definición a cada programa.

Para lograr esto, Python tiene un modo de poner definiciones en un fichero y utilizarlas en un guion o en una instancia interactiva del intérprete. Tal fichero se llama *módulo*; las definiciones de un módulo se pueden *importar* a otros módulos o al módulo *principal* (la colección de variables accesible desde un guion ejecutado desde el nivel superior y en el modo de calculadora).

Un módulo es un fichero que contiene definiciones y sentencias de Python. El nombre del fichero es el nombre del módulo con el sufijo .py. Dentro de un módulo, el nombre del módulo (como cadena) es accesible mediante la variable global `__name__`. Por ejemplo, utiliza tu editor de texto favorito para crear un fichero llamado fibo.py en el directorio actual, con el siguiente contenido:

```
# Módulo de los números de Fibonacci

def fib(n):    # escribir la serie de Fibonacci hasta n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n): # devolver la serie de Fibonacci hasta n
    resultado = []
    a, b = 0, 1
    while b < n:
        resultado.append(b)
        a, b = b, a+b
    return resultado
```

Ahora entra en el intérprete de Python e importa este módulo con la siguiente orden:

```
>>> import fibo
```

Esto no introduce los nombres de las funciones definidas en `fibo` directamente en la tabla de símbolos actual; sólo introduce el nombre del módulo `fibo`. Utilizando el nombre del módulo puedes acceder a las funciones:

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Si pretendes utilizar una función a menudo, la puedes asignar a un nombre local:

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

6.1 Más sobre los módulos

Un módulo puede contener sentencias ejecutables además de definiciones de funciones. Estas sentencias sirven para inicializar el módulo. Sólo se ejecutan la *primera* vez que se importa el módulo en alguna parte[6.1].

Cada módulo tiene su propia tabla de símbolos, que utilizan todas las funciones definidas por el módulo como tabla de símbolos global. Por ello, el autor de un módulo puede utilizar variables globales dentro del módulo sin preocuparse por conflictos con las variables globales de un usuario del módulo. Por otra parte, si sabes lo que haces, puedes tocar las variables globales de un módulo con la misma notación utilizada para referirse a sus funciones, `nombreMod.nombreElem`.

Los módulos pueden importar otros módulos. Es una costumbre no obligatoria colocar todas las sentencias `import` al principio del módulo (o guion). Los nombres del módulo importado se colocan en la tabla de símbolos global del módulo (o guion) que lo importa.

Existe una variación de la sentencia `import` que importa los nombres de un módulo directamente a la tabla de símbolos del módulo que lo importa. Por ejemplo:

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Esto no introduce el nombre del módulo del que se toman los elementos importados en la tabla de símbolos local (por lo que, en el ejemplo, no está definido `fibo`).

Además, existe una variación que importa todos los nombres que define un módulo:

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Esto importa todos los nombres, excepto los que empiezan por un guion bajo (_).

6.1.1 El camino de búsqueda de módulos

Cuando se importa un módulo denominado `fiambre`, el intérprete busca un fichero denominado fiambre.py en el directorio actual y, luego, en la lista de directorios especificada por la variable de entorno PYTHONPATH. Tiene la misma sintaxis que la variable de línea de órdenes PATH de UNIX, que es una lista de nombres de directorios. Cuando PYTHONPATH no tiene ningún valor o no se encuentra el fichero, se continúa la búsqueda en un camino dependiente de la instalación. En UNIX, normalmente es .:/usr/local/lib/python.

En realidad, se buscan los módulos en la lista de directorios dada por la variable `sys.path`, que se inicializa desde el directorio que contiene el guion de entrada (o el directorio actual), PYTHONPATH y el valor por omisión dependiente de la instalación. Esto permite que los programas que saben lo que hacen modifiquen o reemplacen el camino de búsqueda de módulos. Obsérvese que, como el directorio que contiene el guion bajo ejecución está en el camino de búsqueda de módulos, es importante que el módulo no tenga el mismo nombre que un módulo estándar, o Python lo intentará cargar el guion como módulo cuando se importe el módulo. Normalmente, esto provocará errores. Ver la sección 6.2, ``Módulos estándar," para obtener más información.

6.1.2 Ficheros Python ``Compilados"

Como mejora considerable del tiempo de arranque de programas cortos que utilizan muchos módulos estándar, si existe un fichero llamado fiambre.pyc en el directorio donde se encuentra fiambre.py, se supone que contiene una versión previamente ``compilada a byte" del módulo `fiambre`. La fecha y hora de la versión de fiambre.py utilizada para generar fiambre.pyc se graba en fiambre.pyc y no se considera el fichero .pyc si no concuerdan.

Normalmente, no hay que hacer nada para generar el fichero fiambre.pyc. Siempre que fiambre.py se compile sin errores, se hace un intento de escribir la versión compilada a fiambre.pyc. No se provoca un error si falla el intento. Si por cualquier motivo no se escribe completamente el fichero, el fichero fiambre.pyc resultante será reconocido como no válido y posteriormente ignorado. El contenido del fichero fiambre.pyc es independiente de la plataforma, por lo que se puede compartir un directorio de módulos entre máquinas de diferentes arquitecturas.

Consejos para los expertos:

- Cuando se llama al intérprete de Python con el indicador **-O**, se genera código optimizado, que se almacena en ficheros .pyo. El optimizador actual no resulta de gran ayuda, sólo elimina sentencias `assert`. Cuando se utiliza **-O**, *todo* el código de byte se optimiza. Se hace caso omiso de los ficheros `.pyc` y se compilan los ficheros `.py` a código byte optimizado.

- Pasar dos indicadores **-O** al intérprete de Python (**-OO**) hace que el compilador a código byte realice optimizaciones que, en casos poco frecuentes, dan como resultado programas que no funcionan correctamente. Actualmente, sólo se eliminan las cadenas `__doc__` del código byte, lo que da como resultado ficheros .pyo más compactos. Como hay programas que suponen que estas cadenas están disponibles, sólo se debería utilizar esta opción con conocimiento de causa.

- Un programa no se ejecuta más rápido cuando se lee de un fichero .pyc o .pyo que cuando se lee de un fichero .py. La única diferencia es el tiempo que tarda en cargarse.

- Cuando se ejecuta un guion dando su nombre en la línea de órdenes, nunca se escribe el código byte en un fichero .pyc o .pyo. Por ello, se puede reducir el tiempo de arranque de un guion moviendo la mayoría del código a un módulo y dejando un pequeño arranque que importa el módulo. También es posible nombrar directamente un fichero .pyc o .pyo en la línea de órdenes.

- Es posible tener un módulo llamado fiambre.pyc (o fiambre.pyo si se utiliza **-O**) sin que exista el fichero fiambre.py en el mismo directorio. De este modo se puede distribuir una biblioteca de código de Python dificultando en cierta medida la ingeniería inversa.

- El módulo `compileall` puede generar los ficheros .pyc (o .pyo si se utiliza **-O**) para todos los

módulos de un directorio.

6.2 Módulos estándar

Python viene con una biblioteca de módulos estándar, descrita en un documento aparte, la *Referencia de las bibliotecas*. Algunos módulos son internos al intérprete y proporcionan acceso a las operaciones que no son parte del núcleo del lenguaje pero se han incluido por eficiencia o para proporcionar acceso a primitivas del sistema operativo, como las llamadas al sistema. El conjunto de dichos módulos es una opción de configuración que también depende de la plataforma subyacente. Por ejemplo, el módulo `amoeba` sólo se proporciona en sistemas que de algún modo tienen acceso a primitivas Amoeba. Hay un módulo en particular que merece una especial atención, el módulo `sys`, que es siempre interno en cualquier intérprete de Python. Las variables `sys.ps1` y `sys.ps2` definen las cadenas utilizadas como indicador principal y secundario:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print '¡Puaj!'
¡Puaj!
C>
```

Estas variables sólo están definidas si el intérprete está en modo interactivo.

La variable `sys.path` es una lista de cadenas que determina el camino de búsqueda de módulos del intérprete. Se inicializa a un valor por omisión tomado de la variable de entorno PYTHONPATH o de un valor por omisión interno, si PYTHONPATH no tiene valor. Se puede modificar mediante operaciones de lista estándar, por ejemplo:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

6.3 La función `dir()`

La función interna `dir()` se utiliza para averiguar qué nombres define un módulo. Devuelve una lista de cadenas ordenada:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__stderr__
'__stdin__', '__stdout__', '_getframe', 'api_version', 'argv',
'builtin_module_names', 'byteorder', 'call_tracing', 'callstats', 'copyr
'displayhook', 'exc_clear', 'exc_info', 'exc_type', 'excepthook', 'exec_
'executable', 'exit', 'getcheckinterval', 'getdefaultencoding', 'getdlop
'getfilesystemencoding', 'getrecursionlimit', 'getrefcount', 'hexversion
'last_traceback', 'last_type', 'last_value', 'maxint', 'maxunicode',
```

```
'meta_path', 'modules', 'path', 'path_hooks', 'path_importer_cache', 'p
'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setdlopenflags', 'setprofi
'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout', 'version',
'version_info', 'warnoptions']
```

Sin argumentos, `dir()` enumera la lista de nombres definidos actualmente (por ti o por el sistema):

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo, sys
>>> fib = fibo.fib
>>> dir()
['__builtins__','__doc__','__file__','__name__', 'a', 'fib', 'fibo', 'sy
```

Observa que enumera todo tipo de nombres: variables, módulos, funciones, etc.

`dir()` no devuelve los nombres de las funciones y variables internas. Si deseas obtener esos nombres, están definidos en el módulo estándar `__builtin__`:

```
>>> import __builtin__
>>> dir(__builtin__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'DeprecationWarn
 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
 'FloatingPointError', 'FutureWarning', 'IOError', 'ImportError',
 'IndentationError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
 'LookupError', 'MemoryError', 'NameError', 'None', 'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError', 'OverflowWarning',
 'PendingDeprecationWarning', 'ReferenceError', 'RuntimeError',
 'RuntimeWarning', 'StandardError', 'StopIteration', 'SyntaxError',
 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'True',
 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError',
 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError',
 'UserWarning', 'ValueError', 'Warning', 'WindowsError',
 'ZeroDivisionError', '_', '__debug__', '__doc__', '__import__',
 '__name__', 'abs', 'apply', 'basestring', 'bool', 'buffer',
 'callable', 'chr', 'classmethod', 'cmp', 'coerce', 'compile',
 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod',
 'enumerate', 'eval', 'execfile', 'exit', 'file', 'filter', 'float',
 'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex',
 'id', 'input', 'int', 'intern', 'isinstance', 'issubclass', 'iter',
 'len', 'license', 'list', 'locals', 'long', 'map', 'max', 'min',
 'object', 'oct', 'open', 'ord', 'pow', 'property', 'quit', 'range',
 'raw_input', 'reduce', 'reload', 'repr', 'reversed', 'round', 'set',
 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super',
 'tuple', 'type', 'unichr', 'unicode', 'vars', 'xrange', 'zip']
```

6.4 Paquetes

Los paquetes son un método de estructurar el espacio nominal de módulos de Python, mediante el uso de ``nombres de módulos con punto". Por ejemplo, el nombre de módulo `A.B` hace referencia a un submódulo denominado "`B`" de un paquete denominado "`A`". Del mismo modo que el uso de módulos evita que los autores de diferentes módulos tengan que preocuparse de los nombres de variables globales de los otros, la utilización de nombres de módulo con puntos evita que los autores de paquetes multi-módulo, como NumPy o PIL (la Biblioteca de tratamiento de imagen de Python), tengan que preocuparse de los nombres de los módulos ajenos.

Supón que deseas diseñar una colección de módulos (un paquete) para tratar de manera uniforme ficheros de sonido y datos de sonido. Existen muchos formatos de fichero de sonido (que se suelen distinguir por la extensión, como .wav, .aiff o .au), por lo que podrías necesitar crear y mantener una colección creciente de módulos de conversión entre los diferentes formatos. También existen muchas operaciones posibles sobre los datos de sonido (tales como mezclar, añadir eco, ecualizar o generar un efecto artificial de estereofonía), por lo que, además, estarías escribiendo una serie de módulos interminable para realizar estas operaciones. He aquí una posible estructura de tu paquete (expresado en términos de sistema de ficheros jerárquico):

```
Sonido/                          Paquete de nivel superior
      __init__.py                Inicializa el paquete de sonido
      Formatos/                  Subpaquete de conversiones de formato (
              __init__.py
              leerwav.py
              escriwav.py
              leeraiff.py
              escriaiff.py
              leerau.py
              escriau.py
              ...
      Efectos/                   Subpaquete de efectos de sonido
              __init__.py
              eco.py
              surround.py
              inverso.py
              ...
      Filtros/                   Subpaquete de filtros
              __init__.py
              ecualizador.py
              vocoder.py
              karaoke.py
              ...
```

Al importar el paquete, Python rastrea los directorios de `sys.path` buscando por el subdirectorio de paquetes.

Los ficheros __init__.py son necesarios para que Python trate los directorios como contenedores de paquetes. Se hace así para evitar que los directorios con nombres comunes, como `"test"`, oculten accidentalmente módulos válidos que aparezcan más tarde dentro del camino de búsqueda. En el caso más sencillo, __init__.py puede ser un fichero vacío, pero también puede ejecutar código de inicialización del paquete o actualizar la variable `__all__`, descrita posteriormente.

Los usuarios del paquete pueden importar módulos individuales del paquete, por ejemplo:

```
import Sonido.Efectos.eco
```

De este modo se carga el submódulo `Sonido.Efectos.eco`. Hay que hacer referencia a él por su nombre completo:

```
Sonido.Efectos.eco.filtroeco(entrada, salida, retardo=0.7, aten=4)
```

Un modo alternativo de importar el submódulo es:

```
from Sonido.Efectos import eco
```

Así también se carga el submódulo `eco` y se hace disponible sin su prefijo de paquete, por lo que se

puede utilizar del siguiente modo:

```
eco.filtroeco(entrada, salida, retardo=0.7, aten=4)
```

Y otra variación es importar la función o variable deseada directamente:

```
from Sonido.Efectos.eco import filtroeco
```

De nuevo, se carga el submódulo `eco`, pero se hace la función `filtroeco` disponible directamente:

```
filtroeco(entrada, salida, retardo=0.7, aten=4)
```

Observa que al utilizar `from` *paquete* `import` *elemento*, el elemento puede ser tanto un submódulo (o subpaquete) del paquete como cualquier otro nombre definido por el paquete, como una función, clase o variable. La sentencia `import` comprueba primero si el elemento está definido en el paquete. Si no, asume que es un módulo e intenta cargarlo. Si no lo consigue, se provoca una excepción `ImportError`.

Sin embargo, cuando se utiliza la sintaxis `import` *elemento.subelemento.subsubelemento*, cada elemento menos el último debe ser un paquete. El último elemento puede ser un módulo o un paquete, pero no una clase, función o variable definida en el nivel superior.

6.4.1 Importar * de un paquete

Y ¿qué ocurre cuando el usuario escribe `from Sonido.Efectos import *`? En teoría, debería rastrearse el sistema para encontrar qué submódulos existen en el paquete e importarlos todos. Por desgracia, esta operación no funciona muy bien en las plataformas Windows y Mac, en las que el sistema de ficheros no tiene una idea muy precisa de las mayúsculas de un fichero. En estas plataformas, no hay un modo garantizado de conocer si un fichero ECO.PY debería ser importado como `eco`, `Eco` o `ECO` (por ejemplo, Windows 95 tiene la molesta costumbre de mostrar todos los nombres de fichero con la primera letra en mayúscula). La restricción de nombres de fichero DOS (8+3) añade otro problema para los nombres de módulo largos.

La única solución es que el autor del paquete proporcione un índice explícito del paquete. La sentencia import utiliza la siguiente convención: Si el código del __init__.py de un paquete define una lista llamada `__all__`, se considera que es la lista de nombres de módulos que se deben importar cuando se encuentre `from` *paquete* `import *`. Depende del autor del paquete mantener la lista actualizada cuando se libere una nueva versión del paquete. Los autores del paquete pueden decidir no mantenerlo, si no es útil importar * del paquete. Por ejemplo, el fichero Sonido/Efectos/__init__.py podría contener el siguiente código:

```
__all__ = ["eco", "surround", "inverso"]
```

Esto significaría que `from Sonido.Efectos import *` importaría los tres submódulos mencionados del paquete `Sonido`.

Si `__all__` no está definido, la sentencia `from Sonido.Efectos import *` *no* importa todos los módulos del subpaquete `Sonido.Efectos` al espacio nominal actual. Sólo se asegura de que el paquete `Sonido.Efectos` ha sido importado (ejecutando posiblemente el código de inicialización de __init__.py) y luego importa cualesquiera nombres definidos en el paquete. Esto incluye cualquier nombre definido (y submódulos cargados explícitamente) por __init__.py. También incluye cualquier submódulo del paquete explícitamente importado por sentencias import anteriores. Mira

este código:

```
import Sonido.Efectos.eco
import Sonido.Efectos.surround
from Sonido.Efectos import *
```

En este ejemplo, los módulos eco y surround se importan al espacio nominal vigente porque están definidos en el paquete `Sonido.Efectos` cuando se ejecuta la sentencia `from...import` (esto también funciona si está definido `__all__`).

Observa que en general se debe evitar importar `*` de un módulo o paquete, ya que suele dar como resultado código poco legible. Sin embargo, se puede usar para evitar teclear en exceso en sesiones interactivas y cuando ciertos módulos estén diseñados para exportar sólo nombres que cumplan ciertas reglas.

Recuerda, ¡no hay nada incorrecto en utilizar `from Paquete import submódulo_concreto`! De hecho, es la notación recomendada salvo que el módulo que importa necesite usar submódulos del mismo nombre de diferentes paquetes.

6.4.2 Referencias internas al paquete

Es común que los submódulos necesiten hacerse referencias cruzadas. Por ejemplo, el módulo `surround` podría utilizar el módulo `eco`. De hecho, tales referencias son tan comunes que la sentencia `import` busca antes en el paquete contenedor que en el camino de búsqueda de módulos estándar. Por ello, basta con que el módulo surround use `import eco` o `from eco import filtroeco`. Si el módulo importado no se encuentra en el paquete actual (el paquete del que el módulo actual es submódulo), la sentencia `import` busca un módulo de nivel superior con el nombre dado.

Cuando se estructuran los paquetes en subpaquetes (como el paquete `Sonido` del ejemplo), no hay un atajo para referirse a los submódulos de los paquetes hermanos y se ha de utilizar el nombre completo del subpaquete. Por ejemplo, si el módulo `Sonido.Filtros.vocoder` necesita utilizar el módulo `eco` del paquete `Sonido.Efectos`, debe utilizar `from Sonido.Efectos import eco`.

6.4.3 Paquetes en directorios múltiples

Los paquetes disponen de un atributo especial más, `__path__`. Este atributo se inicializa con una lista que contiene el nombre del directorio que tiene el __init__.py del paquete antes de que el código de ese fichero se ejecute. Esta variable se puede modificar; hacerlo afecta a futuras búsquedas de módulos y subpaquetes contenidos en el paquete.

A pesar de que no se suele necesitar esta característica, se puede usar para extender el juego de módulos que se encuentran en un paquete.

Notas al pie

... parte[6.1]
: En realidad, las definiciones de funciones también son `sentencias' que se ejecutan. La ejecución introduce el nombre de la función en la tabla de símbolos global.

Release 2.4.1a0, documentation updated on septiembre 11, 2005.
Consultar en Acerca de este documento... información para sugerir cambios.

7. Entrada y salida

Hay varios modos de presentar la salida de un programa; se pueden imprimir datos en un modo legible a los humanos o escribirlos en un fichero para uso posterior. Este capítulo explora algunas de las posibilidades.

7.1 Formato de salida mejorado

Hasta ahora hemos encontrado dos modos de escribir valores: las *sentencias de expresión* y la sentencia `print` Hay un tercer modo, utilizar el método `write()` de los objetos fichero, donde se puede acceder al fichero de salida estándar es accesible mediante `sys.stdout`. Consulta la Referencia de las bibliotecas si deseas obtener información sobre el tema.

A menudo, querrás tener más control sobre el formato de la salida que escribir valores separados por espacios. Hay dos modos de dar formato a la salida: El primer modo es gestionar tú mismo las cadenas. Mediante corte y empalme de cadenas se puede generar cualquier formato. El módulo estándar `string` contiene operaciones útiles para ajustar cadenas a un ancho de columna dado. Esto se discutirá en breve. El segundo modo es utilizar el operador `%` con una cadena como argumento izquierdo. El operador `%` interpreta el argumento izquierdo como una cadena de formato estilo `sprintf()`, que ha de ser aplicado sobre el argumento derecho para devolver la cadena resultante del formato.

Queda una cuestión, por supuesto: ¿Cómo convertir valores a cadenas? Afortunadamente, Python tiene modos de convertir cualquier valor a cadena: pasarle la función `repr()` o `str()`. Las comillas inversas (``) equivalen a `repr()`, pero se desaconseja su uso.

La función `str()` debe devolver representaciones legibles por un humano de los valores, mientras `repr()` debe generar representaciones legibles por el intérprete (o harán saltar un `SyntaxError` si no hay una sintaxis equivalente). Para los objetos que no tienen una representación particular para consumo humano, `str()` devolverá el mismo valor que `repr()`. Muchos valores, tales como números o estructuras como listas y diccionarios, tienen la misma representación en cualquiera de las dos funciones. En concreto, las cadenas y los números en coma flotante tienen representaciones diferentes.

He aquí algunos ejemplos:

```
>>> s = 'Hola, mundo.'
>>> str(s)
'Hola, mundo.'
>>> repr(s)
"'Hola, mundo.'"
>>> str(0.1)
'0.1'
>>> repr(0.1)
'0.10000000000000001'
```

```
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'El valor de x es ' + repr(x) + ', e y es ' + repr(y) + '...'
>>> print s
El valor de x es 32.5, e y es 40000...
>>> # Aplicar repr() a una cadena le añade comillas y barras invertidas:
... hola = 'hola, mundo\n'
>>> holas = repr(hola)
>>> print holas
'hola, mundo\n'
>>> # El argumento de repr() puede ser cualquier objeto de Python:
... repr((x, y, ('magro', 'huevos')))
"(32.5, 40000, ('magro', 'huevos'))"
>>> # Las comillas invertidas son prácticas en las sesiones interactivas:
... `x, y, ('magro', 'huevos')`
"(32.5, 40000, ('magro', 'huevos'))"
```

He aquí dos modos de escribir una tabla de cuadrados y cubos:

```
>>> for x in range(1, 11):
...     print repr(x).rjust(2), repr(x*x).rjust(3),
...     # Observa la coma final de la línea anterior
...     print repr(x*x*x).rjust(4)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
>>> for x in range(1,11):
...     print '%2d %3d %4d' % (x, x*x, x*x*x)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

Observa que se ha añadido un espacio entre columnas por el modo en que funciona `print`: siempre añade un espacio entre sus argumentos.

Este ejemplo utiliza el método `rjust()` de los objetos cadena, que ajusta a la derecha una cadena dada una anchura determinada, añadiendo espacios a la izquierda. Existen los métodos relacionados `ljust()` y `center()`. Estos métodos no escriben nada, sólo devuelven una cadena nueva. Si la cadena de entrada es demasiado larga, no la recortan, sino que la devuelven sin cambios. Se embrollará la salida, pero suele ser mejor que falsear los valores (si es preferible truncar la salida, siempre se puede agregar una operación de corte, como "`x.ljust(n)[:n]`").

Existe otro método, `zfill()`, que rellena una cadena numérica por la izquierda con ceros. Entiende de signos positivo y negativo:

```
>>> import string
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

Usar el operador `%` tiene este aspecto:

```
>>> import math
>>> print 'El valor de PI es aproximadamente %5.3f.' % math.pi
El valor de PI es aproximadamente 3.142.
```

Si hay más de un formato en la cadena, se ha de pasar una tupla como operando derecho, como aquí:

```
>>> tabla = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for nombre, telef in tabla.items():
...     print '%-10s ==> %10d' % (nombre, telef)
...
Jack       ==>       4098
Dcab       ==>       7678
Sjoerd     ==>       4127
```

La mayoría de los formatos funcionan como en C y exigen que se pase el tipo correcto. Sin embargo, de no hacerlo así, sólo se causa una excepción y no un volcado de memoria (o error de protección general). El formato `%s` es más relajado: Si el argumento correspondiente no es un objeto de cadena, se convierte a cadena mediante la función interna `str()`. Es posible pasar `*` para indicar la precisión o anchura como argumento (entero) aparte. No se puede utilizar los formatos de C `%n` ni `%p`.

Si tienes una cadena de formato realmente larga que no deseas dividir, sería un detalle hacer referencia a las variables por su nombre, en vez de por su posición. Esto se logra con `%(nombre) formato`, por ejemplo:

```
>>> tabla = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print 'Jack: %(Jack)d; Sjoerd: %(Sjoerd)d; Dcab: %(Dcab)d' % tabla
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Esto resulta particularmente práctico en combinación con la nueva función interna `vars()`, que devuelve un diccionario que contiene todas las variables locales.

7.2 Lectura y escritura de ficheros

`open()`, abrir, devuelve un objeto fichero. Se suele usar con dos argumentos: "`open(`*nombreFichero*`,` *modo*`)`".

```
>>> f=open('/tmp/fichTrabajo', 'w')
>>> print f
<open file '/tmp/fichTrabajo', mode 'w' at 80a0960>
```

El primer argumento es una cadena que contiene el nombre del fichero. El segundo argumento es otra cadena que contiene caracteres que describen el modo en que se va a utilizar el fichero. El *modo* puede ser `'r'`, cuando sólo se va a leer del fichero, `'w'`, si sólo se va a escribir (y si existe un fichero del mismo nombre se borra) o `'a'`, que abre el fichero para añadir datos. En el modo `'a'`, cualquier dato que se escriba en el fichero se añadirá al final de los datos existentes. El argumento de *modo* es opcional. Se supone `'r'` si se omite.

En Windows y Macintosh, al añadir `'b'` al modo, el fichero se abre en modo binario, por lo que existen modos como `'rb'`, `'wb'` y `'r+b'`. Windows distingue entre ficheros de texto y binarios: los caracteres de fin de línea de los ficheros de texto se alteran ligeramente de forma automática al leer o escribir datos. Esta modificación oculta no afecta en el caso de ficheros de texto ASCII, pero corrompe los ficheros de datos binarios, tales como ficheros JPEG o .EXE. Ten mucho cuidado de utilizar el modo binario al leer y escribir dichos ficheros (observa que el comportamiento preciso del modo de texto en Macintosh depende de la biblioteca C subyacente).

7.2.1 Métodos de los objetos fichero

El resto de los ejemplos de esta sección supondrán que se ha creado previamente un objeto fichero denominado `f`.

Para leer el contenido de un fichero, llama a `f.read(`*cantidad*`)`, que lee cierta cantidad de datos y los devuelve como cadena. *cantidad* es un argumento numérico opcional. Si se omite o es negativo, se leerá y devolverá el contenido completo del fichero. Es problema tuyo si el fichero tiene un tamaño descomunal. Si se incluye el argumento y es positivo, se leen y devuelven como máximo *cantidad* bytes. Si se había alcanzado el final de fichero, `f.read()` sólo devuelve una cadena vacía (`""`).

```
>>> f.read()
'Esto es el fichero completo.\n'
>>> f.read()
''
```

`f.readline()` lee una sola línea del fichero. Se deja un carácter de cambio de línea (`\n`) al final de la cadena, que se omite sólo en la última línea, siempre que el fichero no termine en un salto de línea. De este modo se consigue que el valor devuelto no sea ambiguo. Si `f.readline()` devuelve una cadena vacía, se ha alcanzado el final del fichero, mientras que una línea en blanco queda representada por `'\n'`, una cadena que sólo contiene un salto de línea.

```
>>> f.readline()
'La primera línea del fichero.\n'
>>> f.readline()
'La segunda línea del fichero\n'
>>> f.readline()
''
```

`f.readlines()` devuelve una lista que contiene todas las líneas de datos del fichero. Si se llama con un parámetro opcional *sizehint* (estimación de tamaño), lee los bytes indicados del fichero, sigue hasta completar una línea y devuelve la lista de líneas. Se suele utilizar esto para leer un fichero grande de una manera eficaz por líneas, pero sin tener que cargar en memoria el fichero entero. Todas las líneas devueltas están completas

```
>>> f.readlines()
['La primera línea del fichero.\n', 'La segunda línea del fichero\n']
```

`f.write(`*cadena*`)` escribe el contenido de *cadena* al fichero y devuelve `None`.

```
>>> f.write('Probando, probando\n')
```

Para escribir algo diferente de una cadena, ha de ser convertido a cadena primero:

```
>>> valor = ('la respuesta', 42)
>>> s = str(valor)
>>> f.write(s)
```

`f.tell()` devuelve un entero que indica la posición actual del objeto fichero dentro del fichero, medida en bytes desde el inicio del fichero. Para cambiar la posición del objeto fichero se usa "`f.seek(`*desplazamiento*`, `*desde_dónde*`)`". La posición se calcula sumando *desplazamiento* a un punto de referencia, seleccionado por el argumento *desde_dónde*. Un *desde_dónde* cero mide desde el inicio del fichero, 1 utiliza la posición actual y 2 utiliza el final del fichero como punto de referencia. *desde_dónde* se puede omitir, en cuyo caso se utiliza el valor cero y se mide desde el principio del fichero.

```
>>> f = open('/tmp/fichTrabajo', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)     # Ir al 6º byte del fichero
>>> f.read(1)
'5'
>>> f.seek(-3, 2) # Ir al 3er byte antes del final
>>> f.read(1)
'd'
```

Cuando termines de usar un fichero, llama a `f.close()` para cerrarlo y liberar los recursos del sistema que utilice el fichero. Tras llamar a `f.close()`, fracasará cualquier intento de usar el objeto fichero.

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

Los ficheros objeto tienen más métodos, como `isatty()` y `truncate()`, de uso menos frecuente. Consulta la Referencia de las bibliotecas si deseas ver una guía completa de los objetos fichero.

7.2.2 El módulo `pickle`

Es fácil escribir y leer cadenas de un fichero. Los números exigen un esfuerzo algo mayor, ya que el método `read()` sólo devuelve cadenas, que tendrán que pasarse a una función como `int()`, que toma una cadena como `'123'` y devuelve su valor numérico 123. Sin embargo, cuando se quiere guardar tipos de datos más complejos, como listas, diccionarios o instancias de clases, las cosas se complican bastante.

Mejor que hacer que los usuarios estén constantemente escribiendo y depurando código para guardar tipos de datos complejos, Python proporciona un módulo estándar llamado `pickle`[7.1]. Es un módulo asombroso que toma casi cualquier objeto de Python (¡hasta algunas formas de código Python!) y lo convierte a una representación de cadena. Este proceso se llama *estibado*. Reconstruir el objeto a partir de la representación en forma de cadena se llama *desestibado*. Entre el estibado y el

desestibado, la cadena que representa el objeto puede ser almacenada en un fichero, en memoria o transmitirse por una conexión de red a una máquina remota.

Si partes de un objeto `x` y un objeto fichero `f` previamente abierto para escritura, el modo más sencillo de estibar el objeto sólo tiene una línea de código:

```
pickle.dump(x, f)
```

Para realizar el proceso inverso, si `f` es un objeto fichero abierto para escritura:

```
x = pickle.load(f)
```

Existen otras variaciones de este tema, que se utilizan para estibar muchos objetos o si no se quiere escribir los datos estibados a un fichero. Se puede consultar la documentación completa de `pickle` en la *Referencia de las bibliotecas de Python*.

`pickle` es el método estándar para hacer que los objetos de Python se puedan almacenar y reutilizar en otros programas o futuras ejecuciones del mismo programa. El término técnico que identifica esto es objeto *persistente*. Como `pickle` se utiliza tanto, muchos autores que escriben extensiones a Python tienen cuidado de asegurarse de que los nuevos tipos de datos, tales como matrices, se estiben y desestiben de la manera adecuada.

Notas al pie

...../lib/module-pickle.html[7.1]

N. del T. *Pickle* significa conservar en formol o encurtir, pero lo voy a traducir como estibar. Estibar es colocar los bultos en una nave para prepararla para el viaje, lo que se adapta bastante bien a la función de `pickle`.

---

8. Errores y excepciones

Hasta ahora no habíamos más que mencionado los mensajes de error, pero si has probado los ejemplos puede que hayas visto algunos. Hay (al menos) dos tipos de errores diferenciables: los *errores de sintaxis* y las *excepciones*.

8.1 Errores de sintaxis

Los errores de sintaxis son la clase de queja más común del intérprete cuando todavía estás aprendiendo Python:

```
>>> while True print 'Hola mundo'
  File "<stdin>", line 1, in ?
    while True print 'Hola mundo'
                ^
SyntaxError: invalid syntax
```

El intérprete sintáctico repite la línea ofensiva y presenta una `flechita' que apunta al primer punto en que se ha detectado el error. La causa del error está (o al menos se ha detectado) en el símbolo *anterior* a la flecha: En este ejemplo, el error se detecta en la palabra clave `print`, porque faltan los dos puntos (":") antes de ésta. Se muestran el nombre del fichero y el número de línea para que sepas dónde buscar, si la entrada venía de un fichero.

8.2 Excepciones

Aun cuando una sentencia o expresión sea sintácticamente correcta, puede causar un error al intentar ejecutarla. Los errores que se detectan en la ejecución se llaman *excepciones* y no son mortales de necesidad, pronto va a aprender a gestionarlos desde programas en Python. Las excepciones no capturadas causan mensajes de error como el siguiente:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
>>> 4 + fiambre*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'fiambre' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

La última línea del mensaje de error indica qué ha pasado. Las excepciones pueden ser de diversos tipos, que se presentan como parte del mensaje: los tipos del ejemplo son `ZeroDivisionError`, `NameError` y `TypeError`. La cadena presentada como tipo de excepción es el nombre de la excepción interna que ha ocurrido. Esto es aplicable a todas las excepciones internas, pero no es necesariamente cierto para las excepciones definidas por el usuario (sin embargo, es una útil convención). Los nombres de excepciones estándar son identificadores internos (no palabras reservadas).

El resto de la línea proporciona detalles que dependen del tipo de excepción y de qué la causó.

La parte anterior del mensaje de error muestra el contexto donde ocurrió la excepción, en forma de trazado de la pila. En general, contiene un trazado que muestra las líneas de código fuente, aunque no mostrará las líneas leídas de la entrada estándar.

La *Referencia de las bibliotecas* enumera las excepciones internas y sus respectivos significados.

8.3 Gestión de excepciones

Es posible escribir programas que gestionen ciertas excepciones. Mira el siguiente ejemplo, que pide datos al usuario hasta que se introduce un entero válido, pero permite al usuario interrumpir la ejecución del programa (con `Control-C` u otra adecuada para el sistema operativo en cuestion); Observa que una interrupción generada por el usuario se señaliza haciendo saltar la excepcion `KeyboardInterrupt`.

```
>>> while True:
...     try:
...         x = int(raw_input("Introduce un número: "))
...         break
...     except ValueError:
...         print "¡Huy! No es un número. Prueba de nuevo..."
...
```

La sentencia `try` funciona de la siguiente manera:

- Primero se ejecuta la *cláusula try* (se ejecutan las sentencias entre `try` y `except`).
- Si no salta ninguna excepción, se omite la *cláusula except* y termina la ejecución de la sentencia `try`.
- Si salta una excepción durante la ejecución de la cláusula try, el resto de la cláusula se salta. Seguidamente, si su tipo concuerda con la excepción nombrada tras la palabra clave `except`, se ejecuta la cláusula except y la ejecución continúa tras la sentencia `try`.
- Si salta una excepción que no concuerda con la excepción nombrada en la cláusula except, se transmite a sentencias `try` anidadas exteriormente. Si no se encuentra un gestor de excepciones, se convierte en una *excepción imprevista* y termina la ejecución con un mensaje como el mostrado anteriormente.

Una sentencia `try` puede contener más de una cláusula except, para capturar diferentes excepciones. Nunca se ejecuta más de un gestor para una sola excepción. Los gestores sólo capturan excepciones que saltan en la cláusula try correspondiente, no en otros gestores de la misma sentencia `try`. Una cláusula try puede capturar más de una excepción, nombrándolas dentro de una lista:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

La última cláusula except puede no nombrar ninguna excepción, en cuyo caso hace de comodín y captura cualquier excepción. Se debe utilizar esto con mucha precaución, pues es muy fácil enmascarar un error de programación real de este modo. También se puede utilizar para mostrar un mensaje de error y relanzar la excepción (permitiendo de este modo que uno de los llamantes gestione la excepción a su vez):

```
import sys

try:
    f = open('mifichero.txt')
    s = f.readline()
    i = int(s.strip())
except IOError, (errno, strerror):
    print "Error de E/S(%s): %s" % (errno, strerror)
except ValueError:
    print "No ha sido posible covertir los datos a entero."
except:
    print "Error no contemplado:", sys.exc_info()[0]
    raise
```

La sentencia `try` ... `except` tiene una *cláusula else* opcional, que aparece tras las cláusulas except. Se utiliza para colocar código que se ejecuta si la cláusula try no hace saltar ninguna excepción. Por ejemplo:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'no se puede abrir', arg
    else:
        print arg, 'contiene', len(f.readlines()), 'líneas'
        f.close()
```

El uso de la cláusula `else` es mejor que añadir código adicional a la cláusula `try` porque evita que se capture accidentalmente una excepción que no fue lanzada por el código protegido por la sentencia `try` ... `except`.

Cuando salta una excepción, puede tener un valor asociado también conocido como el/los *argumento/s* de la excepción. Que el argumento aparezca o no y su tipo dependen del tipo de excepción.

La cláusula except puede especificar una variable tras el nombre de la excepción o lista de excepciones. La variable se asocia a una instancia de excepción y se almacenan los argumentos en `instance.args`. Por motivos prácticos, la instancia de la excepción define `__getitem__` y `__str__` para que los argumentos se puedan acceder o presentar sin tener que hacer referencia a `.args`.

```
>>> try:
...     raise Exception('magro', 'huevos')
... except Exception, inst:
...     print type(inst)     # la instancia de la excepción
...     print inst.args      # los argumentos guardados en .args
...     print inst           # __str__ permite presentar los args directa
...     x, y = inst          # __getitem__ permite desempaquetar los args
...     print 'x =', x
...     print 'y =', y
...
<type 'instance'>
```

```
('magro', 'huevos')
('magro', 'huevos')
x = magro
y = huevos
```

Si una excepción no capturada tiene argumento, se muestra como última parte (detalle) del mensaje de error.

Los gestores de excepciones no las gestionan sólo si saltan inmediatamente en la cláusula try, también si saltan en funciones llamadas (directa o indirectamente) dentro de la cláusula try. Por ejemplo:

```
>>> def esto_casca():
...     x = 1/0
...
>>> try:
...     esto_casca()
... except ZeroDivisionError, detalle:
...     print 'Gestión de errores:', detalle
...
Gestión de errores: integer division or modulo by zero
```

8.4 Hacer saltar excepciones

La sentencia `raise`, hacer saltar, permite que el programador fuerce la aparición de una excepción. Por ejemplo:

```
>>> raise NameError, 'MuyBuenas'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: MuyBuenas
```

El primer argumento para `raise` indica la excepción que debe saltar. El segundo argumento, opcional, especifica el argumento de la excepción. De manera alternativa, se puede escribir lo anterior como `raise NameError('MuyBuenas')`. Cualquiera de las dos formas funciona perfectamente, pero parece haber una preferencia estilística creciente por la segunda.

Si necesitas determinar si se lanzó una excepción pero no tienes intención de gestionarla, una manera más sencilla de la sentencia `raise` permite relanzar la excepción:

```
>>> try:
...     raise NameError, 'MuyBuenas'
... except NameError:
...     print '¡Ha saltado una excepción!'
...     raise
...
¡Ha saltado una excepción!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
  NameError: MuyBuenas
```

8.5 Excepciones definidas por el usuario

Los programas pueden nombrar sus propias excepciones creando una nueva clase de excepción. Suele ser adecuado que las excepciones deriven de la clase `Exception`, directa o indirectamente. Por ejemplo:

```
>>> class MiError(Exception):
...     def __init__(self, valor):
...         self.valor = valor
...     def __str__(self):
...         return `repr(self.valor)
...

>>> try:
...     raise raise MiError(2*2)
... except MiError, e:
...     print 'Ha saltado mi excepción, valor:', e.valor
...
Ha saltado mi excepción, valor: 4
>>> raise MiError, '¡Huy!'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MiError: '¡Huy!'
```

En este ejemplo, el `__init__` predeterminado de `Exception` se ha redefinido. El nuevo comportamiento simplemente crea el atributo *value*. Esto sustituye al comportamiento predefinido de crear el atributo *args*.

Se pueden definir clases de excepción que hagan cualquier cosa que pudiera hacer cualquier otra clase, pero suelen hacerse simples, limitándose a ofrecer atributos que permiten a los gestores del error extraer información sobre el error. Cuando se crea un módulo que puede hacer saltar excepciones, es una práctica común crear una clase base para todas las excepciones definidas en el módulo y crear subclases específicas para las diferentes condiciones de error:

```
class Error(Exception):
    """Clase base para todas las excepciones del módulo."""
    pass

class EntradaError(Error):
    """Excepción lanzada para errores de entrada.

    Atributos:
        expresion -- expresión de entrada en la que ocurrió el error
        mensaje -- explicación del error
    """

    def __init__(self, expresion, mensaje):
        self.expresion = expresion
        self.mensaje = mensaje

class TransicionError(Error):
    """Salta cuando una operación intenta una transición de estado
    no permitida.

    Atributos:
        actual -- estado inicial de la transición
       nuevo -- nuevo estado deseado
```

```
        mensaje -- explicación de por qué no se permite la transición
    """

    def __init__(self, actual, nuevo, mensaje):
        self.actual = actual
        self.nuevo = nuevo
        self.mensaje = mensaje
```

La mayoría de las excepciones se nombran terminando en ``Error,'' tal como las excepciones estándar.

Muchos módulos estándar definen sus propias excepciones para informar de errores que pueden darse en las funciones que definen. Hay más información sobre clases en el capítulo 9, ``Clases.''

8.6 Definir acciones de limpieza

La sentencia `try` tiene otra cláusula opcional cuya intención es definir acciones de limpieza que se han de ejecutar en cualquier circunstancia. Por ejemplo:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print '¡Adiós, mundo cruel!'
...
¡Adiós, mundo cruel!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
KeyboardInterrupt
```

La cláusula *finally* (finalmente) se ejecuta tanto si ha saltado una excepción dentro de la cláusula try como si no. Si ocurre una excepción, vuelve a saltar tras la ejecución de la cláusula finally. También se ejecuta la cláusula finally ``a la salida'' si se abandona la sentencia `try` mediante `break` o `return`.

El código de la cláusula finally es útil para liberar recursos externos (como ficheros o conexiones de red), independientemente de si el uso del recurso fue satisfactorio o no.

Una sentencia `try` debe tener una o más cláusulas except o una cláusula finally, pero no las dos (ya que no estaría claro cuál de las cláusulas ejecutar).

Release 2.4.1a0, documentation updated on septiembre 11, 2005.
Consultar en Acerca de este documento... información para sugerir cambios.

9. Clases

El mecanismo de clases de Python añade clases al lenguaje con un mínimo de sintaxis y semántica nueva. Es una mezcla de los mecanismos de clase de C++ y Modula-3. Como en los módulos, las clases de Python no ponen una barrera absoluta entre la definición y el usuario, sino que más bien se fían de la buena educación del usuario para no ``invadir la definición''. Se mantienen las características más importantes con plena potencia. El mecanismo de herencia de clases permite múltiples clases base, una clase derivada puede redefinir cualquier método de sus clases base y un método puede llamar a un método de una clase base con el mismo nombre. Los objetos pueden contener una cantidad arbitraria de datos privados.

En terminología C++, todos los miembros de la clase (datos incluidos) son *públicos* y todas las funciones miembro son *virtuales*. No hay constructores ni destructores especiales. Como en Modula-3, no existen abreviaturas para hacer referencia a los miembros del objeto desde sus propios métodos. La función método se declara con un primer argumento explícito que representa al objeto y que se proporciona implícitamente al llamar a la función. Como en Smalltalk, las clases son ellas mismas objetos, aunque en un sentido más amplio de la palabra: en Python, todos los tipos de datos son objetos. Esto proporciona la semántica para importar y renombrar. A diferencia de C++ o en Modula3, los tipos internos pueden ser la clase base para extensiones del usuario. Además, como en C++ y al contrario de Modula-3, la mayoría de operadores internos con sintaxis especial (operadores aritméticos, índices, etc.) se pueden redefinir en las clases.

9.1 Unas palabras sobre la terminología

A falta de una terminología universalmente aceptada para hablar de clases, haré uso ocasional de términos de Smalltalk y C++ (haría uso de términos de Modula-3, ya que la semántica orientada al objeto es más cercana a Python que la de C++, pero no espero que muchos lectores la dominen).

Los objetos tienen individualidad. Se pueden asociar múltiples nombres (y en diferentes ámbitos) al mismo objeto, lo que se conoce como ``generar alias'' en otros lenguajes. Esto no se aprecia a primera vista en Python y no hace falta tenerlo en cuenta cuando se trata con tipos inmutables (números, cadenas, tuplas...). Sin embargo los alias tienen un efecto (¡buscado!) en la semántica del código Python que involucra los objetos mutables, como listas, diccionarios y la mayoría de los tipos que representan entidades externas al programa (ficheros, ventanas...). Se suele usar en beneficio del programa, ya que los alias se comportan como punteros en algunos aspectos. Por ejemplo, pasar un objeto es poco costoso, ya que la implementación sólo pasa un puntero. Si la función modifica el objeto que pasa como argumento, el que llama a la función verá los cambios. De este modo se elimina la necesidad de tener los dos mecanismos de traspaso de argumentos de Pascal.

9.2 Ámbitos y espacios nominales en Python

Antes de presentar las clases, debo contar algo sobre las reglas de alcance de Python. Las

definiciones de clases realizan truquitos con los espacios nominales y se necesita saber cómo funcionan los alcances y espacios nominales para comprender plenamente lo que ocurre. Incidentalmente, el conocimiento de este tema es útil para cualquier programador en Python avanzado.

Empecemos con unas definiciones.

Un *espacio nominal* es una correspondencia entre nombres y objetos. La mayoría de los espacios de nombres se implementan en la actualidad como diccionarios, pero eso no se nota en modo alguno (salvo por el rendimiento) y puede cambiar en el futuro. Ejemplos de espacios nominales son:

- El conjunto de nombres internos (funciones como `abs()` y las excepciones internas).
- Los nombres globales de un módulo.
- Los nombres locales dentro de una llamada a función.

En cierto sentido, el conjunto de atributos de un objeto también forma un espacio nominal. Lo que hay que tener claro de los espacios nominales es que no existe absolutamente ninguna relación entre los nombres de diferentes espacios. Por ejemplo, dos módulos pueden definir una función ``maximizar'' sin posibilidad de confusión; los usuarios de los módulos deben preceder el nombre con el nombre del módulo.

Por cierto, utilizo la palabra *atributo* para referirme a cualquier nombre que venga detrás de un punto; por ejemplo, en la expresión `z.real`, `real` es un atributo del objeto `z`. Hablando estrictamente, las referencias a nombres en un módulo son referencias a atributos. En la expresión `nombremod.nombrefunc`, `nombremod` es un objeto módulo y `nombrefunc` es atributo suyo. En este caso, resulta que existe una correspondencia directa entre los atributos del módulo y los nombres globales definidos en el módulo: ¡comparten el mismo espacio nominal[9.1]!

Los atributos pueden ser de sólo lectura o de lectura/escritura. En este caso, es posible asignar valores a los atributos. Los atributos de los módulos son de lectura/escritura: Se puede escribir "`nombremod.respuesta = 42`". Los atributos de lectura/escritura se pueden borrar con la sentencia `del`, por ejemplo: "`del nombremod.respuesta`" eliminará el atributo `respuesta` del objeto denotado por `nombremod`.

Los espacios nominales se crean en diferentes momentos y tienen tiempos de vida diferentes. El espacio nominal que contiene los nombres internos se crea al arrancar el intérprete de Python y nunca se borra. El espacio nominal global de un módulo se crea al leer la definición del módulo. Normalmente, los espacios nominales de los módulos también duran hasta que se sale del intérprete. Las sentencias ejecutadas por el nivel superior de llamadas, leídas desde un guion o interactivamente, se consideran parte de un módulo denominado `__main__`, así que tienen su propio espacio nominal (los nombres internos también residen en un módulo, llamado `__builtin__`).

El espacio nominal local de una función se crea cuando se llama a la función y se borra al salir de la función, por una sentencia "`return`"o si salta una excepción que la función no captura (en realidad, lo más parecido a lo que ocurre es el olvido). Por supuesto, las llamadas recursivas generan cada una su propio espacio nominal.

Un *ámbito* es una región textual de un programa Python en que el espacio nominal es directamente accesible. ``Directamente accesible'' significa en este contexto una referencia sin calificar (sin puntos) que intenta encontrar un nombre dentro de un espacio nominal.

A pesar de que los ámbitos se determinan estáticamente, se utilizan dinámicamente. En cualquier momento de la ejecución, hay al menos tres ámbitos anidados cuyos espacios nominales están

directamente disponibles: El ámbito interno, el primero en el que se busca, contiene los nombres locales; los espacios nominales de las posibles funciones llamadas hasta llegar al ámbito actual, en los que se busca empezando por el ámbito llamante inmediato hasta los más externos; el ámbito intermedio, el siguiente en la búsqueda, contiene los nombres globales del módulo actual; y el ámbito externo (el último para la búsqueda) es el espacio nominal que contiene los nombres internos (__builtin__).

Si se declara un nombre como global, todas las referencias y asignaciones afectan directamente al ámbito de enmedio, que contiene los nombres globales del módulo. De otro modo, todas las variables que se encuentren fuera del ámbito interno son de sólo lectura.

Normalmente, el ámbito local hace referencia a los nombres locales de la función en curso (textualmente). Fuera de las funciones, el ámbito local hace referencia al mismo espacio nominal que el ámbito global: el espacio nominal del módulo. Las definiciones de clases colocan otro espacio nominal en el ámbito local.

Es importante darse cuenta de que los ámbitos se determinan textualmente: El ámbito global de una función definida en un módulo es el espacio nominal de este módulo, sin importar desde dónde o con qué alias se haya llamado a la función. Por otra parte, la búsqueda real de nombres se lleva a cabo dinámicamente, en tiempo de ejecución. Sin embargo, la definición del lenguaje tiende a la resolución estática de los nombres, así que ¡no te fíes de la resolución dinámica de los nombres! De hecho ya se determinan estáticamente las variables locales.

Un asunto especial de Python es que las asignaciones siempre van al ámbito más interno. Las asignaciones no copian datos, simplemente enlazan nombres a objetos. Lo mismo vale para los borrados: la sentencia "`del x`" elimina el enlace de `x` del espacio nominal al que hace referencia el ámbito local. De hecho, todas las operaciones que introducen nombres nuevos utilizan el ámbito local. Particularmente, las sentencias import y las definiciones de funciones asocian el nombre del módulo o función al ámbito local. Se puede utilizar la sentencia `global` para indicar que ciertas variables residen en el ámbito global.

9.3 Un primer vistazo a las clases

Las clases introducen una pizca de sintaxis nueva, tres tipos de objeto nuevos y algo de semántica nueva.

9.3.1 Sintaxis de definición de clases

La forma más simple de definición de clase tiene este aspecto:

```
class nombreClase:
    <sentencia-1>
    .
    .
    .
    <sentencia-N>
```

Las definiciones de clases, como las definiciones de funciones (sentencias `def`) deben ejecutarse para tener efecto (es perfectamente correcto colocar una definición de clase en una rama de una sentencia `if` o dentro de una función).

En la práctica, las sentencias de dentro de una definición de clase serán definiciones de funciones, pero se permite otro tipo de sentencias, lo que resulta útil en algunos casos, ya veremos esto. Las definiciones de funciones interiores a la clase suelen tener una lista de argumentos un poco especial, dictada por las convenciones de llamada a método. Esto también se explica más adelante.

Cuando se entra en una definición de clase, se genera un nuevo espacio nominal, que se utiliza como ámbito local; así que todas las asignaciones a variables locales caen dentro de este nuevo espacio nominal. En particular, las definiciones de funciones enlazan aquí el nombre de la nueva función.

Cuando se abandona una definición de clase de manera normal (se ejecuta la última línea de su código), se crea un *objeto de clase*. Es, sencillamente, un envoltorio del contenido del espacio nominal creado por la definición de la clase. Se verá con más detalle en la siguiente sección. El ámbito local original (el que estaba activo cuando se entró en la definición de clase) se reinstancia y el objeto clase se enlaza con el nombre de clase dado en la cabecera de la función (en el ejemplo `nombreClase`).

9.3.2 Objetos clase

Los objetos de clase soportan dos tipos de operaciones: referencia a atributos e instanciación.

Las referencias a atributos utilizan la sintaxis estándar que se utiliza para todas las referencias a atributos en Python: `obj.nombre`. Los nombres de atributos válidos son todos los nombres que estaban en el espacio nominal de la clase cuando fue creada la clase. Por lo tanto, si la definición de la clase tiene este aspecto:

```
class MiClase:
    "Simple clase de ejemplo"
    i = 12345
    def f(self):
        return 'hola, mundo'
```

`MiClase.i` y `MiClase.f` son referencias a atributos válidas, que devuelven un entero y un objeto método, respectivamente. También se puede asignar valores a los atributos de una clase; puedes cambiar el valor de `MiClase.i` con una asignación. `__doc__` es también un atributo válido, que devuelve la cadena de documentación que corresponde a la clase: `"Simple clase de ejemplo"`.

La *instanciación* de clases utiliza notación de función. Basta con imaginar que el objeto clase es una función sin parámetros que devuelve una instancia nueva de la clase. Por ejemplo, siguiendo con el ejemplo anterior:

```
x = MiClase()
```

crea una nueva *instancia* de la clase y la asigna a la variable local `x`. La operación de instanciación (``la llamada" a un objeto clase) genera un objeto vacío. Muchas clases prefieren generar los objetos en un estado inicial conocido. Por ello, una clase puede definir un método especial denominado `__init__()`, así:

```
    def __init__(self):
        self.vaciar()
```

Cuando una clase define un método `__init__()`, la instanciación de clases llama automáticamente a `__init__()` para la instancia de clase recién creada. Así que, en el ejemplo de la `Bolsa`, se puede

obtener una instancia de clase inicializada nueva mediante:

```
x = Bolsa()
```

Por supuesto, el método `__init__()` podría tener argumentos que le añadirían flexibilidad. En tal caso, los argumentos proporcionados al operador de instanciación de clase se pasan a `__init__()`. Por ejemplo,

```
>>> class Complejo:
...     def __init__(self, parteReal, parteImaginaria):
...         self.r = parteReal
...         self.i = parteImaginaria
...
>>> x = Complejo(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

9.3.3 Objetos instancia

¿Qué se puede hacer con los objetos instancia? Las únicas operaciones que entienden son las referencias a atributos. Hay dos tipos de nombres de atributo válidos, los atributos de datos y los métodos.

Los *atributos de datos* corresponden a las ``variables de instancia'' de Smalltalk o a los ``miembros dato'' de C++. No es necesario declarar los atributos de datos. Como las variables locales, aparecen por arte de magia la primera vez que se les asigna un valor. Por ejemplo, si `x` es una instancia de la clase `MiClase` creada anteriormente, el código siguiente mostrará el valor `16` sin dejar rastro:

```
x.contador = 1
while x.contador < 10:
    x.contador = x.contador * 2
print x.contador
del x.contador
```

El otro tipo de referencia a atributo de los objetos instancia son los *métodos*. Un método es una función que ``pertenece a'' un objeto. En Python, el término método no se limita a las instancias de una clase, ya que otros tipos de objeto pueden tener métodos también. Por ejemplo, los objetos de tipo lista tienen métodos llamados append, insert, remove, sort, etc. Sin embargo, vamos a utilizar ahora el término exclusivamente para referirnos a los métodos de objetos instancia de una clase, salvo que se indique lo contrario.

Los nombres válidos de métodos de un objeto instancia dependen de su clase. Por definición, todos los atributos de una clase que son objetos función definen los métodos correspondientes de sus instancias. Así que, en nuestro ejemplo, `x.f` es una referencia a método correcta, ya que `MiClase.f` es una función, pero `x.i` no lo es, ya que `MiClase.i` no es una función. Pero `x.f` no es lo mismo que `MiClase.f` - es un *objeto método*, no un objeto función.

9.3.4 Objetos método

Normalmente, se llama a un método de manera inmediata, por ejemplo:

```
x.f()
```

En nuestro ejemplo, esto devuelve la cadena `'hola, mundo'`. Sin embargo, no es necesario llamar a un método inmediatamente: `x.f` es un objeto método y se puede almacenar y recuperar más tarde, por ejemplo:

```
xf = x.f
while True:
    print xf()
```

mostrará "`hola, mundo`" hasta que las ranas críen pelo.

¿Qué ocurre exactamente cuando se llama a un método? Habrás observado que `x.f()` fue invocado sin argumento, aunque la definición del método `f` especificaba un argumento. ¿Qué le ha pasado al argumento? Desde luego, Python hace saltar una excepción cuando se llama a una función que necesita un argumento sin especificar ninguno (aunque no se utilice)...

En realidad, te puedes imaginar la respuesta: Lo que tienen de especial los métodos es que el objeto que los llama se pasa como primer argumento de la función. En nuestro ejemplo, la llamada `x.f()` es totalmente equivalente a `MiClase.f(x)`. En general, llamar a un método con una lista de argumentos es equivalente a llamar a la función correspondiente con la lista de argumentos resultante de insertar el objeto del método al principio de la lista de argumentos original.

Si todavía no entiendes cómo funcionan los métodos, igual te aclara las cosas un vistazo a la implementación. Cuando se hace referencia a un atributo de una instancia que no es un atributo de datos, se busca en su clase. Si el nombre denota un atributo de clase válido que resulta ser un objeto función, se crea un objeto método empaquetando juntos (punteros hacia) el objeto instancia y el objeto función recién encontrado en un objeto abstracto: el objeto método. Cuando se llama al objeto método con una lista de argumentos, se desempaqueta de nuevo, se construye una nueva lista de argumentos a partir del objeto instancia y la lista de argumentos original y se llama al objeto función con la nueva lista de argumentos.

9.4 Cajón de sastre

Los atributos de datos se tienen en cuenta en lugar de los atributos método con el mismo nombre. Para evitar conflictos nominales accidentales, que podrían causar errores difíciles de rastrear en programas grandes, conviene utilizar algún tipo de convención que minimice la probabilidad de conflictos. Las convenciones incluyen poner en mayúsculas los nombres de métodos, preceder los nombre de atributos de datos con una pequeña cadena única (o sólo un guion bajo) o usar verbos para los métodos y nombres para los atributos de datos.

Los métodos pueden hacer referencia a atributos de datos tanto como los usuarios normales (los clientes) de un objeto. En otras palabras, no es posible usar las clases para implementar tipos de datos abstractos puros. De hecho, no hay nada en Python para posibilitar la ocultación de datos, todo se basa en convenciones (por otra parte, la implementación de Python escrita en C puede ocultar completamente los detalles de implementación y el control de acceso a un objeto si es necesario, lo que pueden hacer también las extensiones a Python escritas en C).

Los clientes deben utilizar los atributos de datos con cuidado. Los clientes pueden embrollar invariantes mantenidas por los métodos, chafándolas con sus atributos de datos. Observa que los clientes pueden añadir atributos de datos propios a una instancia de objeto sin afectar a la validez de

los métodos, siempre que se eviten los conflictos de nombres. Nuevamente, ser coherente en los nombres puede ahorrarnos un montón de dolores de cabeza.

No hay un atajo para hacer referencia a los atributos dato (¡ni a otros métodos!) desde los métodos. Encuentro que esto, en realidad, favorece la legibilidad de los métodos, porque así no hay manera de confundir las variables locales y las variables de la instancia cuando se repasa un método.

Por convención, el primer argumento de un método se suele llamar `self` (yo mismo). No es más que una convención, el nombre `self` no le dice nada a Python (sin embargo, no seguir esta convención hace que tu código sea menos legible por otros programadores y no sería extraño que haya *navegadores de la jerarquía de clases* que suponen que la sigues).

Cualquier objeto función que es atributo de una clase define un método para las instancias de esa clase. No es necesario que la definición de la función esté textualmente encerrada en la definición de la clase. Asignar un objeto función a una variable local de la clase también vale. Por ejemplo:

```
# Función definida fuera de la clase
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1
    def g(self):
        return 'hola, mundo'
    h = g
```

Ahora `f`, `g` y `h` son atributos de la clase `C` que hacen referencia a objetos función, por lo que los tres son métodos de las instancias de la clase `C`, siendo `h` exactamente equivalente a `g`. Observa que esta práctica suele valer sólo para confundir al lector del programa.

Los métodos pueden llamar a otros métodos utilizando los atributos método del argumento `self`:

```
class Bolsa:
    def __init__(self):
        self.datos = []
    def agregar(self, x):
        self.datos.append(x)
    def agregarDosVeces(self, x):
        self.agregar(x)
        self.agregar(x)
```

Los métodos puede hacer referencia a los nombres globales del mismo modo que las funciones normales. El ámbito global asociado a un método en un método es el módulo que contiene la definición de la clase (la clase en sí nunca se utiliza como ámbito global). Aunque es raro encontrar un buen motivo para usar un dato global en un método, hay bastantes usos legítimos del ámbito global: de momento, los métodos pueden usar las funciones y los módulos importados al ámbito global, al igual que las funciones y las clases definidas en él. Normalmente, la clase que contiene el método está definida en este ámbito global. En la siguiente sección encontraremos algunas buenas razones para que un método haga referencia a su propia clase.

9.5 Herencia

Por supuesto, una característica de un lenguaje no sería digna del nombre ``clase'' si no aportara

herencia. La sintaxis de una definición de clase derivada tiene este aspecto:

```
class nombreClaseDerivada(nombreClaseBase):
    <sentencia-1>
    .
    .
    .
    <sentencia-N>
```

El nombre `nombreClaseBase` debe estar definido en un ámbito que contenga la definición de la clase derivada. En lugar de un nombre de clase base, se permite poner una expresión. Esto es útil cuando la clase base está definida en otro módulo,

```
class nombreClaseDerivada(nombreMod.nombreClaseBase):
```

La ejecución de la definición de una clase derivada se lleva a cabo del mismo modo que la clase base. Cuando se construye el objeto de la clase, se recuerda la clase base. Esto se utiliza para resolver referencias a atributos: si no se encuentra un atributo solicitado en la clase, se busca en la clase base. Esta regla se aplica recursivamente si la clase base es a su vez derivada.

No hay nada especial sobre la instanciación de las clases derivadas: `nombreClaseDerivada()` crea una nueva instancia de la clase. Las referencias a métodos se resuelven de la siguiente manera: Se busca el atributo de la clase correspondiente, descendiendo por la cadena de clases base, si es necesario, y la referencia a método es correcta si de este modo se obtiene un objeto función.

Las clases derivadas pueden redefinir métodos de sus clases base. Como los métodos no tienen privilegios especiales al llamar a otros métodos del mismo objeto, un método de una clase base que llama a otro método definido en la misma clase base puede acabar llamando a un método de una clase derivada que lo redefina (para los programadores de C++: todos los métodos en Python son `virtuales`).

Puede que una redefinición de método en una clase derivada quiera ampliar, en lugar de reemplazar, el método de la clase base del mismo nombre. Existe un modo sencillo de llamar al método de la clase base directamente: simplemente, utilizar "`nombreClaseBase.nombreMétodo(self, argumentos)`". Esto también les vale a los clientes, en ciertas ocasiones (observa que esto sólo funciona si la clase base está definida o se ha importado directamente en el ámbito global).

9.5.1 Herencia múltiple

Python también aporta una forma limitada de herencia múltiple. Una definición de clase con múltiples clases base tiene este aspecto:

```
class nombreClaseDerivada(Base1, Base2, Base3):
    <sentencia-1>
    .
    .
    .
    <sentencia-N>
```

La única regla necesaria para explicar la semántica es la regla de resolución utilizada para las referencias a atributos de la clase. Se busca primero por profundidad y luego de izquierda a derecha. De este modo, si no se encuentra un atributo en `nombreClaseDerivada`, se busca en `Base1`, en las clases base de `Base1` y, si no se encuentra, en `Base2`, en las clases base de ésta y así sucesivamente.

Para algunos parece más natural buscar en `Base2` y en `Base3` antes que en las clases base de `Base1`. Sin embargo, esto exigiría conocer si un atributo particular de `Base1` está definido realmente en `Base1` en una de sus clases base, antes de imaginarse las consecuencias de un conflicto de nombres con un atributo de la `Base2`. La regla de buscar primero por profundidad evita la diferencia entre atributos de la `Base1` directos y heredados.

Queda claro que el uso indiscriminado de la herencia múltiple hace del mantenimiento una pesadilla, dada la confianza de Python en las convenciones para evitar los conflictos de nombre accidentales. Un problema bien conocido de la herencia múltiple es el de una clase derivada de dos clases que resulta que tienen una clase base común. A pesar de que resulta sencillo, en este caso, figurarse qué ocurre (la instancia tendrá un sola copia de las ``variables de la instancia" o atributos de datos utilizada por la base común), no queda clara la utilidad de este tipo de prácticas.

9.6 Variables privadas

Se pueden utilizar, de manera limitada, identificadores privados de la clase. Cualquier identificador de la forma `__fiambre` (al menos dos guiones bajos iniciales, no más de un guion bajo final) se reemplaza textualmente con `_nombreClase__fiambre`, donde `nombreClase` es la clase en curso, eliminando los guiones bajos iniciales. Esta reescritura se realiza sin tener en cuenta la posición sintáctica del identificador, por lo que se puede utilizar para definir, de manera privada, variables de clase e instancia, métodos y variables globales. También sirve para almacenar variables de instancia privadas de esta clase en instancias de *otras* clases. Puede que se recorten los nombres cuando el nombre reescrito tendría más de 255 caracteres. Fuera de las clases o cuando el nombre de la clase consta sólo de guiones bajos, no se reescriben los nombres.

La reescritura de nombres pretende dar a las clases un modo sencillo de definir métodos y variables de instancia ``privados", sin tener que preocuparse por las variables de instancia definidas por las clases derivadas ni guarrear con las variables de instancia por el código externo a la clase. Observa que las reglas de reescritura se han diseñado sobre todo para evitar accidentes; aún es posible, con el suficiente empeño, leer o modificar una variable considerada privada. Esto puede ser útil en casos especiales, como el depurador, por lo que no se ha cerrado esta puerta falsa. Hay un pequeño fallo: la derivación de una clase con el mismo nombre que su clase base hace posible el uso de las variables privadas de la clase base.

Observa que el código pasado a `exec`, `eval()` o `evalfile()` no considera el nombre de la clase llamante la clase actual. Es similar al efecto de la sentencia `global`, cuyo efecto está, de igual manera, restringido al código de un fichero. Se aplica la misma restricción a `getattr()`, `setattr()`, `delattr()` y también cuando se hace referencia a `__dict__` directamente.

9.7 Remates

A veces es útil tener un tipo de dato similar al ``record" de Pascal o a la ``struct" de C, que reúnan unos cuantos datos con nombre. Para realizar esto, se puede usar una definición de clase vacía:

```
class Empleado:
    pass

juan = Empleado() # Creación de una ficha de empleado vacía
```

```
# Rellenamos los campos de la ficha
juan.nombre = 'Juan Pérez'
juan.departamento = 'Centro de cálculo'
juan.sueldo = 1000
```

Si hay código en Python que espere recibir un tipo abstracto de datos concreto, es posible pasarle una clase que emule los métodos de este tipo de dato, en lugar de la clase genuina. Por ejemplo, si disponemos de una función que da formato a unos datos de un objeto fichero, podemos definir una clase con los métodos `read()` y `readline()` que tome los datos de una cadena de almacenamiento temporal y pasar dicha clase como argumento.

Los objetos de métodos de instancia también tienen atributos: `m.im_self` es el objeto que tiene el método `m` y `m.im_func` es el objeto función correspondiente al método.

9.7.1 Las excepciones también son clases

Las excepciones definidas por usuario se identifican mediante clases. Utilizando este mecanismo, es posible crear jerarquías ampliables de excepciones.

Hay dos formas válidas nuevas de sentencia `raise`:

```
raise Clase, instancia

raise instancia
```

En la primera forma, `instancia` debe ser una instancia de `Clase` o de una clase derivada de ella. La segunda forma es un atajo de:

```
raise instancia.__class__, instancia
```

Una clase de una cláusula except captura una excepción si es de la misma clase que la excepción que ha saltado o es de una clase base de ella (al revés no; una clase derivada no captura ninguna de sus clases base). Por ejemplo, el código a continuación mostrará B, C, D, en ese orden:

```
class B:
    pass
class C(B):
    pass
class D(C):
    pass

for c in [B, C, D]:
    try:
        raise c()
    except D:
        print "D"
    except C:
        print "C"
    except B:
        print "B"
```

Observa que si invertimos las cláusulas ("`except B`" primero), se mostraría B, B, B, ya que la primera cláusula captura todas las excepciones, por ser clase base de todas ellas.

Cuando se presenta un mensaje de error para una excepción sin capturar, se muestra el nombre de la clase, dos puntos, un espacio y, por último, la instancia convertida a cadena mediante la función interna `str()`.

9.8 Iteradores

Ya debes de haberte dado cuenta de que la mayoría de los objetos contenedores se pueden recorrer con una sentencia `for`:

```
for elemento in [1, 2, 3]:
    print elemento
for element in (1, 2, 3):
    print elemento
for clave in {'uno':1, 'dos':2}:
    print clave
for car in "123":
    print car
for linea in open("myfich.txt"):
    print linea
```

Este estilo de acceso es claro, conciso y práctico. El uso de iteradores impregna y unifica Python. En la trastienda, la sentencia `for` llama a `iter()` sobre el objeto contenido. La función devuelve un objeto iterador que define el método `next()` que accede a los elementos del contenedor de uno a uno. Cuando no quedan más elementos, `next()` lanza una excepción `StopIteration` que indica al bucle `for` que ha de terminar. Este ejemplo muestra cómo funciona todo esto:

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> it.next()
'a'
>>> it.next()
'b'
>>> it.next()
'c'
>>> it.next()

Traceback (most recent call last):
  File "<pyshell#6>", line 1, in -toplevel-
    it.next()
StopIteration
```

Después de ver los mecanismos del protocolo iterador, es fácil añadir comportamiento de iterador a tus propias clases. Define un método `__iter__()` que devuelva un objeto con un método `next()`. Si la clase define `next()`, `__iter__()` puede simplemente devolver `self`:

```
class Delreves:
    "Iterador para recorrer una secuencia en orden inverso"
    def __init__(self, data):
        self.datos = datos
        self.indice = len(datos)
    def __iter__(self):
        return self
```

```
        def next(self):
            if self.index == 0:
                raise StopIteration
            self.indice = self.indice - 1
            return self.data[self.indice]

    >>> for chr in Delreves('magro'):
    ...     print car
    ...
    o
    r
    g
    a
    m
```

9.9 Generadores

Los generadores son una herramienta simple y potente para crear iteradores. Se escriben como funciones normales, pero usan la sentencia `yield` siempre que quieren devolver datos. Cada vez que se llama a `next()`, el generador reanuda la ejecución donde se quedó (recordando todos los valores de los datos y la última sentencia en ejecutarse). Este ejemplo muestra que es inmediato crear generadores:

```
    def delreves(data):
        for indice in range(len(datos)-1, -1, -1):
            yield datos[indice]

    >>> for car in delreves('Wanda'):
    ...     print car
    ...
    a
    d
    n
    a
    W
```

Cualquier cosa que se pueda hacer con generadores se puede hacer también con iteradores basados en clases, como se describió en la anterior sección. Lo que hace que los generadores sean tan compactos es que su métodos `__iter__()` y `next()` se crean automáticamente.

Otra característica clave es que las variables locales y el estado de ejecución se guardan automáticamente entre llamadas. Esto hizo que la función fuese más fácil de escribir y mucho más clara que la solución utilizando variables de instancia como `self.indice` y `self.datos`.

Además de la creación de métodos y gestión del estado automáticos, cuando terminan los generadores, hacen saltar la excepción `StopIteration` automáticamente. La combinación de estas características hace que la creación de iteradores no lleve más esfuerzo que el de crear una función normal.

9.10 Expresiones generadoras

Hay generadores simples que se pueden codificar brevemente como expresiones utilizando una sintaxis similar a las listas autodefinidas, con paréntesis en lugar de corchetes. Estas expresiones se utilizan cuando el generador lo utiliza de inmediato la función que lo contiene. Las expresiones generadoras son más compactas pero menos versátiles que las funciones generadoras completas y tienden a ser menos agresivas con la memoria que las listas autodefinidas.

Ejemplos:

```
>>> sum(i*i for i in range(10))                 # suma de los cuadrados
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))         # producto escalar
260

>>> from math import pi, sin
>>> tabla_senos = dict((x, sin(x*pi/180)) for x in range(0, 91))

>>> palabras_sin_repes = set(palabra for linea in pagina  for palabra in

>>> mejorOpositor = max((opositor.nota, opositor.nombre) for opositor in

>>> datos = 'golf'
>>> list(datos[i] for i in range(len(datos)-1,-1,-1))
['f', 'l', 'o', 'g']
```

Notas al pie

... nominal[9.1]

Excepto en una cosa. Los objetos de módulo tienen un atributo de sólo lectura secreto llamado `__dict__` que devuelve el diccionario utilizado para implementar el espacio nominal del módulo. El nombre `__dict__` es un atributo, pero no un nombre global. Evidentemente, al utilizar esto se transgrede la abstracción de la implementación del espacio nominal, por lo que su uso debe restringirse a herramientas especiales, como depuradores póstumos.

---

10. Viaje rápido por la biblioteca estándar

10.1 Interfaz con el sistema operativo

El módulo `os` proporciona funciones para interactuar con el sistema operativo:

```
>>> import os
>>> os.system('time 0:02')
0
>>> os.getcwd()      # Devuelve el directorio de trabajo actual
'C:\\Python24'
>>> os.chdir('/server/accesslogs')
```

Asegúrate de usar el estilo "`import os`" en lugar de "`from os import *`". Esto evitará que `os.open()` oculte la función iterna `open()`, que tiene un funcionamiento muy diferente.

Las funciones internas `dir()` y `help()` son útiles como ayudantes interactivos para trabajar con módulos grandes, como `os`:

```
>>> import os
>>> dir(os)
<devuelve una lista de todas las funciones del módulo>
>>> help(os)
<devuelve un extenso manual creado a partir de las cadenas de documentac
```

Para tareas de gestión de ficheros y directorios diarias, el módulo `shutil` proporciona una interfaz de nivel más alto, más fácil de usar:

```
>>> import shutil
>>> shutil.copyfile('datos.db', 'copia.db')
>>> shutil.move('/build/ejecutables', 'dirinstalacion')
```

10.2 Comodines de ficheros

El módulo `glob` proporciona una función para crear listas de ficheros a partir de búsquedas con comodines por directorios:

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

10.3 Argumentos de la línea de órdenes

Los guiones de utillería usuales necesitan procesar argumentos de la línea de órdenes a menudo. Estos argumentos se guardan en el atributo *argv* del módulo sys como una lista. Por ejemplo, éste es el resultado de ejecutar "`python demo.py un dos tres`" en la línea de órdenes:

```
>>> import sys
>>> print sys.argv
['demo.py', 'un', 'dos', 'tres']
```

El módulo getopt procesa *sys.argv* utilizando las convenciones de la función `getopt()` de UNIX. El módulo optparse proporciona un procesado de línea de órdenes más potente y flexible.

10.4 Redirección de la salida de errores y terminación del programa

El módulo sys también tiene atributos *stdin*, *stdout* y *stderr*. Éste último es útil para emitir mensajes de error para hacerlos visibles si se ha redirigido *stdout*:

```
>>> sys.stderr.write('Aviso: No hay fichero de registro, genero uno nuev
Aviso: No hay fichero de registro, genero uno nuevo
```

El modo más simple de terminar un guiones es usar "`sys.exit()`".

10.5 Búsqueda de patrones de cadenas

El módulo re proporciona herramientas de expresiones regulares para procesado avanzado de cadenas. En problemas que requieren búsquedas y manipulaciones complejas, las expresiones regulares proporcionan soluciones breves y optimizadas:

```
>>> import re
>>> re.findall(r'\bq[a-z]*', 'sabes lo que pasa cuando dices que me quie
['que', 'que', 'quieres']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'arde la calle al sol de de poniente
'arde la calle al sol de poniente'
```

Si sólo se necesitan cosas simples, se prefieren los métodos de las cadenas porque son más sencillos de leer y depurar:

```
>>> 'cien siluetas'.replace('cien', 'mil')
'mil siluetas'
```

10.6 Matemáticas

El módulo math permite el acceso a las funciones de la biblioteca C subyacente para las matemáticas

de coma flotante:

```
>>> import math
>>> math.cos(math.pi / 4.0)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

El módulo `random` proporciona herramientas para hacer selecciones aleatorias:

```
>>> import random
>>> random.choice(['manzana', 'pera', 'banana'])
'manzana'
>>> random.sample(xrange(100), 10)   # muestreo sin reemplazo
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # número en coma flotante aleatorio
0.17970987693706186
>>> random.randrange(6)    # entero aleatorio elegido de range(6)
4
```

10.7 Acceso a internet

Hay varios módulos para acceder a internet y procesar protocolos de internet. Dos de los más sencillos son `urllib2` para recuperar datos de URLs y `smtplib` para enviar correo:

```
>>> import urllib2
>>> for linea in urllib2.urlopen('http://tycho.usno.navy.mil/cgi-bin/tir
...     if 'EST' in linea:      # Buscar Eastern Standard Time
...         print linea

<BR>Nov. 25, 09:43:32 PM EST

>>> import smtplib
>>> servidor = smtplib.SMTP('localhost')
>>> servidor.sendmail('adivino@example.org', 'jcesar@example.org',
"""To: jcesar@example.org
From: adivino@example.org

Guardaos de los idus de marzo.
""")
>>> server.quit()
```

10.8 Fechas y horas

El módulo `datetime` proporciona clases para manipular fechas y horas en operaciones sencillas y complejas. Aunque da soporte a la aritmética de fechas y horas, el énfasis de la implementación es la extracción eficaz de los miembros para formato y manipulación de salida. El módulo también proporciona objetos sensibles a los husos horarios.

```
# es fácil construir y dar formato a fechas
>>> from datetime import date
```

```
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'

# las fechas tienen aritmética de calendario
>>> cumple = date(1967, 11, 10)
>>> edad = now - cumple
>>> edad.days
13592
```

10.9 Compresión de datos

Hay módulos para gestionar los formatos comunes de archivado y compresión, incluyendo:

`zlib`, `gzip`, `bz2`, `zipfile`, and `tarfile`.

```
>>> import zlib
>>> s = 'campana sobre campana y sobre campana otra'
>>> len(s)
42
>>> t = zlib.compress(s)
>>> len(t)
31
>>> zlib.decompress(t)
'campana sobre campana y sobre campana otra'
>>> zlib.crc32(s)
1480756653
```

10.10 Medidas de rendimiento

Algunos usuarios de Python desarrollan un profundo interés en conocer el rendimiento comparativo de diferentes ataques al mismo problema. Python proporciona una herramienta de medida que responde a estas cuestiones de inmediato.

Por ejemplo, es tentador utilizar la característica de empaquetar y desempaquetar tuplas en lugar del método tradicional para intercambiar argumentos. El módulo `timeit` rápidamente demuestra una ligera ventaja de rendimiento:

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

A diferencia de la granularidad de `timeit`, los módulos `profile` y `pstats` proporcionan herramientas para identificar secciones críticas en cuanto al tiempo en bloques de código mayores.

10.11 Control de calidad

Una posible solución para desarrollar software de alta calidad es escribir pruebas para cada función según se desarrolla y ejecutar a menudo dichas pruebas a lo largo del proceso de desarrollo.

El módulo `doctest` proporciona una herramienta para rastrear un módulo y las pruebas de validación incrustados en las cadenas de documentación de un programa. La construcción de las pruebas es una simple tarea de cortaypega en las cadenas de documentación. Esto mejora la documentación proporcionando al usuario un ejemplo y permite al módulo doctest asegurar que el código permanece fiel a la documentación:

```
def media(valores):
    """Calcula la media aritmética de una lista de números.

    >>> print media([20, 30, 70])
    40.0
    """
    return sum(valores, 0.0) / len(valores)

import doctest
doctest.testmod()   # valida automáticamente las pruebas incrustadas
```

El módulo `unittest` requiere algo más de esfuerzo que el módulo `doctest`, pero permite mantener un conjunto completo de pruebas en un fichero aparte:

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        self.assertRaises(ZeroDivisionError, average, [])
        self.assertRaises(TypeError, average, 20, 30, 70)

unittest.main() # Para llamar a todas las pruebas, desde la línea de órd
```

10.12 Pilas incluidas

Python mantiene una filosofía de ``pilas incluidas''. Esto se percibe mejor a partir de la complejidad y robustez de su paquetes más grandes. Por ejemplo:

- Los módulos `xmlrpclib` y `SimpleXMLRPCServer` convierten en trivial la tarea de implementar llamadas a procedimientos remotos. A pesar de su denominación, no hace falta conocer ni manipular XML.
- El paquete `email` es una biblioteca para gestionar mensajes de correo electrónico, incluyendo los documentos de mensajes MIME u otros basados en la RFC 2822. A diferencia de `smtplib` y `poplib` que realmente envían y reciben los mensajes, el paquete email dispone de un juego de herramientas completo para construir o descodificar estructuras de mensajes complejos (incluyendo adjuntos) y para implementar codificaciones de protocolos de internet y cabeceras.
- Los paquetes `xml.dom` y `xml.sax` proporcionan servicios robustos de análisis de este popular

formato de intercambio de datos. De igual modo, el módulo `csv` permite lecturas y escrituras en este formato común de base de datos. Juntos, estos módulos y paquetes simplifican enormemente el intercambio de datos entre aplicaciones en Python y otras herramientas.

- La internacionalización está apoyada por diversos módulos, incluyendo `gettext`, `locale` y el paquete `codecs`.

Release 2.4.1a0, documentation updated on septiembre 11, 2005.
Consultar en Acerca de este documento... información para sugerir cambios.

11. Viaje rápido por la biblioteca estándar II

Este segundo viaje recorre módulos más avanzados que atienden a necesidades de programación profesional. Estos módulos no aparecen usualmente en los guiones más ligeros.

11.1 Formato de salida

El módulo `repr` proporciona una versión de `repr()` para presentaciones limitadas o contenedores muy anidados:

```
>>> import repr
>>> repr.repr(set('supercalifragilísticoexpialidoso'))
"set(['a', 'c', 'd', 'e', 'f', 'g', ...])"
```

El módulo `pprint` ofrece un control más fino sobre la presentación tanto con los objetos internos como los definidos por el usuario en un modo legible por el intérprete. Cuando el resultado es más largo que una línea, la ``imprenta estética" añade cortes de línea y sangrado para hacer evidente la estructura de los datos:

```
>>> import pprint
>>> t = [[[['negro', 'cian'], 'blanco', ['verde', 'rojo']], [['mager
...     'amarillo'], 'azul']]]
...
>>> pprint.pprint(t, width=30)
[[[['negro', 'cian'],
   'blanco',
   ['verde', 'rojo']],
  [['magenta', 'amarillo'],
   'azul']]]
```

El módulo `textwrap` da formato a párrafos para encajar en una anchura de pantalla dada:

```
>>> import textwrap
>>> doc = """El método wrap() es como fill(), pero devuelve una list
... cadenas en lugar de una cadena larga con saltos de línea para se
... las líneas cortadas."""
...
>>> print textwrap.fill(doc, width=40)
El método wrap() es como fill(), pero
devuelve una lista de cadenas en lugar
de una cadena larga con saltos de línea
para separar las líneas cortadas.
```

El módulo `locale` otorga acceso a una base de datos de formatos específicos de cada cultura. El atributo de agrupación de la función de formato de locale proporciona una manera directa de dar formato a números con separadores de grupo:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'English_United States.1252')
'English_United States.1252'
>>> conv = locale.localeconv()          # diccionario de convencione
>>> x = 1234567.8
>>> locale.format("%d", x, grouping=True)
'1,234,567'
>>> locale.format("%s%.*f", (conv['currency_symbol'],
...       conv['int_frac_digits'], x), grouping=True)
'$1,234,567.80'
```

11.2 Plantillas

El módulo `string` incluye una versátil clase `Template` con una sintaxis simplificada adecuada para ser modificada por los usuarios finales. Esto permite que los usuarios personalicen sus aplicaciones sin tener que alterar la aplicación.

El formato utiliza nombres de parámetros formados por "`$`" con identificadores de Python válidos (caracteres alfanuméricos y guiones de subrayado). Para poder poner parámetros pegados a palabras, hay que poner el nombre del parámetro entre llaves. Si se escribe "`$$`" se crea un carácter "`$`" sin significado especial:

```
>>> from string import Template
>>> t = Template('${modo}mente abrazada a $persona.')
>>> t.substitute(modo='Suave', persona='tu loco impasible')
'Suavemente abrazada a tu loco impasible.'
```

El método `substitute` lanza `KeyError` cuando falta un parámetro en el diccionario o argumento por nombre. Para las aplicaciones tipo envío masivo, los datos proporcionados por el usuario pueden estar incompletos, por lo que puede resultar más adecuado el método `safe_substitute`; se dejarán sin tocar los parámetros si faltan datos:

```
>>> t = Template('Devolver $elemento a $remitente.')
>>> d = dict(elemento='golondrina sin carga')
>>> t.substitute(d)
Traceback (most recent call last):
  . . .
KeyError: 'remitente'
>>> t.safe_substitute(d)
'Devolver golondrina sin carga a $remitente.'
```

Las subclases de Template pueden especificar un delimitador específico. Por ejemplo, un utensilio de renombrado masivo de un visualizador de fotos podría elegir los símbolos de porcentaje para parámetros como la fecha actual, número de secuencia de la imagen o un formato de fichero:

```
>>> import time, os.path
>>> ficheros = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class RenombradoMasivo(Template):
...     delimitador = '%'
>>> fmt = raw_input('Introducir estilo de renombrado (%d-fecha %n-numsec
Introducir estilo de renombrado (%d-fecha %n-numsec %f-formato):  Blas_%

>>> t = RenombradoMasivo(fmt)
>>> fecha = time.strftime('%d%b%y')
```

```
>>> for i, nombrefich in enumerate(ficheros):
...     base, ext = os.path.splitext(nombrefich)
...     nombrenuevo = t.substitute(d=fecha, n=i, f=ext)
...     print '%s --> %s' % (nombrefich, nombrenuevo)

img_1074.jpg --> Blas_0.jpg
img_1076.jpg --> Blas_1.jpg
img_1077.jpg --> Blas_2.jpg
```

Otra aplicación de las plantillas es separar la lógica de programación de los detalles de los diferentes formatos de salida. Esto permite utilizar plantillas a medida ficheros XML, informes en texto plano e informes para la web en HTML.

11.3 Trabajo con formatos de registros de datos binarios

El módulo `struct` proporciona las funciones `pack()` y `unpack()` para trabajar con formatos de registro binario de longitud variable. El siguiente ejemplo muestra cómo recorrer la información de cabecera de un fichero ZIP (con códigos de empaquetado `"H"` y `"L"` que representan números sin signo de dos y cuatro bytes, respectivamente):

```
import struct

datos = open('mifich.zip', 'rb').read()
inicio = 0
for i in range(3):                      # muestra las tres primeras
    inicio += 14
    campos = struct.unpack('LLLHH', datos[inicio:inicio+16])
    crc32, tam_comp, tam_descomp, tamnombrefich, tam_extra =  campos

    inicio += 16
    nombrefich = datos[inicio:inicio+tamnombrefich]
    inicio += tamnombrefich
    extra = datos[inicio:inicio+tam_extra]
    print nombrefich, hex(crc32), tam_comp, tam_descomp

    inicio += tam_extra + tam_comp     # saltar hasta la siguiente c
```

11.4 Multi-hilo

El multihilado es una técnica para independizar tareas que no son secuencialmente dependientes. Se pueden utilizar los hilos para mejorar la respuesta de las aplicaciones que aceptan entrada del usuario mientras otras tareas se ejecutan en segundo plano. Un caso parecido es la ejecución de E/S en paralelo con los cálculos de otro hilo.

El siguiente código muestra cómo el módulo de alto nivel `threading` puede ejecutar tareas en segundo plano mientras el programa principal prosigue su ejecución:

```
import threading, zipfile
```

```
class AsincZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile
    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print 'Finished background zip of: ', self.infile

segundoplano = AsincZip('misdatos.txt', 'comprimido.zip')
segundoplano.start()
print 'El programa principal sigue ejecutándode en primer plano.'

segundoplano.join()    # Esperar a que termine la tarea en segundo p
print 'El programa principal ha esperado hasta que la tarea en 2º p
```

El mayor reto de las aplicaciones multihilo es coordinar los hilos que comparten datos u otros recursos. A este fin, el módulo de multihilo proporciona varias primitivas de sincronización que incluyen bloqueos, eventos, variables de condición y semáforos.

Aunque estas herramientas son potentes, los errores ligeros de diseño pueden causar problemas difíciles de reproducir. Por ello, el método aconsejado para coordinar tareas es concentrar todo el acceso a un recurso en un solo hilo y utilizar el módulo `Queue` para alimentar ese hilo con peticiones desde los otros hilos. Las aplicaciones que usan objetos `Queue` para comunicación y coordinación entre hilos son más fáciles de diseñar, más legibles y más fiables.

11.5 Registro de actividad

El módulo `logging` ofrece un sistema potente y flexible de registro de actividad. En su forma más simple, los mensajes de registro se envían a un fichero o a `sys.stderr`:

```
import logging
logging.debug('Información de depuración')
logging.info('Mensaje informativo')
logging.warning('Aviso:falta el fichero de configuración %s ', 'serv
logging.error('Ha ocurrido un error')
logging.critical('Error crítico; apagando')
```

Esto produce la siguiente salida:

```
WARNING:root:Aviso:falta el fichero de configuración server.conf
ERROR:root:Ha ocurrido un error
CRITICAL:root:Error crítico; apagando
```

De manera predeterminada, los mensajes informativos y de depuración se suprimen cuando la salida se dirige al flujo de error estándar. Otras opciones de salida incluyen encaminar los mensajes a correo electrónico, datagramas, zócalos (sockets) o a un servidor HTTP. Los filtros pueden seleccionar el encaminado basándose en la prioridad del mensaje: `DEBUG`, `INFO`, `WARNING`, `ERROR` y `CRITICAL`.

El sistema de registro se puede configurar directamente desde Python o se puede cargar a partir de

un fichero de configuración para personalizar el registro sin alterar la aplicación.

11.6 Referencias débiles

Python gestiona la memoria automáticamente (cuenta de referencias en la mayoría de los objetos y recogida de basura para eliminar ciclos). La memoria se libera poco después de que se elimine la última referencia a ella.

Esta técnica funciona bien para la mayoría de las aplicaciones pero ocasionalmente hay necesidad de seguir objetos mientras los usen otros. Desafortunadamente, el hecho de seguirlos crea una referencia a ellos que los hace permanentes. El módulo `weakref` proporciona herramientas para seguir objetos sin crear una referencia a ellos. Cuando el objeto ya no se necesita, se retira automáticamente de una tabla weakref y se dispara una llamada de respuesta (callback) a los objetos weakref. Las aplicaciones típicas incluyen hacer una caché de objetos de creación costosa:

```
>>> import weakref, gc
>>> class A:
...     def __init__(self, value):
...             self.value = value
...     def __repr__(self):
...             return str(self.value)
...
>>> a = A(10)                    # crea una referencia
>>> d = weakref.WeakValueDictionary()
>>> d['primaria'] = a            # no crea una referencia
>>> d['primaria']                # obtiene el objeto si está vivo au
10
>>> del a                        # elimina la única referencia
>>> gc.collect()                 # recoge la basura ahora mismo
0
>>> d['primaria']                # la entrada se eliminó automáticar
Traceback (most recent call last):
  File "<pyshell#108>", line 1, in -toplevel-
    d['primaria']               # la entrada se eliminó automáticar
  File "C:/PY24/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primaria'
```

11.7 Herramientas para trabajar con listas

Muchas de las cuestiones relativas a estructuras de datos se pueden solucionar con el tipo de lista interno. Sin embargo, a veces hay necesidad de implementaciones alternativas con diversas opciones de rendimiento.

El módulo `array` proporciona un objeto `array()` que es como una lista que sólo almacena datos homogéneos pero de una forma más compacta. El siguiente ejemplo muestra cómo guardar un vector de números almacenados como números de dos bytes sin signo (código de tipo `"H"`) en lugar de los 16 bytes por entrada de las listas normales de objetos enteros de Python:

```
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
```

```
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])
```

El módulo `collections` proporciona un objeto `deque()` que es como una lista con operaciones append y pop por el lado izquierdo más rápidas pero búsquedas más lentas enmedio. Estos objetos son adecuados para implementar colas y búsquedas de árbol a lo ancho (breadth first).

```
>>> from collections import deque
>>> d = deque(["tarea1", "tarea2", "tarea3"])
>>> d.append("tarea4")
>>> print "Handling", d.popleft()
Gestionando tarea1

unsearched = deque([starting_node])
def breadth_first_search(unsearched):
    node = unsearched.popleft()
    for m in gen_moves(node):
        if is_goal(m):
            return m
        unsearched.append(m)
```

Además de implementaciones alternativas de las listas, la biblioteca también ofrede herramientas como el módulo `bisect`, que contiene funciones para manipular listas ordenadas:

```
>>> import bisect
>>> puntuaciones = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500,
>>> bisect.insort(puntuaciones, (300, 'ruby'))
>>> puntuaciones
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'py
```

El módulo `heapq` proporciona funciones para implementar montículos (heaps) basados en listas normales. El elemento de menos valor siempre se mantiene en la posición cero. Esto es útil para las aplicaciones que acceden repetidamente al menor elemento pero no necesitan la ordenación de la lista entera:

```
>>> from heapq import heapify, heappop, heappush
>>> datos = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(datos)                      # reordena la lista en orden
>>> heappush(datos, -5)                 # añade un nuevo elemento
>>> [heappop(datos) for i in range(3)]  # obtiene los tres elementos
[-5, 0, 1]
```

11.8 Aritmética de coma flotante decimal

El módulo `decimal` proporciona un tipo de dato `Decimal` para la aritmética de coma flotante decimal. Comparada con la implementación interna de `float` de la aritmética de coma flotante binaria, la nueva clase es especialmente útil para aplicaciones financieras y otros usos que exigen una representación decimal exacta, control de la precisión, control sobre el redondeo para amoldarse a requisitos legales, seguimiento del número de cifras significativas o aplicaciones en las que el usuario espera que los resultados coincidan con los cálculos hechos a mano.

Por ejemplo, calcular un impuesto del 5% sobre una tarifa telefónica de 70 céntimos da diferentes resultados en coma flotante decimal y coma flotante binaria. La diferencia resulta significativa si se redondean los resultados al céntimo más cercano:

```
>>> from decimal import *
>>> Decimal('0.70') * Decimal('1.05')
Decimal("0.7350")
>>> .70 * 1.05
0.73499999999999999
```

El resultado `Decimal` mantiene un cero por detrás, obteniendo automáticamente cuatro cifras significativas a partir de los operandos de dos cifras. `Decimal` reproduce las cuentas como si se hicieran a mano y evita problemas que pudieran surgir cuando la coma flotante no puede representar exactamente cantidades decimales.

La representación exacta permite que la clase `Decimal` realice cálculos de restos y pruebas de igualdad que son impropias en la coma flotante binaria:

```
>>> Decimal('1.00') % Decimal('.10')
Decimal("0.00")
>>> 1.00 % 0.10
0.09999999999999995

>>> sum([Decimal('0.1')]*10) == Decimal('1.0')
True
>>> sum([0.1]*10) == 1.0
False
```

El módulo `decimal` proporciona tanta precisión como sea necesaria:

```
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal("0.142857142857142857142857142857142857")
```

Release 2.4.1a0, documentation updated on septiembre 11, 2005.
Consultar en Acerca de este documento... información para sugerir cambios.

12. Y ahora, ¿qué?

Probablemente, la lectura de esta guía de aprendizaje haya aumentado tu interés en utilizar Python; deberías estar ansioso de aplicar Python para resolver problemas de la vida real. ¿Qué es lo que hay que hacer ahora?

Deberías leer, o al menos hojear, la *Referencia de las librerías de Python*, que ofrece una referencia completa (un poco dura) de los tipos, funciones y módulos que pueden ahorrarte un montón de tiempo al escribir programas en Python. La distribución estándar de Python incluye *mucho* código en C y en Python. Existen módulos para leer buzones UNIX, recuperar documentos por HTTP, generar números aleatorios, extraer opciones de una línea de órdenes, escribir programas CGI, comprimir datos... Un vistazo a la Referencia de las bibliotecas te dará una idea de lo que hay disponible.

La web de Python más importante es http://www.python.org/. Contiene código, documentación y direcciones de páginas relacionadas con Python por toda la web. Esta web tiene réplicas en varios lugares del planeta: en Europa, Japón y Australia. Dependiendo de su ubicación, las réplicas pueden ofrecer una respuesta más rápida. Existe una web más informal en http://starship.python.net/ que contiene unas cuantas páginas personales relativas a Python. Mucha gente tiene software descargable en estas páginas. Hay muchos módulos de Python creados por usuarios en el Índice de paquetes de Python (PyPI).

Si tienes preguntas relativas a Python o deseas comunicar problemas, puedes publicar en el grupo de discusión comp.lang.python o enviar correo a la lista de correos de python-list@python.org. El grupo y la lista tienen una pasarela automática, por lo que un mensaje enviado a uno de ellos se remitirá automáticamente al otro. Hay unos 120 mensajes al día (con picos de varios cientos), de preguntas (y respuestas), sugerencias de nuevas características y anuncios de nuevos módulos. Antes de publicar, asegúrate de comprobar la lista de preguntas frecuentes (de las iniciales inglesas, FAQ) o buscar en el directorio Misc/ de la distribución en fuentes de Python. Los archivos de la lista de correos están disponibles en http://www.python.org/pipermail/. La FAQ da respuesta a muchas de las preguntas que surgen una y otra vez y puede que contenga la solución de tu problema.

Release 2.4.1a0, documentation updated on septiembre 11, 2005.
Consultar en Acerca de este documento... información para sugerir cambios.

A. Edición de entrada interactiva y sustitución de historia

Algunas versiones del intérprete de Python permiten la edición de la línea de entrada en curso y la sustitución histórica, servicios similares a los existentes en ksh y bash de GNU. Esto se consigue mediante la biblioteca de GNU *Readline*, que permite edición estilo Emacs y estilo vi. Esta biblioteca tiene su propia documentación, que no voy a duplicar aquí. Sin embargo, es fácil contar lo más básico. La edición interactiva y el histórico descritos aquí están disponibles opcionalmente en las versiones UNIX y CygWin del intérprete.

Este capítulo *no* documenta los servicios de edición del Paquete PythonWin de Mark Hammond ni el entorno basado en Tk, IDLE, distribuido con Python. La recuperación de historia de la línea de órdenes que funciona en DOS o en NT u otras cosas es también otra historia.

A.1 Edición de línea

Si está disponible, la edición de línea de entrada está activa siempre que el intérprete imprime un indicador principal o secundario. Se puede modificar la línea en curso utilizando los caracteres de control normales de Emacs. Los más importantes son: `C-A` (Control-A) mueve el cursor al principio de la línea, `C-E` final. `C-K` borra hasta el final de la línea, `C-Y` recupera la última cadena eliminada. `C-_` deshace el último cambio realizado (se puede deshacer varias veces).

A.2 Sustitución de historia

La sustitución de historia funciona de la siguiente manera. Cualquier línea de entrada no vacía se guarda en un histórico. Cuando se emite un nuevo indicador, estás situado en una línea nueva debajo de todo el histórico. `C-P` sube una línea (hacia líneas anteriores) en el histórico y `C-N` baja una línea. Se puede editar cualquier línea del histórico: aparece un asterisco en frente del indicador para indicar que una línea se ha modificado. Al pulsar la tecla de `retorno`, se pasa la línea actual al intérprete. `C-R` comienza una búsqueda inversa incremental y `C-S` empieza una búsqueda hacia delante.

A.3 Teclas

Es posible personalizar las asignaciones de teclas y otros parámetros de la biblioteca Readline insertando órdenes en un fichero de arranque denominado ~/.inputrc. Las asignaciones de teclas tienen esta forma:

```
nombre-tecla: nombre-función
```

o

```
"cadena": nombre-función
```

y se cambian las opciones con

```
set nombre-opción valor
```

Por ejemplo:

```
# Prefiero edición tipo vi:
set editing-mode vi

# Editar con una sola línea:
set horizontal-scroll-mode On

# Reasignar varias teclas:
Meta-h: backward-kill-word
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
```

Observa que la asignación por omisión del `tabulador` en Python corresponde a insertar un tabulador, en lugar de la función de completado de nombre de fichero por omisión en Readline. Si insistes, se puede forzar esto poniendo

```
Tab: complete
```

en tu fichero ~/.inputrc (por supuesto, esto dificulta teclear líneas de continuación sangradas si usabas el tabulador para ello).

Opcionalmente, está disponible el completado automático de nombres de módulos y variables. Para activarlo en el modo interactivo, añade lo siguiente al fichero de arranque[A.1].

```
import rlcompleter, readline
readline.parse_and_bind('tab: complete')
```

Esto asocia el tabulador a la función de completado, por lo que pulsar dos veces el tabulador sugiere terminaciones posibles de la palabra. Busca en los nombres de sentencias Python, las variables locales actuales y los nombres de módulos disponibles. Para expresiones con punto, como `cadena.a`, primero evalúa la expresión hasta el último "`.`" y sugiere los atributos del objeto resultante. Fíjate que esto puede provocar la ejecución de código definido por la aplicación si hay un objeto con un método `__getattr__()` como parte de la expresión.

Un fichero de arranque más funcional podría parecerse a este ejemplo. Observa que éste borra los nombres que crea una vez que no se necesitan más; esto se hace porque el fichero de arranque se ejecuta en el mismo espacio nominal que las órdenes interactivas, así que eliminar dichos nombres evita efectos secundarios sobre el entorno interactivo. Podría interesarte mantener algunos de los nombres importados, tales como `os`, que acaban siendo necesarios en la mayoría de sesiones con el intérprete.

```
# Añadir auto-completar y un fichero persistente de la historia de las
# órdenes al intérprete interactivo de Python. Necesita Python 2.0 o sup
# readline. Auto-completar está asociado a la tecla Esc de manera
# predeterminada (se puede cambiar, leer la doc. de readline).
#
```

```
# Guardar el fichero en ~/.pystartup, y establecer una variable de entor
# que apunte a él: "export PYTHONSTARTUP=/max/home/itamar/.pystartup" en
#
# Ojo: PYTHONSTARTUP *no* expande "~", así que hay que poner la ruta com
# hasta el directorio del usuario

import atexit
import os
import readline
import rlcompleter

historyPath = os.path.expanduser("~/.pyhistory")

def save_history(historyPath=historyPath):
    import readline
    readline.write_history_file(historyPath)

if os.path.exists(historyPath):
    readline.read_history_file(historyPath)

atexit.register(save_history)
del os, atexit, readline, rlcompleter, save_history, historyPath
```

A.4 Comentarios

Este servicio es un enorme paso adelante comparado con las anteriores versiones del intérprete. Sin embargo, quedan muchos deseos por cumplir: Sería cómodo que se pusiera automáticamente el sangrado correcto en líneas de continuación (el analizador sabe si hace falta un sangrado). El mecanismo de completado automático podría utilizar la tabla de símbolos del intérprete. También vendría bien una orden para comprobar (y hasta sugerir) las parejas de paréntesis, comillas, etc.

Notas al pie

... arranque[A.1]
Python ejecutará el contenido de un fichero identificado por la variable de entorno PYTHONSTARTUP al arrancar una sesión interactiva del intérprete

---

Release 2.4.1a0, documentation updated on septiembre 11, 2005.
Consultar en Acerca de este documento... información para sugerir cambios.

B. Aritmética de coma flotante: Consideraciones y limitaciones

Los números de coma flotante se representan dentro del hardware de la computadora como fracciones en base 2 (binarias). Por ejemplo, la fracción decimal

```
0.125
```

tiene un valor de 1/10 + 2/100 + 5/1000 y, del mismo modo, la fracción binaria

```
0.001
```

tiene el valor 0/2 + 0/4 + 1/8. Estas dos fracciones tienen idéntico valor. La única diferencia real entre ambas es que la primera está escrita en notación fraccional de base 10 y la segunda en base 2.

Desafortunadamente, la mayoría de las fracciones decimales no se pueden representar como fracciones binarias. Una consecuencia de ello es que, en general, los números de coma flotante decimales que se introducen se almacenan como números sólo aproximados en la máquina.

El problema es más fácil de empezar a entender en base 10. Considera la fracción 1/3. Se puede aproximar mediane una fracción de base 10:

```
0.3
```

o mejor,

```
0.33
```

o mejor,

```
0.333
```

y así sucesivamente. No importa cuántos dígitos se escriban, el resultado nunca será 1/3 exactamente, sino una aproximación cada vez mejor de 1/3.

Del mismo modo, no importa cuantos dígitos de base 2 se usen, el valor decimal 0,1 no se puede representar exactamente como una fracción de base 2. En base 2, 1/10 es la fracción periódica

```
0.0001100110011001100110011001100110011001100110011...
```

Se puede parar en cualquier número finito de bits y se obtiene una aproximación. Por este motivo se

ven cosas como:

```
>>> 0.1
0.10000000000000001
```

En la mayoría de las máquinas actuales, esto es lo que se ve si se introduce 0.1 en el símbolo de entrada de Python. Es posible que no, sin embargo, porque el número de bits utilizado por el hardware para almacenar valores de coma flotante puede variar entre máquinas y Python sólo presenta una aproximación decimal de la aproximación binaria almacenada en la máquina. En la mayoría de las máquinas, si Python hubiera de presentar la aproximación binaria de 0.1, ¡tendría que presentar

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

en su lugar! El modo interactivo de Python (implícitamente) utiliza la función interna `repr()` para obtener una versión en forma de cadena de todo lo que presenta. Para los valores de coma flotante, `repr(`*float*`)` redondea el valor decimal real a 17 dígitos significativos, resultando

```
0.10000000000000001
```

`repr(`*float*`)` produce 17 dígitos significativos porque resulta que es suficiente para que `eval(repr(`*x*`)) ==` *x* exactamente para todos los valores de coma flotante finitos *x*, pero redondear a 16 dígitos no es suficiente para que esto sea cierto.

Hay que darse cuenta de que esto es la naturaleza de la coma flotante binaria, no un bug en Python ni en el código del usuario. Este efecto se observa en todos los lenguajes que dan soporte a la aritmética de coma flotante del hardware (aunque es posible que algunos lenguajes no *muestren* la diferencia de forma predeterminada o en todos los modos).

La función interna de Python `str()` produce sólo 12 dígitos significativos, puede que te resulte más útil. No es común que `eval(str(`*x*`))` reproduzca *x*, pero el resultado visible puede ser más agradable:

```
>>> print str(0.1)
0.1
```

Es importante darse cuenta de que esto es, en sentido estricto, una ilusión: el valor dentro de la máquina no es exactamente 1/10, sólo estás redondeando el valor *presentado* del verdadero valor almacenado en la máquina.

Se pueden ver otros resultados sorprendentes. Por ejemplo, tras ver

```
>>> 0.1
0.10000000000000001
```

se puede ver uno tentado a utilizar la función `round()` para recortarlo al único dígito decimal esperado. El resultado no cambia:

```
>>> round(0.1, 1)
0.10000000000000001
```

El problema es que el valor de coma flotante almacenado para "0,1" ya era la mejor aproximación posible a 1/10 e intentar redondearlo no lo puede mejorar: ya era todo lo bueno posible.

Otra consecuencia es que como 0,1 no es exactamente 1/10, sumar 0,1 consigo mismo 10 veces no da como resultado 1,0, tampoco:

```
>>> suma = 0.0
>>> for i in range(10):
...     suma += 0.1
...
>>> suma
0.99999999999999989
```

La aritmética de coma flotante reserva muchas sorpresas del estilo. El problema con "0,1" se explica con precisión más tarde, en la sección "Error de representación". Consultar en *The Perils of Floating Point* una recopilación más completa de otras sorpresas habituales.

Como dice cerca del final, ``no hay respuestas fáciles". Aun así, ¡no hay que temer a la coma flotante! Los errores de las operaciones de coma flotante de Python se heredan del hardware de coma flotante, y en la mayoría de los casos están en el orden de 1 entre 2**53 por operación. Esto basta en la mayoría de los casos, pero hay que mantener presente que no es aritmética decimal y que cada operación en coma flotante puede acumular un nuevo error de redondeo.

Aunque existen casos patológicos, en la mayoría de aritmética de coma flotante informal verás el resultado esperado al final si redondeas la presentación del resultado final al número de cifras esperado. Suele bastar `str()` para ello. Si se desea un control más fino, consultar la discusión del operador de formato `%` de Python: los códigos de formato `%g`, `%f` y `%e` proporcionan modos flexibles y sencillos de redondear los resultados de coma flotante para su presentación.

B.1 Error de representación

Esta sección explica el ejemplo del ``0.1" en detalle y muestra cómo puedes realizar tú mismo un análisis exacto de casos como éste. Se supone una familiaridad básica con la representación de coma flotante binaria.

El *error de representación* se refiere a que algunas (la mayoría, en realidad) fracciones decimales no se pueden representar exactamente como fracciones binarias (de base 2). Ésta es la principal razón de que Python (o Perl, C C++, Java, Fortran y muchos otros) no suelan presentar el número decimal que esperas:

```
>>> 0.1
0.10000000000000001
```

¿A qué se debe esto? 1/10 no es exactamente representable como una fracción binaria. Casi todas las máquinas de hoy en día (noviembre de 2000) usan aritmética de coma flotante de ``doble precisión" IEEE-754. Los dobles de 753 contienen 53 bits de precisión, así que a la entrada el ordenador se las ve y se las desea para convertir 0,1 a la fracción más cercana de la forma *J*/2**_N_ donde *J* es un entero de exactamente 53 bits. Reescribiendo

```
1 / 10 ~= J / (2**N)
```

como

```
J ~= 2**N / 10
```

y recordando que *J* tiene 53 bits (es `>= 2**52` pero `< 2**53`), el mejor valor para *N* es 56:

```
>>> 2**52
4503599627370496L
>>> 2L**53
9007199254740992L
>>> 2L**56/10
7205759403792793L
```

Es decir, 56 es el único valor de *N* que deja *J* con exactamente 53 bits. El mejor valor posible para *J* es, pues, este cociente redondeado:

```
>>> q, r = divmod(2L**56, 10)
>>> r
6L
```

Como el resto es mayor que la mitad de 10, la mejor aproximación se obtiene redondeando hacia arriba:

```
>>> q+1
7205759403792794L
```

Por consiguiente, la mejor aproximación posible de 1/10 en precision doble 754 es dicho resultado partido por 2**56, o

```
7205759403792794 / 72057594037927936
```

Observa que, como hemos redondeado hacia arriba, esto supera ligeramente 1/10; si no hubiéramos redondeado hacia arriba, el cociente hubiera sido ligeramente menor que 1/10. ¡Pero en ningún caso puede ser *exactamente* 1/10!

Así que el ordenador nunca ``ve'' 1/10: lo que ve es la fracción exacta dada anteriormente, la mejor aproximación doble 754 que puede conseguir:

```
>>> .1 * 2L**56
7205759403792794.0
```

Si multiplicamos dicha fracción por 10**30, podemos ver el valor (truncado) de sus 30 cifras decimales más significativas:

```
>>> 7205759403792794L * 10L**30 / 2L**56
100000000000000000055511151231 25L
```

lo que significa que el número exacto almacenado en el ordenador es aproximadamente igual al valor decimal 0,100000000000000000055511151231 25. Redondeando esto a 17 cifras significativas da el 0.10000000000000001 que presenta Python (bueno, lo hará en cualquier plataforma compatible 754 que realice la mejor conversión posible en entrada y salida, ¡igual en la tuya no!).

Release 2.4.1a0, documentation updated on septiembre 11, 2005.
Consultar en Acerca de este documento... información para sugerir cambios.

D. Glosario

>>>
El indicador típico de la interfaz interactiva de Python. Suele indicar código de ejemplo que se puede probar directamente en el intérprete.

...
El indicador típico de la interfaz interactiva de Python al introducir código de un bloque sangrado.

ámbitos anidados
La capacidad de referirse a una variable de una definición que envuelve a la actual. Por ejemplo, una función definida dentro de otra función puede referirse a las variables de la función más externa. Es importante recordar que los ámbitos anidados sólo funcionan en referencia y no en asignación, que siempre escribirá sobre la función externa. Por contra, las variables locales se escriben y leen en el ámbito más interno. Igualmente, las variables globales se leen y escriben en el espacio nominal global.

BDFL
Dictador bienintencionado vitalicio (Benevolent Dictator For Life), alias Guido van Rossum, el creador de Python.

byte code
La representación interna de un programa Python en el intérprete. El bytecode también se almacena en modo caché en los ficheros `.pyc` y `.pyo` para que sea más rápido ejecutar el mismo fichero la segunda vez (la compilación a bytecode vale de una vez a otra). Se dice que este ``lenguaje intermedio'' se ejecuta sobre una ``máquina virtual'' que llama a las subrutinas correspondientes a cada código del bytecode.

clase tradicional
Cualquier clase que no herede de `object`. Ver *clase moderna*.

clase moderna
Cualquier clase que herede de `object`. Esto incluye todos los tipos internos como `list` y `dict`. Sólo las clases modernas pueden utilizar las características más novedosas y versátiles de Python, como los `__slots__`, descriptores, propiedades, `__getattribute__()`, métodos de clase y métodos estáticos.

coerción

La conversión implícita de una instancia de un tipo a otro durante una operación que involucra dos argumentos del mismo tipo. Por ejemplo, `int(3.15)` convierte el número de coma flotante al entero `3`, pero en `3+4.5`, cada argumento es de un tipo diferente (un entero y un flotante) y hay que convertir los dos en el mismo tipo antes de que puedan ser sumados o se lanzará un `TypeError`. La coerción entre dos operandos se puede realizar con la función interna `coerce`; por ello, `3+4.5` equivale a `operator.add(*coerce(3, 4.5))` y devuelve `operator.add(3.0, 4.5)`. Sin coerción, todos los argumentos, incluso de tipos compatibles, habrían de ser normalizados al mismo tipo por el programador, es decir, `float(3)+4.5` en

lugar de simplemente `3+4.5`.

correspondencia
Un objeto contenedor (como `dict`) que permite búsquedas de claves arbitrarias utilizando el método especial `__getitem__()`.

descriptor
Cualquier objeto *moderno* que defina los métodos `__get__()`, `__set__()` o `__delete__()`. Cuando un atributo de clase es un descriptor, se dispara su comportamiento de enlace especial para cualquier búsqueda de atributos. Normalmente, escribir *a.b* busca el objeto *b* en el diccionario de la clase de *a*, pero si *b* es un descriptor, se llama al método definido. La comprensión de los descriptores es clave para una comprensión profunda de Python, porque son la base de muchas características, incluyendo funciones, métodos, propiedades, métodos de clase, métodos estáticos y referencia a súper clases.

diccionario
Un vector asociativo, donde claves arbitrarias se hacen corresponder a valores. El uso de `dict` se parece en mucho al de `list`, pero las claves pueden ser cualquier objeto con una función `__hash__()`, no se limitan a enteros partiendo de cero. En Perl se llama hash.

División entera
La división matemática que descarta el resto. Por ejemplo, la expresión `11/4` se evalúa en la actualidad a `2` a diferencia del valor `2.75` que devuelve la división de coma flotante. Cuando se dividen dos enteros el resultado siempre será otro entero (se aplicará la función truncar al resultado). Sin embargo, si uno de los operandos es de otro tipo numérico (por ejemplo, un `float`), el resultado se adaptará (ver *coerción*) a un tipo común. Por ejemplo, un entero dividido por un flotante dará un valor flotante, posiblemente con una fracción decimal. Se puede forzar la división entera utilizando el operador `//` en lugar del operador `/`. Ver también *__future__*.

duck-typing
El estilo de programación de Python que determina el tipo de un objeto mediante la inspección de la signatura de sus métodos y atributos más que por una relación explícita a algún objeto tipo (``Si parece un pato y grazna como un pato, debe de ser un pato"). Haciendo énfasis en las interfaces más que en tipos específicos, el código bien diseñado mejora su flexibilidad permitiendo la sustitución polimórfica. El tipado del pato intenta evitar las comprobaciones con `type()` o `isinstance()`. En su lugar, suele emplear pruebas con `hasattr()` o programación *EAFP*.

EAFP
Siglas en inglés de ``Mejor pedir perdón que permiso". Este estilo de programación, común en Python, asume la existencia de claves o atributos válidos y captura las excepciones que se producen si tal suposición se revela falsa. Este estilo limpio y rápido se caracteriza por la existencia de muchas sentencias `try` y `except`. Esta técnica contrasta con el estilo *LBYL*, común en muchos otros lenguajes, como C.

espacio nominal
El lugar donde se almacena una variable. Los espacios nominales se implementan como diccionarios. Hay espacios nominales locales, globales e internos, además de los espacios nominales anidados de los objetos (sus métodos). Los espacios nominales posibilitan la modularidad al prevenir los conflictos entre nombres. Por ejemplo, las funciones `__builtin__.open()` y `os.open()` se distinguen entre sí por sus espacios nominales. Los espacios nominales también mejoran la legibilidad y la mantenibilidad dejando claro qué módulos implementan una función. Por ejemplo, escribir `random.seed()` o `itertools.izip`

`()` deja claro qué funciones se implementan en el módulo `random` o en el módulo `itertools`, respectivamente.

expresión generadora
Una expresión que devuelve un generador. Parece una expresión normal seguida por una expresión `for` que define una variable de bucle, rango y una expresión opcional `if`. La expresión combinada genera valores para una función que la contenga:

```
>>> sum(i*i for i in range(10)) # la suma de los cuadrados 0, 1, 4.
285
```

__future__
Un pseudo-módulo que pueden usar los programadores para activar características nuevas del lenguaje incompatibles con el intérprete actual. Por ejemplo, la expresión `11/4` actualmente se evalúa como `2`. Si el módulo en que se ejecuta ha activado la *división fiel* ejecutando:

```
from __future__ import division
```

la expresión `11/4` se evaluaría como `2.75`. Si se importa el módulo `__future__` propiamente dicho y se evalúan sus variables, se puede ver cuándo se añadió una nueva característica y cuando será activa de manera predeterminada:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

generador
Una función que devuelve un iterador. Parece una función normal, salvo que los valores se devuelven al llamante utilizando una sentencia `yield` en lugar de una sentencia `return`. Las funciones generadoras suelen contener uno o más bucles `for` o `while` que devuelven (`yield`) elementos al llamante. La ejecución de la función se para en la palabra clave `yield` (devolviendo el resultado) y se reanuda cuando se solicita el siguiente elemento llamando al método `next()` del iterador devuelto.

GIL
Ver *bloqueo global del intérprete*.

bloqueo global del intérprete

El bloqueo que utilizan los hilos de Python para asegurar que sólo se ejecute un hilo a la vez. Esto simplifica Python asegurando que dos procesos no acceden a la memoria simultáneamente. Bloquear el intérprete entero facilita al intérprete ser multihilo, a costa de perder paralelismo en máquinas multiprocesador. Se han realizado esfuerzos para crear un intérprete de ``hilos libres'' (uno que bloquee los datos con un nivel más fino), pero el rendimiento se resentía en el caso habitual de un solo procesador.

IDLE
Un Entorno de Desarrollo Integrado para Python. IDLE es un entorno básico de editor e intérprete que se incluye en la distribución estándar de Python. Es adecuado para novatos y sirve de código de ejemplo claro para los que busquen implementar una aplicación con interfaz de usuario gráfica multiplataforma y de complejidad mediana.

inmutable
Un objeto con valor fijo. Los objetos inmutables incluyen los números, las cadenas y las

tuplas. Un objeto inmutable no puede ser alterado. Si se desea un valor diferente es necesario crear un nuevo objeto. Cumplen un papel importante en los sitios donde se necesita un valor hash constante, como en las claves de un diccionario.

interactivo
Python cuenta con un intérprete interactivo, lo que significa que se pueden probar cosas y ver de inmediato los resultados. Basta con lanzar `python` sin argumentos (o seleccionarlo desde el menú principal del ordenador). Es un medio muy potente de experimentar o inspeccionar módulos y paquetes (recuerda `help(x)`).

interpretado
Python es un lenguaje interpretado, a diferencia de uno compilado. Esto significa que los archivos de fuentes se pueden ejecutar directamente sin tener que crear un ejecutable. Los lenguajes interpretados suelen tener un ciclo de desarrollo/depuración más rápido, aunque sus programas también suelen más lentos en ejecución. Ver también *interactive*.

iterable
Un objeto contenedor capaz de devolver sus miembros de uno en uno. Ejemplos de iterables incluyen todos los tipos secuencia (como `list`, `str` y `tuple`) y algunos otros, como `dict` y `file` o cualquier objeto que se defina con un método `__iter__()` o `__getitem__()`. Los iterables se pueden usar en un bucle `for` y en muchos otros lugares donde se necesite una secuencia (`zip()`, `map()`...). Cuando se pasa un objeto iterable como argumento de la función interna `iter()`, devuelve un iterador para el objeto. Este iterador es válido para un recorrido por el conjunto de valores. Al usar iterables, no suele ser necesario llamar a `iter()` o gestionar los objetos iteradores explícitamente. La sentencia `for` lo hace automáticamente, creando una variable temporal anónima para guardar el iterador durante la ejecución del bucle. Ver también *iterador*, *secuencia* y *generador*.

iterador
Un objeto que representa un flujo de datos. Las sucesivas llamadas al método `next()` del iterador devuelven elementos sucesivos del flujo. Cuando no quedan más datos, se lanza una excepción `StopIteration` en lugar de devolver más datos. En este punto, el objeto iterador está agotado y cualquier llamada a su método `next()` se limitará a lanzar de nuevo `StopIteration`. Se exige que los iteradores devuelvan un método `__iter__()` que devuelva el propio objeto iterador para que cada iterador sea a su vez iterable y pueda ser utilizado en la mayoría de los casos en que se puedan utilizar otros iterables. Una importante excepción es el código que intenta varias pasadas por la iteración. Un objeto contenedor (como una `list`) genera un iterador nuevecito cada vez que se le pasa a la función `iter()` o se usa en un bucle `for`. Si se intenta esto con un iterador, se reutilizará el mismo objeto iterador agotado que se usó en la iteración anterior, haciendo que parezca vacío.

LBYL
Look before you leap (mira antes de saltar). Este estilo de codificar verifica explicitamente las condiciones previas antes de hace llamadas o búsquedas. Este estilo contrasta con el *EAFP* y se caracteriza por la abundancia de sentencias `if`.

lista autodefinida
Una manera compacta de procesar todos o parte de los elementos de una secuencia y devolver una lista con los resultados. `resultado = ["0x%02x" % x for x in range(256) if x % 2 == 0]` genera una lista de cadenas que contienen los números hexadecimales (0x...) pares del rango 0..255. La cláusula `if` es opcional. Si se hubiese omitido, se habrían procesado todos los elementos de `range(256)`.

metaclase

La clase de las clases. Las definiciones de clases generan un nombre de clase, un diccionario de la clase y una lista de clases base. La metaclase es responsable de tomar estos tres argumentos y crear la clase. La mayoría de los lenguajes de programación orientados a objetos proporcionan una implementación predeterminada. Lo que hace especial a Python en que es posible crear metaclases a medida. La mayoría de los usuarios nunca hacen uso de esta herramienta, pero si surge la necesidad, las metaclases permiten soluciones potentes y elegantes. Se han utilizado para registrar los accesos a atributos, añadir seguridad multihilo, trazar la creación de objetos, implementar el patrón singleton y muchas otras tareas.

mutable
Los objetos mutables pueden cambiar su valor pero mantener su `id()`. Ver también *inmutable*.

número complejo

Una extensión del habitual sistema de números reales en el que todos los números se expresan como una suma de una parte real y una imaginaria. Los números imaginarios son múltiplos reales de la unidad imaginaria (la raíz cuadrada de `-1`), que se escribe `i` en matemáticas o `j` en ingeniería. Python tiene soporte de serie para números complejos, que se escriben en la segunda notación; la parte imaginaria se escribe con un sufijo `j`, por ejemplo, `3+1j`. Para acceder a los equivalentes complejos del módulo `math`, usa `cmath`. El uso de números complejos es una técnica relativamente avanzada. Si no sabes si los necesitas, es casi seguro que puedes hacerles caso omiso con tranquilidad.

Python3000
Una versión mítica de Python, a la que se permitiría no ser compatible hacia atrás, con interfaz telepática.

__slots__
Una declaración dentro de una *clase moderna* que ahorra memoria al declarar el espacio para guardar los atributos de las instancias y eliminar los diccionarios de instancia. A pesar de ser popular, la técnica es enrevesada de aplicar correctamente y se reserva para los casos infrecuentes en que hay una gran cantidad de instancias de un objeto en una aplicación crítica en memoria.

secuencia
Un *iterable* a cuyos elementos se puede acceder de manera eficiente mediante índices enteros, a través de los métodos especiales `__getitem__()` y `__len__()`. Ejemplos de tipos secuencia internos son `list`, `str`, `tuple` y `unicode`. Hay que destacar que `dict` también permite `__getitem__()` y `__len__()`, pero se considera una correspondencia más que una secuencia porque las búsquedas se realizan con claves arbitrarias *inmutables* en lugar de enteros.

Zen de Python
Enumeración de los principios de diseño y la filosofía de Python útiles para comprender y utilizar el lenguaje. Esta lista se puede obtener tecleando ``` `import this` " en el intérprete interactivo.

Release 2.4.1a0, documentation updated on septiembre 11, 2005.
Consultar en Acerca de este documento... información para sugerir cambios.