Tutorial de Python

Release:	2.5.2
Date:	July 21, 2009

Python es un lenguaje de programación poderoso y fácil de aprender. Cuenta con estructuras de datos eficientes y de alto nivel y un enfoque simple pero efectivo a la programación orientada a objetos. La elegante sintaxis de Python y su tipado dinámico, junto con su naturaleza interpretada, hacen de éste un lenguaje ideal para scripting y desarrollo rápido de aplicaciones en diversas áreas y sobre la mayoría de las plataformas.

El intérprete de Python y la extensa biblioteca estándar están a libre disposición en forma binaria y de código fuente para las principales plataformas desde el sitio web de Python, http://www.python.org/, y puede distribuirse libremente. El mismo sitio contiene también distribuciones y enlaces de muchos módulos libres de Python de terceros, programas y herramientas, y documentación adicional.

El intérprete de Python puede extenderse fácilmente con nuevas funcionalidades y tipos de datos implementados en C o C++ (u otros lenguajes accesibles desde C). Python también puede usarse como un lenguaje de extensiones para aplicaciones personalizables.

Este tutorial introduce de manera informal al lector a los conceptos y características básicas del lenguaje y el sistema de Python. Es bueno tener un interprete de Python a mano para experimentar, sin embargo todos los ejemplos están aislados, por lo tanto el tutorial puede leerse estando desconectado.

Para una descripción de los objetos y módulos estándar, mira la Referencia de la Biblioteca de Python. El Manual de Referencia de Python provee una definición más formal del lenguaje. Para escribir extensiones en C o C++, lee Extendiendo e Integrando el Intérprete de Python y la Referencia de la API Python/C. Hay también numerosos libros que tratan a Python en profundidad.

Este tutorial no pretende ser exhaustivo ni tratar cada una de las características, o siquiera las características más usadas. En cambio, introduce la mayoría de las características más notables de Python, y te dará una buena idea del gusto y estilo del lenguaje. Luego de leerlo, serás capaz de leer y escribir módulos y programas en Python, y estarás listo para aprender más de los variados módulos de la biblioteca de Python descriptos en la Referencia de la Biblioteca de Python.

También vale la pena mirar el *glosario*.

Saciando tu apetito

Si trabajás mucho con computadoras, eventualmente encontrarás que te gustaría automatizar alguna tarea. Por ejemplo, podrías desear realizar una búsqueda y reemplazo en un gran número de archivos de texto, o renombrar y reorganizar un montón de archivos con fotos de una manera compleja. Tal vez quieras escribir alguna pequeña base de datos personalizada, o una aplicación especializada con interfaz gráfica, o un juego simple.

Si sos un desarrollador de software profesional, tal vez necesites trabajar con varias bibliotecas de C/C++/Java pero encuentres que se hace lento el ciclo usual de escribir/-compilar/testear/recompilar. Tal vez estás escribiendo una batería de pruebas para una de esas bibliotecas y encuentres que escribir el código de testeo se hace una tarea tediosa. O tal vez has escrito un programa al que le vendría bien un lenguaje de extensión, y no quieres diseñar/implementar todo un nuevo lenguaje para tu aplicación.

Python es el lenguaje justo para ti.

Podrías escribir un script (o programa) en el interprete de comandos o un archivo por lotes de Windows para algunas de estas tareas, pero los scripts se lucen para mover archivos de un lado a otro y para modificar datos de texto, no para aplicaciones con interfaz de usuario o juegos. Podrías escribir un programa en C/C++/Java, pero puede tomar mucho tiempo de desarrollo obtener al menos un primer borrador del programa. Python es más fácil de usar, está disponible para sistemas operativos Windows, MacOS X y Unix, y te ayudará a realizar tu tarea más velozmente.

Python es fácil de usar, pero es un lenguaje de programación de verdad, ofreciendo mucho mucho mayor estructura y soporte para programas grandes que lo que lo que pueden ofrecer los scripts de Unix o archivos por lotes. Por otro lado, Python ofrece mucho más chequeo de error que C, y siendo un *lenguaje de muy alto nivel*, tiene tipos de datos de alto nivel incorporados como arreglos de tamaño flexible y diccionarios. Debido a sus tipos de datos más generales Python puede aplicarse a un dominio de problemas mayor que Awk o incluso Perl, y aún así muchas cosas siguen siendo al menos igual de fácil en Python que en esos lenguajes.

Python te permite separar tu programa en módulos que pueden reusarse en otros programas en Python. Viene con una gran colección de módulos estándar que puedes usar como base de tus programas, o como ejemplos para empezar a aprender a programar en Python. Algunos de estos módulos proveen cosas como entrada/salida a archivos, llamadas al sistema, sockets, e incluso interfaces a sistemas de interfaz gráfica de usuario como Tk.

Python es un lenguaje interpretado, lo cual puede ahorrarte mucho tiempo durante el desarrollo ya que no es necesario compilar ni enlazar. El intérprete puede usarse interactivamente, lo que facilita experimentar con características del lenguaje, escribir programas descartables, o probar funciones cuando se hace desarrollo de programas de abajo hacia arriba. Es también una calculadora de escritorio práctica.

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

- los tipos de datos de alto nivel permiten expresar operaciones complejas en una sola instrucción
- la agrupación de instrucciones se hace por snagría en vez de llaves de apertura y cierre
- no es necesario declarar variables ni argumentos.

Python es *extensible*: si ya sabes programar en C es fácil agregar una nueva función o módulo al intérprete, ya sea para realizar operaciones críticas a velocidad máxima, o para enlazar programas Python con bibliotecas que tal vez sólo estén disponibles en forma binaria (por ejemplo bibliotecas gráficas específicas de un fabricante). Una vez que estés realmente entusiasmado, podés enlazar el intérprete Python en una aplicación hecha en C y usarlo como lenguaje de extensión o de comando para esa aplicación.

Por cierto, el lenguaje recibe su nombre del programa de televisión de la BBC "Monty Python's Flying Circus" y no tiene nada que ver con reptiles. Hacer referencias a sketches de Monty Python en la documentación no sólo esta permitido, ¡sino que también está bien visto!

Ahora que ya estás emocionado con Python, querrás verlo en más detalle. Como la mejor forma de aprender un lenguaje es usarlo, el tutorial te invita a que juegues con el intérprete de Python a medida que vas leyendo.

En el próximo capítulo se explicará la mecánica de uso del intérprete. Esta es información bastante mundana, pero es esencial para poder probar los ejemplos que aparecerán más adelante.

El resto del tutorial introduce varias características del lenguaje y el sistema Python a través de ejemplos, empezando con expresiones, instrucciones y tipos de datos simples, pasando por funciones y módulos, y finalmente tocando conceptos avanzados como excepciones y clases definidas por el usuario.

Usando el Intérprete de Python

Invocando al Intérprete

Por lo general, el intérprete de Python se instala en file:*/usr/local/bin/python* en las máquinas dónde está disponible; poner `/usr/local/bin` en el camino de búsqueda de tu intérprete de comandos Unix hace posible iniciarlo tipeando el comando:

```
python
```

en la terminal. Ya que la elección del directorio dónde vivirá el intérprete es una opción del proceso de instalación, puede estar en otros lugares; consultá a tu Gurú Python local o administrador de sistemas. (Por ejemplo, `/usr/local/python` es una alternativa popular).

En máquinas con Windows, la instalación de Python por lo general se encuentra en `C:\Python26`, aunque se puede cambiar durante la instalación. Para añadir este directorio al camino, puedes tipear el siguiente comando en el prompt de DOS:

```
set path=%path%;C:\python26
```

Se puede salir del intérprete con estado de salida cero tipeando el caracter de fin de archivo (`Control-D` en Unix, `Control-Z` en Windows) en el prompt primario. Si esto no funciona, se puede salir del intérprete tipeando el siguiente comando: `import sys; sys.exit()`.

Las características para editar líneas del intérprete no son muy sofisticadas. En Unix, quien instale el intérpreté tendrá habilitado el soporte para la biblioteca GNU readlines, que añade una edición interactiva más elaborada e historia. Tal vez la forma más rápida de detectar si las características de edición están presentes es tipear Control-P en el primer prompt de Python que aparezca. Si se escucha un beep, las características están presentes; ver Apéndice *tut-interacting* para una introducción a las teclas. Si no pasa nada, o si aparece `^P`, estas características no están disponibles; solo vas a poder usar backspace para borrar los caracteres de la línea actual.

La forma de operar del intérprete es parecida a la línea de comandos de Unix:cuando se la llama con la entrada estándar conectada a un dispostivo tty, lee y ejecuta comandos en forma interactiva; cuando es llamada con un nombre de archivo como argumento o con un archivo como entrada estánddar, lee y ejecuta un *script* del archivo.

Una segunda forma de iniciar el intérprete es `python -c command [arg] ...`, que ejecuta las sentencias en *command*, similar a la opción *-c* de la línea de comandos. Ya que las sentencias de Python suelen tener espacios en blanco u otros caracteres que son especiales en la línea de comandos, es mejor citar *command* entre comillas dobles.

Algunos módulos de Python son también útiles como scripts. Pueden ser invocados

usando `python -m module [arg] ...`, que ejecuta el código de *module* como si se hubiese tipeado su nombre completo en la línea de comandos.

Notá que existe una diferencia entre `python file` y `python <file`. En el último caso, la entrada solicitada por el programa, como en llamadas a `input()` y `raw_input()`, son satisfechas desde *file*. Ya que este archivo ya fue leído hasta el final por el analizador antes de que el programa empiece su ejecución, se encontrará el fin de archivo enseguida. En el primer caso (lo que usualmente vas a querer) son satisfechas por cualquier archivo o dispositivo que esté conectado a la entrada estándar del intérprete de Python.

Cuando se usa un script, a veces es útil correr primero el script y luego entrar al modo interactivo. Esto se puede hacer pasándole la opción *-i* antes del nombre del script. (Esto no funciona si el script es leido desde la entrada estándar, por la misma razón explicada en el párrafo anterior).

Pasaje de Argumentos

Cuando son conocidos por el intérprete, el nombre del escript y los argumentos adicionales son entonces pasados al script en la variable `sys.argv`, una lista de cadenas de texto. Su logitud es al menos uno; cuando ningún script o argumentos son pasados, `sys.argv[0]` es una cadena vacía. Cuando se pasa el nombre del script con `'-'` (lo que significa la entrada estándar), `sys.argv[0]` vale `'-'`. Cuando se usa *-c command*, `sys.argv[0]` vale `'-c'`. Cuando se usa *-m module*, `sys.argv[0]` toma el valor del nombre completo del módulo. Las opciones encontradas luego de *-c command* o *-m module* no son consumidas por el procesador de opciones de Python pero de todas formas almacenadas en `sys.argv` para ser manejadas por el comando o módulo.

Modo Interactivo

Se dice que estamos usando el intérprete en modo interactivo, cuando los comandos son leídos desde una tty. En este modo espera el siguiente comando con el *prompt primario*, usualmente tres signos mayor-que (>>>); para las líneas de continuación espera con el *prompt secundario*, por defecto tres puntos (`...`). Antes de mostrar el prompt primario, el intérprete muestra un mensaje de bienvenida reportando su número de versión y una nota de copyright:

```
python
Python 2.6 (#1, Feb 28 2007, 00:02:06)
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Las líneas de continuación son necesarias cuando queremos ingresar un constructor multi-línea. Como en el ejemplo, mirá la sentencia `if`:

```
>>> el_mundo_es_plano = 1
>>> if el_mundo_es_plano:
...     print "¡Tené cuidado de no caerte!"
...
¡Tené cuidado de no caerte!
```

El Intérprete y su Entorno

Manejo de Errores

Cuando ocurre un error, el intérprete imprime un mensaje de error y la traza del error. En el modo interactivo, luego retorna al prompt primario; cuando la entrada viene de un archivo, el programa termina con código de salida distinto a cero luego de imprimir la traza del error. (Las excepciones manejadas por una clausula `except` en una sentecina a `try` no son errores en este contexto). Algunos errores son incondicionalmente fatales y causan una terminación con código de salida distinto de cero; esto se debe

ainconcistencias internas o a que

el intérprete se queda sin memoria. Todos los mensajes de error se escriben en el flujo de errores estándar; las salidas normales de comandos ejecutados se escribe en la salida estándar.

Al tipear el caracter de interrupción (por lo general Control-C o DEL) en el prompt primario o secundario, se cancela la entrada y retorna al promt primario. [1] Tipear una interrupción mientras un comando se están ejecutando lanza la excepción `KeyboardInterrupt`, que puede ser manejada con una sentencia `try`.

Scripts Python Ejecutables

En los sistemas Unix tipo BSD, los scripts Python pueden convertirse directamente en ejecutables, como scripts del intérprete de comandos, poniendo la linea:

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

al principio del script y dándole al archivo permisos de ejecución (asumiendo que el intérprete están en la variable de entorno **PATH** del usuario). `#!` deben ser los primeros dos caracteres del archivo. En algunas plataformas, la primer línea debe terminar al estilo Unix (`'\n'`), no como en Mac OS (`'\r'`) o Windows (`'\r\n'`). Notá que el caracter numeral `'#'` se usa en Python para comenzar un comentario.

Se le puede dar permisos de ejecución al script usando el comando **chmod**:

```
$ chmod +x myscript.py
```

En sistemas Windows, no existe la noción de "modo ejecutable". El instalador de Python asocia automáticamente la extensión `.py` con `python.exe` para que al hacerle doble click a un archivo Python se corra el script. La extensión también puede ser `.pyw`, en este caso, la ventana con la consola que normalmente aparece es omitida.

Codificación del Código Fuente

Es posible utilizar una codifición distinta a ASCII en el código fuente de Pyhton. La mejor forma de hacerlo es poner otro comentario especial enseguida después de la línea con `#!` para definir la codificación:

```
# -*- coding: encoding -*-
```

Con esa declaración, todos los caracteres en el archivo fuente serán traducidos utilizando la codificación *encoding*, y será posible escribir directamente cadenas de texto literales Unicode en la codificación seleccionada. La lista de posibles codificaciones se puede encontrar en la Referencia de la Biblioteca de Python, en la sección sobre `codecs`.

Por ejemplo, para esrcibir literales Unicode, incluyendo el símbolo de la moneda Euro, se puede usar la codificación ISO-8859-15, en la que el símbolo Euro tiene el valor 164. Este script imprimirá el valor 8364 (el código Unicode correspondiente al símbolo Euro) y luego saldrá:

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

moneda = u"€"
print ord(moneda)
```

Si tu editor tiene soporte para guardar archivos como `UTF-8` con *marca de orden de byte* UTF-8 (también conocida como BOM), podés usar eso en lugar de la declaración de codificación. IDLE lo soporta si se activa `Options/General/Default Source Encoding/UTF-8`. Notá que esto no funciona en versiones antiguas de Python (2.2 y anteriores), ni por el sistema operativo en scripts con la línea con `#!` (solo usado en sistemas Unix).

Usando UTF-8 (ya sea mediante BOM o la declaración de codificación), los caracteres de la mayoría de los idiomas del mundo pueden ser usados simultaneamente en cadenas de texto o comentarios. No se soporta usar carcteres no-ASCII en identificadores. Para mostrar todos estos caracteres en forma apropiada, tu editor debe reconocer que el archivo es UTF-8, y debe usar una fuente que soporte todos los caracteres del archivo.

El Archivo de Inicio Interactivo

Cuando usás Python en forma interactiva, suele ser útil que algunos comandos estándar se ejecuten cada vez que el intérprete se inicia. Podés hacer esto configurando la variable de entorno **PYTHONSTARTUP** con el nombre de un archivo que contenga tus comandos de inicio. Esto es similar al archivo `.profile` en los intérpretes de comandos de Unix.

Este archivo es solo leído en las sesiones interactivas del intérprete, no cuando Python leer comandos de un script ni cuando file*:/dev/tty* se explicita como una fuente de comandos (que de otro modo se comporta como una sesión interactiva). Se ejecuta en el mismo espacio de nombres en el que los comandos interactivos se ejecutan, entonces los objetos que define o importa pueden ser usandos sin cualificaciones en la sesión interactiva. En este archivo también podés cambiar los prompts `sys.ps1` y `sys.ps2`.

Sin querés leer un archivo de inicio adicional desde el directorio actual, podés programarlo en el archivo de inicio global usando algo como `if os.path.isfile('.pythonrc.py'): execfile('.pythonrc.py')`. Si querés usar el archivo de inicio en un script, tenés que hacer lo siguiente en forma explífica en el script:

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

Footnotes

1 Un problema con el paquete GNU Readline puede evitar que funcione.

Una introducción informal a Python

En los siguientes ejemplos, las entradas y salidas son distinguidas por la presencia o ausencia de los prompts (`>>>` and `...`): para reproducir los ejemplos, debés escribir todo lo que esté después del prompt, cuando este aparezca; las líneas que no comiencen con el prompt son las salidas del intérprete. Tené en cuenta que el prompt secundario que aparece por si sólo en una línea de un ejemplo significa que debés escribir una línea en blanco; esto es usado para terminar un comando multilínea.

Muchos de los ejemplos de este manual, incluso aquellos ingresados en el prompt interactivo, incluyen comentarios. Los comentarios en Python comienzan con el carácter numeral, #, y se extienden hasta el final físico de la línea. Un comentario quizás aparezca al comienzo de la línea o seguidos de espacios blancos o código, pero sin una cadena de caracteres. Un carácter numeral dentro de una cadena de caracteres es sólo un carácter numeral.

Algunos ejemplos:

```
# este es el primer comentario
SPAM = 1                 # y este es el segundo comentario
                         # ... y ahora un tercero!
STRING = "# Este no es un comentario".
```

Usar Python como una calculadora

Vamos a probar algunos comandos simples en Python. Iniciá un intérprete y esperá por el prompt primario, >>>. (No debería demorar tanto).

Números

El intérprete actúa como una simple calculadora; podés ingrsar una expresión y este escribirá los valores. La sintaxis es sencilla: los operadores +, -, * y / funcionan como en la mayoría de los lenguajes (por ejemplo, Pascal o C); los paréntesis pueden ser usados para agrupar. Por ejemplo:

```
>>> 2+2
4
>>> # Este es un comentario
... 2+2
4
>>> 2+2  # y un comentario en la misma línea que el código
4
>>> (50-5*6)/4
```

```
5
>>> # La división entera retorna redondeado al piso:
... 7/3
2
>>> 7/-3
-3
```

El signo igual (=) es usado para asignar un valor a una variable. Luego, ningún resultado es mostrado antes del próximo prompt:

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

Un valor puede ser asignado a varias variables simultáneamente:

```
>>> x = y = z = 0  # Cero a x, y, y z
>>> x
0
>>> y
0
>>> z
0
```

Se soporta completamente los números de punto flotante; las operaciones con mezclas en los tipos de los operandos convierten los enteros a punto flotante:

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

Los números complejos también están soportados; los números imaginarios son escritos con el sufijo de `j` o `J`. Los números complejos con un componente real que no sea cero son escritos como `(real+imagj)`, o pueden ser escrito 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 son siempre representados como dos números de punto flotante, la parte real y la imaginaria. Para extraer estas partes desde un número complejo *z*, usá `z.real` y `z.imag`.

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

La función de conversión de los punto flotante y enteros (`float()`, `int()` y `long()`) no funciona para números complejos; aquí no hay una forma correcta de convertir un número complejo a un número real. Usá `abs(z)` para obtener esta magnitud (como un flotante) o `z.real` para obtener la 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 abs(z)
>>> a.real
3.0
>>> a.imag
4.0
>>> abs(a)  # sqrt(a.real**2 + a.imag**2)
5.0
>>>
```

En el modo interactivo, la última expresión impresa es asignada a la variable _. Esto significa que cuando estés usando Python como una calculadora de escritorio, es más fácil seguir calculando, por ejemplo:

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

Esta variable debería ser tratada como de sólo lectura por el usuario. No le asignes

explícitamente un valor; crearás una variable local independiente con el mismo nombre enmascarando la variable con el comportamiento mágico.

Cadenas de caracteres

Además de números, Python puede manipular cadenas de texto, las cuales pueden ser expresadas de distintas formas. Pueden estar encerradas en comillas simples o dobles:

```
>>> 'huevos y pan'
'huevos y pan'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Si," le dijo.'
'"Si," le dijo.'
>>> "\"Si,\" le dijo."
'"Si," le dijo.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

Las cadenas de texto literales pueden contener múltiples líneas de distintas formas. Las líneas continuas se pueden usar, con una barra invertida como el último carácter de la línea para indicar que la siguiente línea es la continuación lógica de la línea:

```
hola = "Esta es una larga cadena que contiene\n\
varias líneas de texto, tal y como se hace en C.\n\
    Notar que los espacios en blanco al principio de la linea\
 son significantes."

print hola
```

Notá que de todas formas se necesita embeber los salto de líneas con `\n`; la nueva línea que sigue a la barra invertida final es descartada. Este ejemplo imprimiría:

```
Esta es una larga cadena que contiene
varias líneas de texto, tal y como se hace en C.
     Notar que los espacios en blanco al principio de la linea son
     significantes.
```

Si se hace de la cadena de texto una cadena "cruda", la secuencia `\n` no es convertida a salto de línea, pero la barra invertida al final de la línea y el carácter de nueva línea en la fuente, ambos son incluidos en la cadena como datos. Así, el ejemplo:

```
hola = r"Esta es una larga cadena que contiene\n\
varias líneas de texto, tal y como se hace en C."
```

```
print hola
```

...imprimirá:

```
Esta es una larga cadena que contiene\n\
varias líneas de texto, tal y como se hace en C.
```

O, las cadenas de texto pueden ser rodeadas en un par de comillas triples: """ o '''. No se necesita escapar los finales de línea cuando se utilizan comillas triples, pero serán incluidos en la cadena.

```
print """
Uso: algo [OPTIONS]
     -h                        Muestra el mensaje de uso
     -H nombrehost             Nombre del host al cual conectarse
"""
```

...produce la siguiente salida:

```
Uso: algo [OPTIONS]
     -h                        Muestra el mensaje de uso
     -H nombrehost             Nombre del host al cual conectarse
```

El interprete imprime el resultado de operaciones entre cadenas de la misma forma en que son tecleadas como entrada: dentro de comillas, y con comillas y otros caracteres raros escapados con barras invertidas, para mostrar el valor preciso. La cadena de texto es encerrada con comillas dobles si contiene una comilla simple y no comillas dobles, sino es encerrada con comillas simples. (La declaración `print`, descrita luego, puede ser usado para escribir cadenas sin comillas o escapes).

Las cadenas de texto pueden ser concatenadas (pegadas juntas) con el operador + y repetidas con *:

```
>>> palabra = 'Ayuda' + 'A'
>>> palabra
'AyudaA'
>>> '<' + palabra*5 + '>'
'<AyudaAAyudaAAyudaAAyudaAAyudaA>'
```

Dos cadenas de texto juntas son automáticamente concatenadas; la primer línea del ejemplo anterior podría haber sido escrita `palabra = 'Ayuda' 'A'`; esto solo funciona con dos literales, no con expresiones arbitrarias:

```
>>> 'cad' 'ena'                  #  <-  Esto es correcto
'cadena'
>>> 'cad'.strip() + 'ena'   #  <-  Esto es correcto
'cadena'
```

```
>>> 'cad'.strip() 'ena'     #  <-  Esto no es correcto
  File "<stdin>", line 1, in ?
    'cad'.strip() 'ena'
                      ^
SyntaxError: invalid syntax
```

Las cadenas de texto se pueden indexar; como en C, el primer carácter de la cadena tiene el índice 0. No hay un tipo de dato para los caracteres; un carácter es simplemente una cadena de longitud uno. Como en Icon, se pueden especificar subcadenas con la *notación de rebanadas*: dos índices separados por dos puntos.

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

Los índices de las rebanadas tienen valores por defecto útiles; el valor por defecto para el primer índice es cero, el valor por defecto para el segundo índice es la longitud de la cadena a rebanar.

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

A diferencia de las cadenas de texto en C, en Python no pueden ser modificadas. Intentar asignar a una posición de la cadena es un error:

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

Sin embargo, crear una nueva cadena con contenido combinado es fácil y eficiente:

```
>>> 'x' + palabra[1:]
'xyudaA'
>>> 'Mas' + palabra[5]
'MasA'
```

Algo útil de las operaciones de rebanada: `s[:i] + s[i:]` es `s`.

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

Los índices degenerados en las rebanadas son manejados bien: un índice muy largo es reemplazado por la longitud de la cadena, un límite superior más chico que el límite menor retorna una cadena vacía.

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

Los índices pueden ser números negativos, para empezar a contar desde la derecha. Por ejemplo:

```
>>> palabra[-1]     # El último caracter
'A'
>>> palabra[-2]     # El penúltimo caracter
'a'
>>> palabra[-2:]    # Los últimos dos caracteres
'aA'
>>> palabra[:-2]    # Todo menos los últimos dos caracteres
'Ayud'
```

Pero notá que -0 es en realidad lo mismo que 0, ¡por lo que no cuenta desde la derecha!

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

Los índices negativos fuera de rango son truncados, pero esto no funciona para índices de un solo elemento (no rebanada):

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

Una forma de recordar cómo funcionan las rebanadas es pensar en los índices como puntos *entre* caracteres, con el punto a la izquierda del primer carácter numerado en 0. Luego, el punto a la derecha del último carácter de una cadena de *n* caracteres tienen índice *n*, por ejemplo:

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

La primer fila de números da la posición de los índices 0...6 en la cadena; la segunda fila da los correspondientes índices negativos. La rebanada de *i* a *j* consiste en todos los caracteres entre los puntos etiquetados *i* y *j*, respectivamente.

Para índices no negativos, la longitud de la rebanada es la diferencia de los índices, si ambos entran en los límites. Por ejemplo, la longitud de `palabra[1:3]` es 2.

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

```
>>> s = 'supercalifrastilisticoespialidoso'
>>> len(s)
33
```

Ver también

typesseq
: Las cadenas de texto y la cadenas de texto Unicode descritas en la siguiente sección son ejemplos de *tipos secuencias*, y soportan las operaciones comunes para esos tipos.

string-methods
: Tanto las cadenas de texto normales como las cadenas de texto Unicode soportan una gran cantidad de métodos para transformaciones básicas y búsqueda.

new-string-formatting
: Aquí se da información sobre formateo de cadenas de texto con `str.format()`.

string-formatting
: Aquí se describe con más detalle las operaciones viejas para formateo usadas cuando una cadena de texto o una cadena Unicode están a la izquierda del operador `%`.

Cadenas de Texto Unicode

Desde la versión 2.0 de Python, se encuentra disponible un nuevo tipo de datos para que los programadores almacenen texto: el objeto Unicode. Puede ser usado para almacenar y manipular datos Unicode (ver http://www.unicode.org/) y se integran bien con los objetos existentes para cadenas de texto, mediante auto-conversión cuando es necesario.

Unicode tiene la ventaja de tener un número ordinal para cada carácter usando en cada script usando tanto en textos modernos como antiguos. Previamente, había solo 256 ordinales posibles para los caracteres en scripts. Los textos eran típicamente asociados a

un código que mapea los ordinales a caracteres en scripts. Esto lleva a mucha confunión especialmente al internacionalizar (usualmente escrito como `i18n` --- `'i'` + 18 caracteres + `'n'`) software. Unicode resuelve estos problemas definiendo una sola codificación para todos los scripts.

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

```
>>> u'¡Hola Mundo!'
u'¡Hola Mundo!'
```

La `'u'` al frente de la comilla indica que se espera una cadena Unicode. Si querés incluir caracteres especiales en la cadena, podés hacerlo usando una forma de escapar caracteres Unicode provista por Python. El siguiente ejemplo muestra cómo:

```
>>> u'¡Hola\u0020Mundo!'
u'¡Hola Mundo!'
```

La secuencia de escape `\u0020` indica que se debe insertar el carácter Unicode con valor ordinal 0x0020 (el espacio en blanco) en la posición dada.

Otros caracteres son interpretados usando su respectivo valor ordinal como ordinales Unicode. Si tenés cadenas de texto literales en la codificación estándar Latin-1 que es muy usada en países occidentales, encontrarás conveniente que los primeros 256 caracteres de Unicode son los mismos primeros 256 caracteres de Latin-1.

También existe un modo crudo para expertos, del mismo modo que con las cadenas de texto normales. Debés anteponer 'ur' a la comilla inicial para que Python use el modo de escape crudo de Unicode. Solo se aplicará la conversión `\uXXXX` si hay un número impar de barras invertidas frente a la 'u'.

```
>>> ur'!Hola\u0020Mundo!'
u'¡Hola Mundo!'
>>> ur'Hola\\u0020Mundo!'
u'¡Hola\\\\u0020Mundo!'
```

El modo crudo es útil principalmente útil cuando tenés que insertar muchas barras invertidas, como puede suceder al trabajar con expresiones regulares.

Además de estas codificaciones estándar, Python provee muchas más formas de crear cadenas de texto Unicode en las bases de codificaciones conocidas.

La función predefinida `unicode()` da acceso a todos los codecs (CODificadores y DECodificadoes). Algunos de las codificaciones más conocidas que estos codecs pueden convertir son *Latin-1*, *ASCII*, *UTF-8*, y *UTF-16*. Las dos últimas son codificaciones de longitud variable que almacenan cada carácter Unicode en uno o más bytes. La codificación por defecto es normalmente seteada a ASCII, que contiene los caracteres del rango 0-127 y rechaza cualquier otro con un error. Cuando una cadena Unicode se imprime, escribe en un archivo, o se convierte con la función `str()`, se realiza la conversión utilizando la codificación por defecto.

```
>>> 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 0-2: ordinal not in range(128)
```

Para convertir una cadena Unicode en una cadena de 8-bit utilizando una codificación en particular, los objetos Unicode tienen un método `encode()` que toma un argumento, el nombre de la codificación. Se prefieren los nombres en minúsculas para los nombres de las codificaciones.

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

Si tenés datos en una codificación en particular y querés producir la cadena Unicode correspondiente, podés usar 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'
```

Listas

Python tiene varios tipos de datos *compuestos*, usados para agrupar otros valores. El más versátil es la *lista*, la cual puede ser escrita como una lista de valores separados por coma (ítems) entre corchetes. No es necesario que los ítems de una lista tengan todos el mismo tipo.

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

Cómo los índices de las cadenas de texto, los índices de las listas comienzan en 0, y las listas pueden ser rebanadas, concatenadas y todo lo demás:

```
>>> a[0]
'pan'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['huevos', 100]
```

```
>>> a[:2] + ['carne', 2*2]
['pan', 'huevos', 'carne', 4]
>>> 3*a[:3] + ['¡Boo!']
['pan', 'huevos', 100, 'pan', 'huevos', 100, 'pan', 'huevos', 100, '¡Boo!']
```

A diferencia de las cadenas de texto, que son *inmutables*, es posible cambiar un elemento individual de una lista:

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

También es posible asignar a una rebanada, y esto incluso puede cambiar la longitud de la lista o vaciarla totalmente:

```
>>> # Reemplazar algunos elementos:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Borrar algunos:
... a[0:2] = []
>>> a
[123, 1234]
>>> # Insertar algunos:
... a[1:1] = ['bruja', 'xyzzy']
>>> a
[123, 'bruja', 'xyzzy', 1234]
>>> # Insertar (una copia de) la misma lista al principio
>>> a[:0] = a
>>> a
[123, 'bruja', 'xyzzy', 1234, 123, 'bruja', 'xyzzy', 1234]
>>> # Vaciar la lista: reemplazar todos los items con una lista vacía
>>> a[:] = []
>>> a
[]
```

La función predefinida `len()` también sirve para las listas:

```
>>> a = ['a', 'b', 'c', 'd']
>>> len(a)
4
```

Es posible anidar listas (crear listas que contengan 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('extra')     # Ver seccion 5.1
>>> p
[1, [2, 3, 'extra'], 4]
>>> q
[2, 3, 'extra']
```

Notá que en el último ejemplo, `p[1]` y `q` ¡realmente hacen referencia al mismo objeto! Volveremos a la *semántica de los objetos* más adelante.

Primeros Pasos Hacia la Programación

Por supuesto, podemos usar Python para tareas más complicadas que sumar dos y dos. Por ejemplo, podemos escribir una subsecuencia inicial de la serie de *Fibonacci* así:

```
>>> # Series de Fibonacci:
... # la suma de dos elementos define 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 primer línea contiene una *asignación múltiple*: las variables``a`` y b toman en forma simultanea los nuevos valores 0 y 1. En la última lìnea esto es vuelto a usar, demostrando que la expresión a la derecha son todas evaluadas antes de que suceda cualquier asignación. Las expresiones a la derecha son evaluadas de izquierda a derecha.

- El bucle `while` se ejecuta mientras la condición (aquí: `b < 10`) sea verdadera. En Python, como en C, cualquier entero distinto de cero es verdadero; cero es falso. La condición también puede ser una cadena de texto o una lista, de hecho cualquier secuencia; cualquier cosa con longitud distinta de cero es verdadero, las secuencias vacías son falso. La prueba usada en el ejemplo es una comparación simple. Los operadores estándar de comparación se escriben igual que en C: < (menor qué), > (mayor qué), == (igual a), <= (menor o igual qué), >= (mayor o igual qué) y `!=` (distinto a).
- El *cuerpo* del bucle está *identado*: la identación es la forma que usa Python para agrupar declaraciones. Python (¡aún!) no provee una facilidad inteligente para editar líneas, así que debés tipear un tab o espacio(s) para cada línea identada. En la práctica vas a preparar entradas más complicadas para Python con un editor de texto; la mayoría de los editores de texto tienen la facilidad de auto identar. Al entrar una declaración compuesta en forma interactiva, debés finalizar con una línea en blanco para indicar qué está completa (ya que el analizador no puede adivinar cuando tipeaste la última línea). Notá que cada línea de un bloque básico debe estar identada de la misma forma.
- La declaración `print` escribe el valor de la o las expresiones que se le pasan. Difiere se simplemente escribir la expresión que se quiere mostrar (como hicimos antes en los ejemplos de la calculadora) en la forma en que maneja múltiples expresiones y cadenas. Las cadenas de texto son impresas sin comillas, y un espacio en blanco es insertado entre los elementos, así podés formatear cosas de una forma agradable, así:

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

Una coma final evita el salto de línea al final de 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
```

Notá que el intérprete inserta un salto de línea antes de imprimir el próximo prompt si la última línea no estaba completa.

Más herramientas para Control de Flujo

Además de la sentencia `while` que acabamos de introducir, Python soporta las sentencias de control de flujo que podemos encontrar en otros lenguajes, con algunos cambios.

La Sentencia `if`

Tal vez el tipo más conocido de sentencia sea la sentencia `if`. Por ejemplo:

```
>>> x = int(raw_input("Ingresá un entero, por favor: "))
>>> if x < 0:
...      x = 0
...      print 'Negativo cambiado a cero'
... elif x == 0:
...      print 'Cero'
... elif x == 1:
...      print 'Simple'
... else:
...      print 'Mas'
...
```

Puede haber cero o más bloques `elif`, y el bloque `else` es opcional. La palabra reservada 'elif' es una abreviación de 'else if', y es útil para evitar identación excesiva. Una secuencia `if` ... `elif` ... `elif` ... sustituye las sentencias `switch` o `case` encontradas en otros lenguajes.

La Sentencia `for`

La sentencia `for` en Python difiere un poco de lo que uno puede estar acostumbrado en lenguajes como C o Pascal. En lugar de siempre iterar sobre una progresión aritmética de números (como en Pascal) o darle al usuario la posibilidad de definir tanto el paso de la iteración como la condición de fin (como en C), la sentencia `for` de Python itera sobre los items de cualquier secuencia (una lista o una cadena de texto), en el orden que aparecen en la secuencia. Por ejemplo (no pun intended):

```
>>> # Midiendo cadenas de texto
... a = ['gato', 'ventana', 'defenestrado']
>>> for x in a:
...     print x, len(x)
...
gato 4
ventana 7
defenestrado 12
```

No es seguro modificar la secuencia sobre la que se está iterando en el loop (esto solo es posible para tipos de secuencias mutables, como las listas). Si se necesita modificar la lista sobre la que se está iterando (por ejemplo, para duplicar items seleccionados) se debe iterar sobre una copia. La notación de rebanada es conveniente para esto:

```
>>> for x in a[:]: # hacer una copia por rebanada de toda la lista
...     if len(x) > 6: a.insert(0, x)
...
>>> a
['defenestrado', 'gato', 'ventana', 'defenestrado']
```

La Función `range()`

Si se necesita iterar sobre una secuencia de números, es apropiado utilizar la función incorporada `range()`. Genera una lista conteniendo progresiones aritméticas:

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

El valor final dado nunca es parte de la lista; `range(10)` genera una lista de 10 valores, los índices correspondientes para los items de una secuencia de longitud 10. Es posible hacer que el rango empiece con otro número, o especificar un incremento diferente (incluso negativo; algunas veces se lo llama 'paso'):

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

Para iterar sobre los índices de una secuencia, se combina `range()` y `len()` así:

```
>>> a = ['Mary', 'tenia', 'un', 'corderito']
>>> for i in range(len(a)):
...     print i, a[i]
...
0 Mary
1 tenia
2 un
3 corderito
```

Las Sentencias `break` y `continue`, y la Cláusula `else` en Loops

La sentencia `break`, como en C, termina el loop `for` o `while` más anidado.

La sentencia `continue`, también tomada prestada de C, continua con la próxima iteración del loop.

Las sentencias de loop pueden tener una cláusula `else` que es ejecutada cuando el loop termina, luego de agotar la lista (con `for`) o cuando la condición se hace falsa (con `while`), pero no cuando el loop es terminado con la sentencia `break`. Se ejemplifica en el siguiente loop, que busca números primos:

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

La Sentencia `pass`

La sentencia `pass` no hace nada. Se puede usar cuando una sentencia es requerida por la sintáxis pero el programa no requiere ninguna acción. Por ejemplo:

```
>>> while True:
...       pass # Espera ocupada hasta iterrupción de teclado
...
```

Definiendo funciones

Podemos crear una función que escriba la serie de Fibonacci hasta un límite determinado:

```
>>> def fib(n):    # escribe la serie de Fibonacci hasta n
...     """Escribe la serie de Fibonacci hasta n."""
...     a, b = 0, 1
...     while b < n:
...         print b,
...         a, b = b, a+b
...
>>> # Ahora llamamos a la funcion que acabamos de definir:
... fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

La palabra reservada `def` se usa para *definir* funciones. Debe seguirle el nombre de la función y la lista de parámetros formales entre paréntesis. Las sentencias que forman el cuerpo de la función empiezan en la línea siguiente, y deben estar identadas. La primer sentencia del cuerpo de la función puede ser opcionalmente una cadena de texto literal; esta es la cadena de texto de documentación de la función, o *docstring*.

Hay herramientas que usan las docstrings para producir automáticamente documentación en línea o imprimible, o para permitirle al usuario que navegue el código en forma interactiva; es una buena práctica incluir docstrings en el código que uno escribe, por lo que se debe intentar hacer un hábito de esto.

La *ejecución* de una función introduce una nueva tabla de símbolos usada para las variables locales de la función. Más precisamente, todas las asignaciones de variables en la función almacenan el valor en la tabla de símbolos local; así mismo la referencia a variables primero mira la tabla de símbolos local, luego en la tabla de símbolos local de las funciones externas, luego la tabla de símbolos global, y finalmente la tabla de nombres predefinidos. Así, no se les puede asignar directamente un valor a las variables globales dentro de una función (a menos se las nombre en la sentencia `global`), aunque si pueden ser referenciadas.

Los parámetros reales (argumentos) de una función se introducen en la tabla de símbolos local de la función llamada cuando esta es llamada; así, los argumentos son pasados *por valor* (dónde el *valor* es siempre una *referencia* a un objeto, no el valor del objeto). [2] Cuando una función llama a otra función, una nueva tabla de símbolos local es creada para esa llamada.

La definición de una función introduce el nombre de la función en la tabla de símbolos actual. El valor del nombre de la función tiene un tipo que es reconocido por el interprete como una función definida por el usuario. Este valor puede ser asignado a otro nombre que luego puede ser usado como una función. Esto sirve como un mecanismo general para renombrar:

```
>>> 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 solo funciones que no retornan un valor. De hecho, técnicamente hablando, los procedimientos si retornan un valor, aunque uno aburrido. Este valor es llamada `None` (es un nombre predefinido). El intérprete por lo general no escribe el valor `None` si va a ser el único valor escrito. Si realmente se quiere, se puede verlo usando `print`:

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

Es simple escribir una función que retorne una lista con los números de la serie de Fibonacci en lugar de imprimirlos:

```
>>> def fib2(n): # retorna la seri de Fibonacci hasta n
...     """Retorna una lista conteniendo la serie de Fibonacci hasta n."""
...     result = []
...     a, b = 0, 1
...     while b < n:
...         result.append(b)    # ver abajo
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # llamarla
>>> f100                # escribir el resultado
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Este ejemplo, como es usual, demuestra algunas características más de Python:

- La sentencia `return` devuelve un valor en una función. `return` sin una expresión como argumento retorna `None`. Si se alcanza el final de un procedimiento, también se retorna `None`.

- La sentencia `result.append(b)` llama a un *método* del objeto lista `result`. Un método es una función que 'pertenece' a un objeto y se nombra `obj.methodname`, dónde `obj` es algún objeto (puede ser una expresión), y `methodname` es el nombre del método que está definido por el tipo del objeto. Distintos tipos definen distintos métodos. Métodos de diferentes tipos pueden tener el mismo nombre sin causar ambigüedad. (Es posible definir tipos de objetos propios, y métodos, usando *clases*, como se discutirá más adelante en el tutorial). El método `append()` mostrado en el ejemplo está definido para objetos lista; añade un nuevo elemento al final de la lista. En este ejemplo es equivalente a `result = result + [b]`, pero más eficiente.

Más sobre Definición de Funciones

También es posible definir funciones con un número variable de argumentos. Hay tres formas que pueden ser combinadas.

Argumentos con Valores por Defecto

La forma más útil es especificar un valor por defecto para uno o más argumentos. Esto crea una función que puede ser llamada con menos argumentos que los que permite. Por ejemplo:

```
def pedir_confirmacion(prompt, reintentos=4, queja='Si o no, por favor!'):
    while True:
        ok = raw_input(prompt)
        if ok in ('s', 'S', 'si', 'Si', 'SI'): return True
        if ok in ('n', 'no', 'No', 'NO'): return False
        reintentos = reintentos - 1
        if reintentos < 0: raise IOError, 'usuario duro'
        print queja
```

Esta función puede ser llamada tanto así: `pedir_confirmacion('¿Realmente queres salir?')` como así: `pedir_confirmacion('¿Sobreescribir archivo?', 2)`.

Este ejemplo también introduce la palabra reservada `in`. Prueba si una secuencia contiene o no un determinado valor.

Los valores por defecto son evaluados en el momento de la definición de la función, en el ámbito de *definición*, entonces:

```
i = 5

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

```
i = 6
f()
```

imprimirá 5.

Advertencia importante: El valor por defecto es evaluado solo una vez. Existe una diferencia cuando el valor por defecto es un objeto mutable como una lista, diccionario, o instancia de la mayoría de las clases. Por ejemplo, la siguiente función acumula los argumentos que se le pasan en subsiguientes llamadas:

```
def f(a, L=[]):
    L.append(a)
    return L

print f(1)
print f(2)
print f(3)
```

Imprimirá:

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

Si no se quiere que el valor por defecto sea compartido entre subsiguientes llamadas, se pueden escribir la función así:

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

Palabras Claves como Argumentos

Las funciones también puede ser llamadas usando palabras claves como argumentos de la forma `keyword = value`. Por ejemplo, la siguiente función:

```
def loro(tension, estado='muerto', accion='explotar', tipo='Azul Nordico'):
    print "-- Este loro no va a", accion,
    print "si le aplicas", voltage, "voltios."
    print "-- Gran plumaje tiene el", tipo
    print "-- Esta", estado, "!"
```

puede ser llamada de cualquiera de las siguientes formas:

```
loro(1000)
loro(accion = 'EXPLOTARRRRR', tension = 1000000)
loro('mil', estado= 'boca arriba')
loro('un millon', 'rostizado', 'saltar')
```

pero estas otras llamadas serían todas inválidas:

```
loro()                      # falta argumento obligatorio
loro(tension=5.0, 'muerto') # argumento no-de palabra clave seguido de
                            # uno que si
loro(110, tension=220)      # valor duplicado para argumento
loro(actor='Juan Garau')    # palabra clave desconocida
```

En general, una lista de argumentos debe tener todos sus argumentos posicionales seguidos por los argumentos de palabra clave, dónde las palabras claves deben ser elegidas entre los nombres de los parámetros formales. No es importante si un parámetro formal tiene un valor por defecto o no. Ningún argumento puede recibir un valor más de una vez (los nombres de parámetros formales correspondientes a argumentos posiciónales no pueden ser usados como palabras clave en la misma llamada). Aquí hay un ejemplo que falla debido a 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 un parámetro formal de la forma `**name` está presente al final, recive un diccionario (ver *typesmapping*) conteniendo todos los argumentos de palabras clave excepto aquellos correspondientes a un parámetro formal. Esto puede ser combinado con un parámetro formal de la forma `*name` (descripto en la siguiente subsección) que recibe una tupla conteniendo los argumentos posicionales además de la lista de parámetros formales. (`*name` debe ocurrir antes de `**name`). Por ejemplo, si definimos una función así:

```
def ventadequeso(tipo, *argumentos, **palabrasclaves):
    print "-- ¿Tiene", tipo, '?'
    print "-- Lo siento, nos quedamos sin", kind
    for arg in argumentos: print arg
    print '-'*40
    claves = palabrasclaves.keys()
    claves.sort()
    for c in claves: print c, ':', palabrasclaves[c]
```

Puede ser llamada así:

```
ventadequeso('Limburger', "Es muy liquito, sr.",
             "Realmente es muy muy liquido, sr.",
             cliente='Juan Garau',
             vendedor='Miguel Paez',
             puesto='Venta de Queso Argentino')
```

y por supuesto imprimirá:

```
-- ¿Tiene Limburger ?
-- Lo siento, nos quedamos sin Limburger
Es muy liquito, sr.
Realmente es muy muy liquido, sr.
----------------------------------------
cliente : Juan Garau
vendedor : Miguel Paez
puesto : Venta de Queso Argentino
```

Se debe notar que el método `sort()` de la lista de nombres de argumentos de palabra clave es llamado antes de imprimir el contenido del diccionario `palabrasclaves`; si esto no se hace, el orden en que los argumentos son impresos no está definido.

Listas de Argumentos Arbitrarios

Finalmente, la opción menos frecuentemente usada es especificar que una función puede ser llamada con un número arbitrario de argumentos. Estos argumentos serán organizados en una tupla. Antes del número variable de argumentos, cero o más argumentos normales pueden estar presentes.:

```
def fprintf(file, template, *args):
    file.write(template.format(args))
```

Desempaquetando una Lista de Argumentos

La situación inversa ocurre cuando los argumentos ya están en una lista o tupla pero necesitan ser desempaquetados para llamar a una función que requiere argumentos posicionales separados. Por ejemplo, la función predefinida `range()` espera los argumentos *inicio* y *fin*. Si no están disponibles en forma separada, se puede escribir la llamada a la función con el operador para desempaquetar argumentos de una lista o una tupla *::

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

Del mismo modo, los diccionarios pueden entregar argumentos de palabra clave con el operador **::

```
>>> def loro(tension, estado='rostizado', accion='explotar'):
...     print "-- Este loro no va a", accion,
...     print "si le aplicas", voltage, "voltios.",
...     print "Esta", estado, "!"
...
>>> d = {"tension": "cuatro millones", "estado": "demacrado",
         "accion": "VOLAR"}
>>> loro(**d)
-- Este loro no va a VOLAR si le aplicas cuatro millones
   voltios. Esta demacrado !
```

Formas con Lambda

Por demanda popular, algunas características comúnmente encontradas en lenguajes de programación funcionales como Lisp fueron añadidas a Python. Con la palabra reservada `lambda` se pueden crear pequeñas funciones anónimas. Esta es una función que retorna la suma de sus dos argumentos: `lambda a, b: a+b`. Las formas con lambda pueden ser usadas en cualquier lugar que se requieran funciones. Semánticamente, son solo azúcar sintáctica para la definición de funciones. Cómo en la definición de funciones anidadas, las formas con lambda pueden referenciar variables del ámbito en el que son contenidas::

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

Cadenas de texto de Documentación

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

La primer línea debe ser siempre un resumen corto y conciso del propósito del objeto. Para ser breve, no se debe mencionar explícitamente el nombre o tipo del objeto, ya que estos están disponibles de otros modos (excepto si el nombre es un verbo que describe el funcionamiento de la función). Esta línea debe empezar con una letra mayúscula y terminar

con un punto.

Si hay más líneas en la cadena de texto de documentación, la segunda línea debe estar en blanco, separando visualmente el resumen del resto de la descripción. Las líneas siguientes deben ser uno o más párrafos describiendo las convenciones para llamar al objeto, efectos secundarios, etc.

El analizador de Python no quita la identación de las cadenas de texto literales multi-líneas, entonces las herramientas que procesan documentación tienen que quitar la identación si así lo quieren. Esto se hace mediante la siguiente convención. La primer línea que no está en blanco *siguiente* a la primer línea de la cadena determina la cantidad de identación para toda la cadena de documentación. (No podemos usar la primer línea ya que generalmente es adyacente a las comillas de apertura de la cadena y la identación no se nota en la cadena de texto). Los espacios en blanco "equivalentes" a esta identación son luego quitados del comienzo de cada línea en la cadena. No deberían haber líneas con menor identación, pero si las hay todos los espacios en blanco del comienzo deben ser quitados. La equivalencia de espacios en blanco debe ser verificada luego de la expansión de tabs (a 8 espacios, normalmente).

Este es un ejemplo de un docstring multi-línea:

```
>>> def mi_funcion():
...     """No hace mas que documentar la funcion.
...
...     No, de verdad. No hace nada.
...     """
...     pass
...
>>> print mi_funcion.__doc__
No hace mas que documentar la funcion.

No, de verdad. No hace nada.
```

Intermezzo: Estilo de Codificación

Ahora que estás a punto de escribir piezas de Python más largas y complejas, es un buen momento para hablar sobre *estilo de codificación*. La mayoría de los lenguajes pueden ser escritos (o mejor dicho, *formateados*) con diferentes estilos; algunos son mas fáciles de leer que otros. Hacer que tu código sea más fácil de leer por otros es siempre una buena idea, y adoptar un buen estilo de codificación ayuda tremendamente a lograrlo.

Para Python, **PEP 8** se erigió como la guía de estilo a la que más proyectos adhirieron; promueve un estilo de codificación fácil de leer y amable con los ojos. Todos los desarrolladores Python deben leerlo en algún momento; aquí están extraídos los puntos más importantes:

- Usar identación de 4 espacios, no tabs.

  4 espacios son un buen compromiso entre identación pequeña (permite mayor nivel de identación) e identación grande (más fácil de leer). Los tabs introducen confusión y es mejor dejarlos de lado.
- Recortar las líneas para que no superen los 79 caracteres.

  Esto ayuda a los usuarios con pantallas pequeñas y hace posible tener varios archivos de código abiertos, uno al lado del otro, en pantallas grandes.
- Usar líneas en blanco para separar funciones y clases, y bloques grandes de código dentro de funciones.
- Cuando sea posible, poner comentarios en una sola línea.
- Usar docstrings.
- Usar espacios alrededor de operadores y luego de las comas, pero no directamente dentro de paréntesis: `a = f(1, 2) + g(3, 4)`.
- Nombrar las clases y funciones consistentemente; la convención es usar `NotacionCamello` para clases y `minusculas_con_guiones_bajos` para funciones y métodos. Siempre usar `self` como el nombre para el primer argumento en los métodos.
- No usar codificaciones estrafalarias si se espera usar el código en entornos internacionales. ASCII plano funciona bien en la mayoría de los casos.

Footnotes

2 En realidad, *llamadas por referencia de objeto* sería una mejor-descripción, ya que si un objeto mutable es pasado, quien realiza la llamaba verá cualquier cambio que el llamado realice sobre el mismo (como items insertados en una lista).

Estructuras de datos

Este capítulo describe algunas cosas que ya aprendiste en más detalle, y agrega algunas cosas nuevas también.

Más sobre listas

El tipo de dato lista tiene algunos métodos más. Aquí están todos los métodos de los objetos lista:

`list.append(x)`

Agrega un ítem al final de la lista; equivale a `a[len(a):] = [x]`.

`list.extend(L)`

Extiende la lista agregándole todos los ítems de la lista dada; equivale a `a[len(a):] = L`.

`list.insert(i,x)`

Inserta un ítem en una posición dada. El primer argumento es el índice del ítem delante del cual se insertará, por lo tanto `a.insert(0, x)` inserta al principio de la lista, y `a.insert(len(a), x)` equivale a `a.append(x)`.

`list.remove(x)`

Quita el primer ítem de la lista cuyo calor sea *x*. Es un error si no existe tal ítem.

`list.pop(font face="Courier" size="8" color="#000000">[,i])`

Quita el ítem en la posición dada de la lista, y lo devuelve. Si no se especifica un índice, `a.pop()` quita y devuelve el último ítem de la lista. (Los corchetes que encierran a *i* en la firma del método denotan que el parámetro es opcional, no que deberías escribir corchetes en esa posición. Verás esta notación con frecuencia en la Referencia de la Biblioteca de Python.)

`list.index(x)`

Devuelve el índice en la lista del primer ítem cuyo valor sea *x*. Es un error si no existe tal ítem.

`list.count(x)`

Devuelve el número de veces que *x* aparece en la lista.

`list.sort()`

Ordena los ítems de la lista, in situ.

`list.reverse()`

Invierte los elementos de la lista, in situ.

Un ejemplo que usa la mayoría de los métodos de 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, 66.25, 333, 333, 1234.5]
```

Usando listas como pilas

Los métodos de lista hacen que resulte muy fácil usar una lista como una pila, donde el último elemento añadido es el primer elemento retirado ("último en entrar, primero en salir"). Para agregar un ítem a la cima de la pila, use `append()`. Para retirar un ítem de la cima de la pila, use `pop()` sin un índice explícito. Por ejemplo:

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

Usando listas como colas

También puedes usar una lista convenientemente como una cola, donde el primer elemento añadido es el primer elemento retirado ("primero en entrar, primero en salir"). Para agregar un ítem al final de la cola, use `append()`. Para retirar un ítem del frente de la pila, use `pop()` con `0` como índice. Por ejemplo:

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

Herramientas de programación funcional

Hay tres funciones integradas que son muy útiles cuando se usan con listas: `filter()`, `map()`, y `reduce()`.

`filter(funcion, secuencia)` devuelve una secuencia con aquellos ítems de la secuencia para los cuales `funcion(item)` es verdadero. Si *secuencia* es un `string` o `tuple`, el resultado será del mismo tipo; de otra manera, siempre será `list`. Por ejemplo, para calcular unos números 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(funcion, secuencia)` llama a `funcion(item)` por cada uno de los ítems de la secuencia y devuelve una lista de los valores retornados. Por ejemplo, para calcular unos 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; la función debe entonces tener tantos argumentos como secuencias haya y es llamada con el ítem correspondiente de cada secuencia (o `None` si alguna secuencia es más corta que otra). Por ejemplo:

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

`reduce(funcion, secuencia)` devuelve un único valor que se construye llamando a la función binaria *funcion* con los primeros dos ítems de la secuencia, entonces con el resultado y el siguiente ítem, y así sucesivamente. Por ejemplo, para calcular la suma de los números de 1 a 10:

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

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

Un tercer argumento puede pasarse para indicar el valor inicial. En este caso el valor inicial se devuelve para una secuencia vacía, y la función se aplica primero al valor inicial y el primer ítem de la secuencia, entonces al resultado y al siguiente ítem, y así sucesivamente. Por ejemplo,

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

No uses la definición de este ejemplo de `sum()`: ya que la sumatoria es una necesidad tan común, se provee una función integrada `sum(secuencia)` que funciona exactamente así.

Listas por comprensión

Las listas por comprensión proveen una forma concisa de crear listas sin tener que recurrir al uso de `map()`, `filter()` y/o `lambda`. La definición resultante de la lista a menudo tiende a ser más clara que las listas formadas usando esas construcciones.

Cada lista por comprensión consiste de una expresión seguida por una cláusula `for`, luego cero o más cláusulas `for` o `if`. El resultado será una lista que resulta de evaluar la expresión en el contexto de las cláusulas `for` y `if` que sigan. Si la expresión evalua a una

tupla, debe encerrarse entre paréntesis.

```
>>> frutafresca = ['  banana', '  mora de Logan ', 'maracuya  ']
>>> [arma.strip() for arma in frutafresca]
['banana', 'mora de Logan', 'maracuya']
>>> 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 requieren paréntesis para tuplas
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]
```

Las listas por comprensión son mucho más flexibles que `map()` y pueden aplicarse a expresiones complejas y funciones anidadas:

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

Listas por comprensión anidadas

Si tienes el estómago suficiente, las listas por comprensión pueden anidarse. Son una herramienta poderosa pero, como toda herramienta poderosa, deben usarse con cuidado, o ni siquiera usarse.

Considera el siguiente ejemplo de una matriz de 3x3 como una lista que contiene tres listas, una por fila:

```
>>> mat = [
...         [1, 2, 3],
...         [4, 5, 6],
```

```
...         [7, 8, 9],
...       ]
```

Ahora, si quisieras intercambiar filas y columnas, podrías usar una lista por comprensión:

```
>>> print [[fila[i] for fila in mat] for i in [0, 1, 2]]
[[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

Se debe tener cuidado especial para la lista por comprensión *anidada*:

> Para evitar aprensión cuando se anidan lista por comprensión, lee de derecha a izquierda.

Una versión más detallada de este retazo de código muestra el flujo de manera explícita:

```
for i in [0, 1, 2]:
    for fila in mat:
        print fila[i],
    print
```

En el mundo real, deberías preferir funciones predefinidas a declaraciones con flujo complejo. La función `zip()` haría un buen trabajo para este caso de uso:

```
>>> zip(*mat)
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

Ver *tut-unpacking-arguments* para detalles en el asterisco de esta línea.

La instrucción `del`

Hay una manera de quitar un ítem de una lista dado su índice en lugar de su valor: la instrucción `del`. Esta es diferente del método `pop()`, el cual devuelve un valor. La instrucción `del` también puede usarse para quitar secciones de una lista o vaciar la lista completa (lo que hacíamos antes asignando una lista vacía a la sección). 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 a[:]
>>> a
[]
```

`del` puede usarse también para eliminar variables:

```
>>> del a
```

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

Tuplas y secuencias

Vimos que las listas y cadenas tienen propiedades en común, como el indizado y las operaciones de seccionado. Estas son dos ejemplos de datos de tipo *secuencia* (ver *typesseq*). Como Python es un lenguaje en evolución, otros datos de tipo secuencia pueden agregarse. Existe otro dato de tipo secuencia estándar: la *tupla*.

Una tupla consiste de un número de valores separados por comas, por ejemplo:

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

Como puedes ver, en la salida las tuplas siempre se encierran entre paréntesis, para que las tuplas anidadas puedan interpretarse correctamente; pueden ingresarse con o sin paréntesis, aunque a menudo los paréntesis son necesarios de todas formas (si la tupla es parte de una expresión más grande).

Las tuplas tienen muchos usos. Por ejemplo: pares ordenados (x, y), registros de empleados de una base de datos, etc. Las tuplas, al igual que las cadenas, son inmutables: no es posible asignar a los ítems individuales de una tupla (aunque puedes simular bastante ese efecto mediante seccionado y concatenación). También es posible crear tuplas que contengan objetos mutables como listas.

Un problema particular es la construcción de tuplas que contengan 0 o 1 ítem: la sintaxis presenta algunas peculiaridades para estos casos. Las tuplas vacías se construyen mediante un par de paréntesis vacío; una tupla con un ítem se construye poniendo una coma a continuación del valor (no alcanza con encerrar un único valor entre paréntesis). Feo, pero efectivo. Por ejemplo:

```
>>> vacia = ()
>>> singleton = 'hola',    # <-- notar la coma al final
>>> len(vacia)
0
>>> len(singleton)
```

```
1
>>> singleton
('hola',)
```

La declaración `t = 12345, 54321, 'hola!'` es un ejemplo de *empaquetado de tuplas*: los valores `12345`, `54321` y `'hola!'` se empaquetan juntos en una tupla.

La operación inversa también es posible:

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

Esto se llama, apropiadamente, *desempaquetado de secuencias*. El desempaquetado de secuencias requiere que la lista de variables a la izquierda tenga el mismo número de elementos que el tamaño de la secuencia. ¡Notá que la asignación múltiple es en realidad sólo una combinación de empaquetado de tuplas y desempaquetado de secuencias!

Hay una pequeña asimetría aquí: empaquetando múltiples valores siempre crea una tupla, y el desempaquetado funciona con cualquier secuencia.

Conjuntos

Python también incluye un tipo de dato para *conjuntos*. Un conjunto es una colección no ordenada y sin elementos repetidos. Los usos básicos de éstos incluyen verificación de pertenencia y eliminación de entradas duplicadas. Los conjuntos también soportan operaciones matemáticas como la unión, intersección, diferencia, y diferencia simétrica.

Una pequeña demostración:

```
>>> canasta = ['manzana', 'naranja', 'manzana', 'pera', 'naranja', 'banana']
>>> fruta = set(canasta)               # crea un conjunto sin repetidos
>>> fruta
set(['pera', 'manzana', 'banana', 'naranja'])
>>> 'naranja' in fruta                 # verificación de pertenencia rápida
True
>>> 'yerba' in fruta
False

>>> # veamos las operaciones para las letras únicas de dos palabras
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # letras únicas en a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b                              # letras en a pero no en b
set(['r', 'b', 'd'])
>>> a | b                              # letras en a o en b
```

```
set(['a', 'c', 'b', 'd', 'm', 'l', 'r', 'z'])
>>> a & b                           # letras en a y en b
set(['a', 'c'])
>>> a ^ b                           # letras en a o b pero no en ambos
set(['b', 'd', 'm', 'l', 'r', 'z'])
```

Diccionarios

Otro tipo de dato útil incluído en Python es el *diccionario* (ver *typesmapping*). Los diccionarios se encuentran a veces en otros lenguajes como "memorias asociativas" o "arreglos asociativos". A diferencia de las secuencias, que se indexan mediante un rango numérico, los diccionarios se indexan con *claves*, que pueden ser cualquier tipo inmutable; las cadenas y números siempre pueden ser claves. Las tuplas pueden usarse como claves si solamente contienen cadenas, números o tuplas; si una tupla contiene cualquier objeto mutable directa o indirectamente, no puede usarse como clave. No podés usar listas como claves, ya que las listas pueden modificarse usando asignación por índice, asignación por sección, o métodos como `append()` y `extend()`.

Lo mejor es pensar en un diccionario como un conjunto no ordenado de pares *clave: valor*, con el requerimiento de que las claves sean únicas (dentro de un diccionario en particular). Un par de llaves crean un diccionario vacío: `{}`. Colocar una lista de pares clave:valor separados por comas entre las llaves añade pares clave:valor iniciales al diccionario; esta también es la forma en que los diccionarios se presentan en la salida.

Las operaciones principales sobre un diccionario son guardar un valor con una clave y extraer ese valor dada la clave. También es posible borrar un par clave:valor con `del`. Si usás una clave que ya está en uso para guardar un valor, el valor que estaba asociado con esa clave se pierde. Es un error extraer un valor usando una clave no existente.

El método `keys()` de un diccionario devuelve una lista de todas las claves en uso de ese diccionario, en un orden arbitrario (si la querés ordenada, simplemente usá el metodo `sort()` sobre la lista de claves). Para verificar si una clave está en el diccionario, utilizá la palabra clave `in`.

Un pequeño ejemplo de uso de un diccionario:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'jack': 4098, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
```

```
{'jack': 4098, 'irv': 4127, 'guido': 4127}
>>> tel.keys()
['jack', 'irv', 'guido']
>>> 'guido' in tel
True
```

El constructor `dict()` crea un diccionario directamente desde listas de pares clave-valor guardados como tuplas. Cuando los pares siguen un patrón, se puede especificar de forma compacta la lista de pares clave-valor por comprensión.

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
>>> dict([(x, x**2) for x in (2, 4, 6)])     # use a list comprehension
{2: 4, 4: 16, 6: 36}
```

Más adelante en este tutorial, aprenderemos acerca de Expresiones Generadoras que están mejor preparadas para la tarea de proveer pares clave-valor al constructor `dict()`.

Cuando las claves son cadenas simples, a veces resulta más fácil especificar los pares usando argumentos por palabra clave:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

Técnicas de iteración

Cuando iteramos sobre diccionarios, se pueden obtener al mismo tiempo la clave y su valor correspondiente usando el método `iteritems()`.

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

Cuando se itera sobre una secuencia, se puede obtener el índice de posición junto a su valor correspondiente usando la función `enumerate()`.

```
>>> for i, v in enumerate(['ta', 'te', 'ti']):
...     print i, v
...
0 ta
1 te
```

```
2 ti
```

Para iterar sobre dos o más secuencias al mismo tiempo, los valores pueden emparejarse con la función `zip()`.

```
>>> preguntas = ['nombre', 'objetivo', 'color favorito']
>>> respuestas = ['lancelot', 'el santo grial', 'azul']
>>> for p, r in zip(preguntas, respuestas):
...     print 'Cual es tu {0}?  {1}.'.format(p, r)
...
Cual es tu nombre?  lancelot.
Cual es tu objetivo?  el santo grial.
Cual es tu color favorito?  azul.
```

Para iterar sobre una secuencia en orden inverso, se especifica primero la secuencia al derecho y luego se llama a la función `reversed()`.

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

Para iterar sobre una secuencia ordenada, se utiliza la función `sorted()` la cual devuelve una nueva lista ordenada dejando a la original intacta.

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

Más acerca de condiciones

Las condiciones usadas en las instrucciones `while` e `if` pueden contener cualquier operador, no sólo comparaciones.

Los operadores de comparación `in` y `not in` verifican si un valor está (o no está) en una secuencia. Los operadores `is` e `is not` comparan si dos objetos son realmente el mismo objeto; esto es significativo sólo para objetos mutables como las listas. Todos los

operadores de comparación tienen la misma prioridad, la cual es menor que la de todos los operadores numéricos.

Las comparaciones pueden encadenarse. Por ejemplo, `a < b == c` verifica si a es menor que b y además si b es igual a c.

Las comparaciones pueden combinarse mediante los operadores booleanos `and` y `or`, y el resultado de una comparación (o de cualquier otra expresión booleana) puede negarse con `not`. Estos tienen prioridades menores que los operadores de comparación; entre ellos `not` tiene la mayor prioridad y `or` la menor, o sea que `A and not B or C` equivale a `(A and (not B)) or C`. Como siempre, los paréntesis pueden usarse para expresar la composición deseada.

Los operadores booleanos `and` y `or` son los llamados operadores *cortocircuito*: sus argumentos se evalúan de izquierda a derecha, y la evaluación se detiene en el momento en que se determina su resultado. Por ejemplo, si `A` y `C` son verdaderas pero `B` es falsa, en `A and B and C` no se evalúa la expresión `C`. Cuando se usa como un valor general y no como un booleano, el valor devuelto de un operador cortocircuito es el último argumento evaluado.

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

```
>>> cadena1, cadena2, cadena3 = '', 'Trondheim', 'Paso Hammer'
>>> non_nulo = cadena1 or cadena2 or cadena3
>>> non_nulo
'Trondheim'
```

Notá que en Python, a diferencia de C, la asignación no puede ocurrir dentro de expresiones. Los programadores de C pueden renegar por esto, pero es algo que evita un tipo de problema común encontrado en programas en C: escribir = en una expresión cuando lo que se quiere escribir es ==.

Comparando secuencias y otros tipos

Las secuencias pueden compararse con otros objetos del mismo tipo de secuencia. La comparación usa orden *lexicográfico*: primero se comparan los dos primeros ítems, si son diferentes esto ya determina el resultado de la comparación; si son iguales, se comparan los siguientes dos ítems, y así sucesivamente hasta llegar al final de alguna de las secuencias. Si dos ítems a comparar son ambos secuencias del mismo tipo, la comparación lexicográfica es recursiva. Si todos los ítems de dos secuencias resultan iguales, se considera que las secuencias son iguales. Si una secuencia es una subsecuencia inicial de la otra, la secuencia más corta es la menor. El orden lexicográfico para cadenas de caracteres utiliza el orden ASCII para caracteres individuales. Algunos 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)
```

Observá que comparar objetos de diferentes tipos es legal. El resultado es determinístico pero arbitrario: los tipos se ordenan por su nombre. Por lo tanto, una lista (`list`) siempre evalúa como menor que una cadena (`string`).

[3] Tipos numéricos diferentes se comparan a su valor numérico, o sea 0 es igual a 0.0, etc.

Footnotes

3 No confiar demasiado en las reglas para comparar objetos de diferentes tipos; pueden cambiar en una versión futura del lenguaje.

Módulos

Si salís del intérprete de Python y entrás de nuevo, las definiciones que hiciste (funciones y variables) se pierden. Por lo tanto, si querés escribir un programa más o menos largo, es mejor que uses un editor de texto para preparar la entrada para el interprete y ejecutarlo con ese archivo como entrada. Esto es conocido como crear un *guión*, o *script*. Si tu programa se vuelve más largo, quizás quieras separarlo en distintos archivos para un mantenimiento más fácil. Quizás también quieras usar una función útil que escribiste desde distintos programas sin copiar su definición a cada programa.

Para soportar esto, Python tiene una manera de poner definiciones en un archivo y usarlos en un script o en una instancia interactiva del intérprete. Tal archivo es llamado *módulo*; las definiciones de un módulo pueden ser *importadas* a otros módulos o al módulo *principal* (la colección de variables a las que tenés acceso en un script ejecutado en el nivel superior y en el modo calculadora).

Un módulo es una archivo conteniendo definiciones y declaraciones de Python. El nombre del archivo es el nombre del módulo con el sufijo `.py` agregado. Dentro de un módulo, el nombre del mismo (como una cadena) está disponible en el valor de la variable global `__name__`. Por ejemplo, usá tu editor de textos favorito para crear un archivo llamado `fibo.py` en el directorio actual, con el siguiente contenido:

```
# módulo de números Fibonacci

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

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

Ahora entrá al intérprete de Python e importá este módulo con la siguiente orden:

```
>>> import fibo
```

Esto no mete los nombres de las funciones definidas en `fibo` directamente en el espacio de nombres actual; sólo mete ahí el nombre del módulo, `fibo`. Usando el nombre del módulo podés 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 pensás usar la función frecuentemente, podés asignarla a un nombre local:

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

Más sobre los módulos

Un módulo puede contener tanto declaraciones ejecutables como definiciones de funciones. Estas declaraciones están pensadas para inicializar el módulo. Se ejecutan solamente la *primera* vez que el módulo se importa en algún lado. [4]

Cada módulo tiene su propio espacio de nombres, el que es usado como espacio de nombres global por todas las funciones definidas en el módulo. Por lo tanto, el autor de un módulo puede usar variables globales en el módulo sin preocuparse acerca de conflictos con una variable global del usuario. Por otro lado, si sabés lo que estás haciendo podés tocar las variables globales de un módulo con la misma notación usada para referirte a sus funciones, `nombremodulo.nombreitem`.

Los módulos pueden importar otros módulos. Es costumbre pero no obligatorio el ubicar todas las declaraciones `import` al principio del módulo (o script, para el caso). Los nombres de los módulos importados se ubican en el espacio de nombres global del módulo que hace la importación.

Hay una variante de la declaración `import` que importa los nombres de un módulo directamente al espacio de nombres del módulo que hace la importación. 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 en el espacio de nombres local el nombre del módulo desde el cual se está importando (entonces, en el ejemplo, `fibo` no se define).

Hay incluso una variante para importar todos los nombres que un módulo define:

```
>>> 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 aquellos que comienzan con un subrayado (_).

> ### Note
>
> Por razones de eficiencia, cada módulo se importa una vez por sesión del intérprete. Por lo tanto, si modificás los módulos, tenés que reiniciar el intérprete -- o, si es sólo un módulo que querés probar interactivamente, usá `reload()`, por ejemplo `reload(nombremodulo)`.

Ejecutando módulos como scripts

Cuando ejecutás un módulo de Python con

```
python fibo.py <argumentos>
```

...el código en el módulo será ejecutado, tal como si lo hubieses importado, pero con `__name__` con el valor de `"__main__"`. Eso significa que agregando este código al final de tu módulo:

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

...podés hacer que el archivo sea utilizable tanto como script, como módulo importable, porque el código que analiza la linea de órdenes sólo se ejecuta si el módulo es ejecutado como archivo principal:

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

Si el módulo se importa, ese código no se ejecuta:

```
>>> import fibo
>>>
```

Esto es frecuentemente usado para proveer al módulo una interfaz de usuario conveniente, o para propósitos de prueba (ejecutar el módulo como un script ejecuta el juego de pruebas).

El camino de búsqueda de los módulos

Cuando se importa un módulo llamado `spam`, el intérprete busca un archivo llamado `spam.py` en el directorio actual, y luego en la lista de directorios especificada por la variable de entorno **PYTHONPATH**. Esta tiene la misma sintáxis que la variable de shell

PATH, o sea, una lista de nombres de directorios. Cuando **PYTHONPATH** no está configurada, o cuando el archivo no se encuentra allí, la búsqueda continua en un camino por omisión que depende de la instalación; en Unix, este es normalmente `.:/usr/lib/python`.

En realidad, los módulos se buscan en la lista de directorios dada por la variable `sys.path`, la cual se inicializa con el directorio que contiene al script de entrada (o el directorio actual), **PYTHONPATH**, y el directorio default dependiente de la instalación. Esto permite que los programas en Python que saben lo que están haciendo modifiquen o reemplacen el camino de búsqueda de los módulos. Notar que como el directorio que contiene el script que se ejecuta está en el camino de búsqueda, es importante que el script no tenga el mismo nombre que un módulo estándar, o Python intentará cargar el script como un módulo cuando ese módulo se importe. Esto generalmente será un error. Mirá la sección *tut-standardmodules* para más información.

Archivos "compilados" de Python

Como una importante aceleración del tiempo de arranque para programas cortos que usan un montón de los módulos estándar, si un archivo llamado `spam.pyc` existe en el directorio donde se encuentra `spam.py`, se asume que contiene una versión ya "compilada a byte" del módulo `spam` (lo que se denomina *bytecode*). La fecha y hora de modificación del archivo `spam.py` usado para crear `spam.pyc` se graba en este último, y el `.pyc` se ignora si estos no coinciden.

Normalmente, no necesitás hacer nada para crear el archivo `spam.pyc`. Siempre que se compile satisfactoriamente el `spam.py`, se hace un intento de escribir la versión compilada al `spam.pyc`. No es un error si este intento falla, si por cualquier razón el archivo no se escribe completamente el archivo `spam.pyc` resultante se reconocerá como inválido luego. El contenido del archivo `spam.pyc` es independiente de la plataforma, por lo que un directorio de módulos puede ser compartido por máquinas de diferentes arquitecturas.

Algunos consejos para expertos:

- Cuando se invoca el intérprete de Python con la opción *-O*, se genera código optimizado que se almacena en archivos `.pyo`. El optimizador actualmente no ayuda mucho; sólo remueve las declaraciones `assert`. Cuando se usa *-O*, se optimiza *todo* el *bytecode*; se ignoran los archivos `.pyc` y los archivos `.py` se compilan a bytecode optimizado.
- Pasando dos opciones *-O* al intérprete de Python (*-OO*) causará que el compilador realice optimizaciones que en algunos raros casos podría resultar en programas que funcionen incorrectamente. Actualmente, solamente se remueven del bytecode a las cadenas `__doc__`, resultando en archivos `.pyo` más compactos. Ya que algunos programas necesitan tener disponibles estas cadenas, sólo deberías usar esta opción si sabés lo que estás haciendo.

- Un programa no corre más rápido cuando se lee de un archivo `.pyc` o `.pyo` que cuando se lee del `.py`; lo único que es más rápido en los archivos `.pyc` o `.pyo` es la velocidad con que se cargan.
- Cuando se ejecuta un script desde la linea de órdenes, nunca se escribe el bytecode del script a los archivos `.pyc` o `.pyo`. Por lo tanto, el tiempo de comienzo de un script puede reducirse moviendo la mayor parte de su código a un módulo y usando un pequeño script de arranque que importe el módulo. También es posible nombrar a los archivos `.pyc` o `.pyo` directamente desde la linea de órdenes.
- Es posible tener archivos llamados `spam.pyc` (o `spam.pyo` cuando se usa la opción *-O*) sin un archivo `spam.py` para el mismo módulo. Esto puede usarse para distribuir el código de una biblioteca de Python en una forma que es moderadamente difícil de hacerle ingeniería inversa.
- El módulo `compileall` puede crear archivos `.pyc` (o archivos `.pyo` cuando se usa la opción *-O*) para todos los módulos en un directorio.

Módulos estándar

Python viene con una biblioteca de módulos estándar, descrita en un documento separado, la Referencia de la Biblioteca de Python (de aquí en más, "Referencia de la Biblioteca"). Algunos módulos se integran en el intérprete; estos proveen acceso a operaciones que no son parte del núcleo del lenguaje pero que sin embargo están integrados, tanto por eficiencia como para proveer acceso a primitivas del sistema operativo, como llamadas al sistema. El conjunto de tales módulos es una opción de configuración el cual también depende de la plataforma subyacente. Por ejemplo, el módulo `winreg` sólo se provee en sistemas Windows. Un módulo en particular merece algo de atención: `sys`, el que está integrado en todos los intérpretes de Python. Las variables `sys.ps1` y `sys.ps2` definen las cadenas usadas como cursores primarios y secundarios:

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

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

La variable `sys.path` es una lista de cadenas que determinan el camino de búsqueda del intérprete para los módulos. Se inicializa por omisión a un camino tomado de la variable de entorno **PYTHONPATH**, o a un valor predefinido en el intérprete si **PYTHONPATH** no

está configurada. Lo podés modificar usando las operaciones estándar de listas:

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

La función `dir()`

La función integrada `dir()` se usa para encontrar qué nombres define un módulo. Devuelve una lista ordenada de cadenas:

```
>>> 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', 'callstats', 'copyright',
 'displayhook', 'exc_clear', 'exc_info', 'exc_type', 'excepthook',
 'exec_prefix', 'executable', 'exit', 'getdefaultencoding', 'getdlopenflags',
 'getrecursionlimit', 'getrefcount', 'hexversion', 'maxint', 'maxunicode',
 'meta_path', 'modules', 'path', 'path_hooks', 'path_importer_cache',
 'platform', 'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setdlopenflags',
 'setprofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout',
 'version', 'version_info', 'warnoptions']
```

Sin argumentos, `dir()` lista los nombres que tenés actualmente definidos:

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

Notá que lista todos los tipos de nombres: variables, módulos, funciones, etc.

`dir()` no lista los nombres de las funciones y variables integradas. Si querés una lista de esos, están definidos en el módulo estándar `__builtin__`:

```
>>> import __builtin__
>>> dir(__builtin__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'DeprecationWarning',
 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
 'FloatingPointError', 'FutureWarning', 'IOError', 'ImportError',
 'IndentationError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
 'LookupError', 'MemoryError', 'NameError', 'None', 'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError',
 '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']
```

Paquetes

Los paquetes son una manera de estructurar los espacios de nombres de Python usando "nombres de módulos con puntos". Por ejemplo, el nombre de módulo `A.B` designa un submódulo llamado `B` en un paquete llamado `A`. Tal como el uso de módulos evita que los autores de diferentes módulos tengan que preocuparse de los respectivos nombres de variables globales, el uso de nombres de módulos con puntos evita que los autores de paquetes de muchos módulos, como NumPy o la Biblioteca de Imágenes de Python (Python Imaging Library, o PIL), tengan que preocuparse de los respectivos nombres de módulos.

Suponete que querés designar una colección de módulos (un "paquete") para el manejo uniforme de archivos y datos de sonidos. Hay diferentes formatos de archivos de sonido (normalmente reconocidos por su extensión, por ejemplo: `.wav`, `.aiff`, `.au`), por lo que tenés que crear y mantener una colección siempre creciente de módulos para la conversión entre los distintos formatos de archivos. Hay muchas operaciones diferentes que quizás quieras ejecutar en los datos de sonido (como mezclarlos, añadir eco, aplicar una función ecualizadora, crear un efecto estéreo artificial), por lo que ademas estarás escribiendo una lista sin fin de módulos para realizar estas operaciones. Aquí hay una posible estructura para tu paquete (expresados en términos de un sistema jerárquico de archivos):

```
sound/                          Paquete superior
      __init__.py               Inicializa el paquete de sonido
      formats/                  Subpaquete para conversiones de formato
              __init__.py
              wavread.py
              wavwrite.py
```

```
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                 Subpaquete para efectos de sonido
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                 Subpaquete para filtros
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

Al importar el paquete, Python busca a través de los directorios en `sys.path`, buscando el subdirectorio del paquete.

Los archivos `__init__.py` se necesitan para hacer que Python trate los directorios como que contienen paquetes; esto se hace para prevenir directorios con un nombre común, como `string`, de esconder sin intención a módulos válidos que se suceden luego en el camino de búsqueda de módulos. En el caso más simple, `__init__.py` puede ser solamente un archivo vacío, pero también puede ejecutar código de inicialización para el paquete o configurar la variable `__all__`, descrita luego.

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

```
import sound.effects.echo
```

Esto carga el submódulo `sound.effects.echo`. Debe hacerse referencia al mismo con el nombre completo.

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

Otra alternativa para importar el submódulos es:

```
from sound.effects import echo
```

Esto también carga el submódulo `echo`, lo deja disponible sin su prefijo de paquete, por lo que puede usarse así:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Otra variación más es importar la función o variable deseadas directamente:

```
from sound.effects.echo import echofilter
```

De nuevo, esto carga el submódulo `echo`, pero deja directamente disponible a la función `echofilter()`:

```
echofilter(input, output, delay=0.7, atten=4)
```

Notá que al usar `from package import item` el ítem puede ser tanto un submódulo (o subpaquete) del paquete, o algún otro nombre definido en el paquete, como una función, clase, o variable. La declaración `import` primero verifica si el ítem está definido en el paquete; si no, asume que es un módulo y trata de cargarlo. Si no lo puede encontrar, se genera una excepción `ImportError`.

Por otro lado, cuando se usa la sintaxis como `import item.subitem.subsubitem`, cada ítem excepto el último debe ser un paquete; el mismo puede ser un módulo o un paquete pero no puede ser una clase, función o variable definida en el ítem previo.

*Importando * desde un paquete*

Ahora, ¿qué sucede cuando el usuario escribe `from sound.effects import *`? Idealmente, uno esperaría que esto de alguna manera vaya al sistema de archivos, encuentre cuales submódulos están presentes en el paquete, y los importe a todos. Desafortunadamente, esta operación no funciona muy bien en las plataformas Windows, donde el sistema de archivos no siempre tiene información precisa sobre mayúsculas y minúsculas. En estas plataformas, no hay una manera garantizada de saber si el archivo *ECHO.PY* debería importarse como el módulo `echo`, `Echo` o `ECHO`. (Por ejemplo, Windows 95 tiene la molesta costumbre de mostrar todos los nombres de archivos con la primer letra en mayúsculas.) La restricción de DOS de los nombres de archivos con la forma 8+3 agrega otro problema interesante para los nombres de módulos largos.

La única solución es que el autor del paquete provea un índice explícito del paquete. La declaración `import` usa la siguiente convención: si el código del `__init__.py` de un paquete define una lista llamada `__all__`, se toma como la lista de los nombres de módulos que deberían ser importados cuando se hace `from package import *`. Es tarea del autor del paquete mantener actualizada esta lista cuando se libera una nueva versión del paquete. Los autores de paquetes podrían decidir no soportarlo, si no ven un uso para importar `*` en sus paquetes. Por ejemplo, el archivo `sounds/effects/__init__.py` podría contener el siguiente código:

```
__all__ = ["echo", "surround", "reverse"]
```

Esto significaría que `from sound.effects import *` importaría esos tres submódulos del paquete `sound`.

Si no se define `__all__`, la declaración `from sound.effects import *` *no* importa todos los submódulos del paquete `sound.effects` al espacio de nombres actual; sólo

se asegura que se haya importado el paquete `sound.effects` (posiblemente ejecutando algún código de inicialización que haya en `__init__.py`) y luego importa aquellos nombres que estén definidos en el paquete. Esto incluye cualquier nombre definido (y submódulos explícitamente cargados) por `__init__.py`. También incluye cualquier submódulo del paquete que pudiera haber sido explícitamente cargado por declaraciones `import` previas. Considerá este código:

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

En este ejemplo, los módulos *echo* y *surround* se importan en el espacio de nombre actual porque están definidos en el paquete `sound.effects` cuando se ejecuta la declaración `from...import`. (Esto también funciona cuando se define `__all__`).

Notá que en general la práctica de importar `*` desde un módulo o paquete no se recomienda, ya que frecuentemente genera un código con mala legibilidad. Sin embargo, está bien usarlo para ahorrar tecleo en sesiones interactivas, y algunos módulos están diseñados para exportar sólo nombres que siguen ciertos patrones.

Recordá que no está mal usar `from paquete import submodulo_especifico`! De hecho, esta notación se recomienda a menos que el módulo que estás importando necesite usar submódulos con el mismo nombre desde otros paquetes.

Referencias internas en paquetes

Los submódulos frecuentemente necesitan referirse unos a otros. Por ejemplo, el módulo `surround` quizás necesite usar el módulo `echo` module. De hecho, tales referencias son tan comunes que la declaración `import` primero mira en el paquete actual antes de mirar en el camino estándar de búsqueda de módulos. Por lo tanto, el módulo `surround` puede simplemente hacer `import echo` o `from echo import echofilter`. Si el módulo importado no se encuentra en el paquete actual (el paquete del cual el módulo actual es un submódulo), la declaración `import` busca en el nivel superior por un módulo con el nombre dado.

Cuando se estructuran los paquetes en subpaquetes (como en el ejemplo `sound`), podés usar `import` absolutos para referirte a submódulos de paquetes hermanos. Por ejemplo, si el módulo `sound.filters.vocoder` necesita usar el módulo `echo` en el paquete `sound.effects`, puede hacer `from sound.effects import echo`.

Desde Python 2.5, además de los `import` relativos implícitos descritos arriba, podés escribir `import` relativos explícitos con la declaración de la forma `from module import name`. Estos `import` relativos explícitos usan puntos adelante para indicar los paquetes actual o padres involucrados en el `import` relativo. En el ejemplo `surround`, podrías hacer:

```
from . import echo
from .. import formats
from ..filters import equalizer
```

Notá que ambos `import`, relativos explícitos e implícitos, se basan en el nombre del módulo actual. Ya que el nombre del módulo principal es siempre `"__main__"`, los módulos pensados para usarse como módulo principal de una aplicación Python siempre deberían usar `import` absolutos.

Paquetes en múltiple directorios

Los paquetes soportan un atributo especial más, `__path__`. Este se inicializa, antes de que el código en ese archivo se ejecute, a una lista que contiene el nombre del directorio donde está el paquete. Esta variable puede modificarse, afectando búsquedas futuras de módulos y subpaquetes contenidos en el paquete.

Aunque esta característica no se necesita frecuentemente, puede usarse para extender el conjunto de módulos que se encuentran en el paquete.

Footnotes

4 De hecho las definiciones de función son también 'declaraciones' que se 'ejecutan'; la ejecución mete el nombre de la función en el espacio de nombres global.

Entrada y salida

Hay diferentes métodos de presentar la salida de un programa; los datos pueden ser impresos de una forma legible por humanos, o escritos a un archivo para uso futuro. Este capítulo discutirá algunas de las posibilidades.

Formateo elegante de la salida

Hasta ahora encontramos dos maneras de escribir valores: *declaraciones de expresión* y la declaración `print`. (Una tercer manera es usando el método `write()` de los objetos tipo archivo; el archivo de salida estándar puede referenciarse como `sys.stdout`. Mirá la Referencia de la Biblioteca para más información sobre esto.)

Frecuentemente querrás más control sobre el formateo de tu salida que simplemente imprimir valores separados por espacios. Hay dos maneras de formatear tu salida; la primera es hacer todo el manejo de las cadenas vos mismo, usando rebanado de cadenas y operaciones de concatenado podés crear cualquier forma que puedas imaginar. El módulo `string` contiene algunas operaciones útiles para emparejar cadenas a un determinado ancho; estas las discutiremos en breve. La otra forma es usar el método `str.format()`.

Nos queda una pregunta, por supuesto: ¿cómo convertís valores a cadenas? Afortunadamente, Python tiene maneras de convertir cualquier valor a una cadena: pasalos a las funciones `repr()` o `str()`. Comillas invertidas (``) son equivalentes a la `repr()`, pero no se usan más en código actual de Python y se eliminaron de versiones futuras del lenguaje.

La función `str()` devuelve representaciones de los valores que son bastante legibles por humanos, mientras que `repr()` genera representaciones que pueden ser leídas por el el intérprete (o forzarían un `SyntaxError` si no hay sintáxis equivalente). Para objetos que no tienen una representación en particular para consumo humano, `str()` devolverá el mismo valor que `repr()`. Muchos valores, como números o estructuras como listas y diccionarios, tienen la misma representación usando cualquiera de las dos funciones. Las cadenas y los números de punto flotante, en particular, tienen dos representaciones distintas.

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) + ', y es ' + repr(y) + '...'
>>> print s
El valor de x es 32.5, y es 40000...
>>> # El repr() de una cadena agrega apóstrofos y barras invertidas
... hola = 'hola mundo\n'
>>> holas = repr(hola)
>>> print holas
'hello, world\n'
>>> # El argumento de repr() puede ser cualquier objeto Python:
... repr((x, y, ('carne', 'huevos')))
"(32.5, 40000, ('carne', 'huevos'))"
```

Acá hay dos maneras de escribir una tabla de cuadrados y cubos:

```
>>> for x in range(1, 11):
...     print repr(x).rjust(2), repr(x*x).rjust(3),
...     # notar la coma al final de la linea 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 '{0:2d} {1:3d} {2:4d}'.format(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
```

(Notar que en el primer ejemplo, un espacio entre cada columna fue agregado por la manera en que `print` trabaja: siempre agrega espacios entre sus argumentos)

Este ejemplo muestra el método `rjust()` de los objetos cadena, el cual ordena una cadena a la derecha en un campo del ancho dado llenándolo con espacios a la izquierda. Hay métodos similares `ljust()` y `center()`. Estos métodos no escriben nada, sólo devuelven una nueva cadena. Si la cadena de entrada es demasiado larga, no la truncan, sino la devuelven intacta; esto te romperá la alineación de tus columnas pero es normalmente mejor que la alternativa, que te estaría mintiendo sobre el valor. (Si realmente querés que se recorte, siempre podés agregarle una operación de rebanado, como en `x.ljust(n)[:n]`.)

Hay otro método, `zfill()`, el cual rellena una cadena numérica a la izquierda con ceros. Entiendo acerca de signos positivos y negativos:

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

El uso básico del método `str.format()` es como esto:

```
>>> print 'Somos los {0} quienes decimos "{1}!"'.format('caballeros', 'Nop')
Somos los caballeros quienes decimos "Nop!"
```

Las llaves y caracteres dentro de las mismas (llamados campos de formato) son reemplazadas con los objetos pasados en el método `format()`. El número en las llaves se refiere a la posición del objeto pasado en el método.

```
>>> print '{0} y {1}'.format('carne', 'huevos')
carne y huevos
>>> print '{1} y {0}'.format('carne', 'huevos')
huevos y carne
```

Si se usan argumentos nombrados en el método `format()`, sus valores serán referidos usando el nombre del argumento.

```
>>> print 'Esta {comida} es {adjetivo}.'.format(comida='carne', adjetivo='espantosa')
Esta carne es espantosa.
```

Se pueden combinar arbitrariamente argumentos posicionales y nombrados:

```
>>> print 'La historia de {0}, {1}, y {otro}.'.format('Bill', 'Manfred', otro='Georg')
La hostoria de Bill, Manfred, y Georg.
```

Un `':'` y especificador de formato opcionales pueden ir luego del nombre del campo. Esto aumenta el control sobre cómo el valor es formateado. El siguiente ejemplo trunca Pi a tres lugares luego del punto decimal.

```
>>> import math
>>> print 'El valor de PI es aproximadamente {0:.3f}.'.format(math.pi)
El valor de PI es aproximadamente 3.142.
```

Pasando un entero luego del `':'` causará que que el campo sea de un mínimo número de caracteres de ancho. Esto es útil para hacer tablas lindas.

```
>>> tabla = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for nombre, telefono in tabla.items():
...     print '{0:10} ==> {1:10d}'.format(nombre, telefono)
...
Jack       ==>       4098
Dcab       ==>       7678
Sjoerd     ==>       4127
```

Si tenés una cadena de formateo realmente larga que no querés separar, podría ser bueno que puedas hacer referencia a las variables a ser formateadas por el nombre en vez de la posición. Esto puede hacerse simplemente pasando el diccionario y usando corchetes `'[]'` para acceder a las claves

```
>>> tabla = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print 'Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; Dcab: {0[Dcab]:d}'.format(tabla)
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Esto se podría también hacer pasando la tabla como argumentos nombrados con la notación '**'.:

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

Esto es particularmente útil en combinación con la nueva función integrada `vars()`, que devuelve un diccionario conteniendo todas las variables locales.

Para una completa descripción del formateo de cadenas con `str.format()`, mirá en *formatstrings*.

Viejo formateo de cadenas

El operador `%` también puede usarse para formateo de cadenas. Interpreta el argumento de la izquierda con el estilo de formateo de `sprintf` para ser aplicado al argumento de la derecha, y devuelve la cadena resultante de esta operación de formateo. Por ejemplo:

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

```
El valor de PI es aproximadamente 3.142.
```

Ya que `str.format()` es bastante nuevo, un montón de código Python todavía usa el operador `%`. Sin embargo, ya que este viejo estilo de formateo será eventualmente eliminado del lenguaje, en general debería usarse `str.format()`.

Podés encontrar más información en la sección *string-formatting*.

Leyendo y escribiendo archivos

La función `open()` devuelve un objeto archivo, y es normalmente usado con dos argumentos: `open(nombre_de_archivo, modo)`.

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

El primer argumento es una cadena conteniendo el nombre de archivo. El segundo argumento es otra cadena conteniendo unos pocos caracteres que describen la forma en que el archivo será usado. El *modo* puede ser `'r'` cuando el archivo será solamente leído, `'w'` para sólo escribirlo (un archivo existente con el mismo nombre será borrado), y `'a'` abre el archivo para agregarle información; cualquier dato escrito al archivo será automáticamente agregado al final. `'r+'` abre el archivo tanto para leerlo como para escribirlo. El argumento *modo* es opcional; si se omite se asume `'r'`.

En Windows y la Macintosh, agregando `'b'` al modo abre al archivo en modo binario, por lo que también hay modos como `'rb'`, `'wb'`, y `'r+b'`. Windows hace una distinción entre archivos binarios y de texto; los caracteres de fin de linea en los archivos de texto son automáticamente apenas alterados cuando los datos son leídos o escritos. Esta modificación en bambalinas para guardar datos está bien con archivos de texto ASCII, pero corromperá datos binarios como en archivos `JPEG` o `EXE`. Se muy cuidadoso en usar el modo binario al leer y escribir tales archivos. En Unix, no hay problema en agregarle una `'b'` al modo, por lo que podés usarlo independientemente de la plataforma para todos los archivos binarios.

Métodos de los objetos Archivo

El resto de los ejemplos en esta sección asumirán que ya se creó un objeto archivo llamado `f`.

Para leer el contenido de una archivo llamá a `f.read(cantidad)`, el cual lee alguna cantidad de datos y los devuelve como una cadena. *cantidad* es un argumento numérico opcional. Cuando se omite *cantidad* o es negativo, el contenido entero del archivo será leido y devuelto; es tu problema si el archivo es el doble de grande que la memoria de tu máquina. De otra manera, a lo sumo una *cantidad* de bytes son leídos y devueltos. Si se

alcanzó el fin del archivo, `f.read()` devolverá una cadena vacía (`""`).

```
>>> f.read()
'Este es el archivo entero.\n'
>>> f.read()
''
```

`f.readline()` lee una sola linea del archivo; el caracter de fin de linea (`\n`) se deja al final de la cadena, y sólo se omite en la última linea del archivo si el mismo no termina en un fin de linea. Esto hace que el valor de retorno no se ambiguo; si `f.readline()` devuelve una cadena vacía, es que se alcanzó el fin del archivo, mientras que una linea en blanco es representada por `'\n'`, una cadena conteniendo sólo un único fin de linea.

```
>>> f.readline()
'Esta es la primer linea del archivo.\n'
>>> f.readline()
'Segunda linea del archivo\n'
>>> f.readline()
''
```

`f.readlines()` devuelve una lista conteniendo todos las lineas de datos en el archivo. Si se da un parámetro opcional *pista_tamaño*, lee esa cantidad de bytes del archivo y lo suficientemente más como para completar una linea, y devuelve las lineas de eso. Esto se usa frecuentemente para permitir una lectura por lineas de forma eficiente en archivos grandes, sin tener que cargar el archivo entero en memoria. Sólo lineas completas serán devueltas.

```
>>> f.readlines()
['Esta es la primer linea del archivo.\n', 'Segunda linea del archivo\n']
```

Una forma alternativa a leer lineas es ciclar sobre el objeto archivo. Esto es eficiente en memoria, rápido, y conduce a un código más simple:

```
>>> for linea in f:
        print linea,

Esta es la primer linea del archivo
Segunda linea del archivo
```

El enfoque alternativo es mucho más simple peor no permite un control fino. Ya que los dos enfoques manejan diferente el buffer de lineas, no deberían mezclarse.

`f.write(cadena)` escribe el contenido de la *cadena* al archivo, devolviendo `None`.

```
>>> f.write('Esto es una prueba\n')
```

Para escribir algo más que una cadena, necesita convertirse primero a una cadena:

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

`f.tell()` devuelve un entero que indica la posición actual en el archivo, medida en bytes desde el comienzo del archivo. Para cambiar la posición use `f.seek(desplazamiento, desde_donde)`. La posición es calculada agregando el *desplazamiento* a un punto de referencia; el punto de referencia se selecciona del argumento *desde_donde*. Un valor *desde_donde* de 0 mide desde el comienzo del archivo, 1 usa la posición actual del archivo, y 2 usa el fin del archivo como punto de referencia. *desde_donde* puede omitirse, el default es 0, usando el comienzo del archivo como punto de referencia.

```
>>> f = open('/tmp/archivodetrabajo', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)     # Va al sexto byte en el archivo
>>> f.read(1)
'5'
>>> f.seek(-3, 2) # Va al tercer byte antes del final
>>> f.read(1)
'd'
```

Cuando hayas terminado con un archivo, llamá a `f.close()` para cerrarlo y liberar cualquier recurso del sistema tomado por el archivo abierto. Luego de llamar `f.close()`, los intentos de usar el objeto archivo fallarán automáticamente.

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

Los objetos archivo tienen algunos métodos más, como `isatty()` y `truncate`()` que son usados menos frecuentemente; consultá la Referencia de la Biblioteca para una guía completa sobre los objetos archivo.

El módulo The `pickle`

Las cadenas pueden facilmente escribirse y leerse de un archivo. Los números toman algo más de esfuerzo, ya que el método `read()` sólo devuelve cadenas, que tendrán que ser pasadas a una función como `int()`, que toma una cadena como `'123'` y devuelve su valor numérico 123. Sin embargo, cuando querés grabar tipos de datos más complejos como listas, diccionarios, o instancias de clases, las cosas se ponen más complicadas.

En lugar de tener a los usuarios constantemente escribiendo y debugueando código para

grabar tipos de datos complicados, Python provee un módulo estándar llamado `pickle`. Este es un asombroso módulo que puede tomar casi cualquier objeto Python (¡incluso algunas formas de código Python!), y convertirlo a una representación de cadena; este proceso se llama *picklear*. Reconstruir los objetos desde la representación en cadena se llama *despicklear*. Entre que se picklea y se despicklear, la cadena que representa al objeto puede almacenarse en un archivo, o enviarse a una máquina distante por una conexión de red.

Si tenés un objeto `x`, y un objeto archivo `f` que fue abierto para escritura, la manera más simple de picklear el objeto toma una sola linea de código:

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

Para despicklear el objeto nuevamente, si `f` es un objeto archivo que fue abierto para lectura:

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

(Hay otras variantes de esto, usadas al picklear muchos objetos o cuando no querés escribir los datos pickleados a un archivo; consultá la documentación completa para `pickle` en la Referencia de la Biblioteca de Python.)

`pickle` es la manera estándar de hacer que los objetos Python puedan almacenarse y reusarse por otros programas o por una futura invocación al mismo programa; el término técnido de esto es un objeto *persistente*. Ya que `pickle` es tan ampliamente usado, muchos autores que escriben extensiones de Python toman el cuidado de asegurarse que los nuevos tipos de datos como matrices puedan ser adecuadamente pickleados y despickleados.

Errores y Excepciones

Hasta ahora los mensajes de error no habían sido más que mencionados, pero si probaste los ejemplos probablemente hayas visto algunos. Hay (al menos) dos tipos diferentes de errores: *errores de sintaxis* y *excepciones*.

Errores de Sintaxis

Los errores de sintaxis, también conocidos como errores de interpretación, son quizás el tipo de queja más común que tenés 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 repite la línea culpable y muestra una pequeña 'flecha' que apunta al primer lugar donde se detectó el error. Este es causado por (o al menos detectado en) el símbolo que *precede* a la flecha: en el ejemplo, el error se detecta en el `print`, ya que faltan dos puntos (`':'`) antes del mismo. Se muestran el nombre del archivo y el número de línea para que sepas dónde mirar en caso de que la entrada venga de un programa.

Excepciones

Incluso si la declaración o expresión es sintácticamente correcta, puede generar un error cuando se intenta ejecutarla. Los errores detectados durante la ejecución se llaman *excepciones*, y no son incondicionalmente fatales: pronto aprenderás cómo manejarlos en los programas en Python. Sin embargo, la mayoría de las excepciones no son manejadas por los programas, y resultan en mensajes de error como los mostrados aquí:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' 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 de los mensajes de error indica qué sucedió. Las excepciones vienen de distintos tipos, y el tipo se imprime como parte del mensaje: los tipos en el ejemplo son: `ZeroDivisionError`, `NameError` y `TypeError`. La cadena mostrada como tipo de la excepción es el nombre de la excepción predefinida que ocurrió. Esto es verdad para todas las excepciones predefinidas del intérprete, pero no necesita ser verdad para excepciones definidas por el usuario (aunque es una convención útil). Los nombres de las excepciones estándar son identificadores incorporados al intérprete (no son palabras clave reservadas).

El resto de la línea provee un detalle basado en el tipo de la excepción y qué la causó.

La parte anterior del mensaje de error muestra el contexto donde la excepción sucedió, en la forma de un *trazado del error* listando líneas fuente; sin embargo, no mostrará líneas leídas desde la entrada estándar.

bltin-exceptions lista las excepciones predefinidas y sus significados.

Manejando Excepciones

Es posible escribir programas que manejen determinadas excepciones. Mirá el siguiente ejemplo, que le pide al usuario una entrada hasta que ingrese un entero válido, pero permite al usuario interrumpir el programa (usando `Control-C` o lo que sea que el sistema operativo soporte); notá que una interrupción generada por el usuario se señaliza generando la excepción `KeyboardInterrupt`.

```
>>> while True:
...     try:
...         x = int(raw_input(u"Por favor ingrese un número: "))
...         break
...     except ValueError:
...         print u"Oops!  No era válido.  Intente nuevamente..."
...
```

La declaración `try` funciona de la siguiente manera:

- Primero, se ejecuta el *bloque try* (el código entre las declaración `try` y `except`).
- Si no ocurre ninguna excepción, el *bloque except* se saltea y termina la ejecución de la declaración `try`.
- Si ocurre una excepción durante la ejecución del *bloque try*, el resto del bloque se saltea. Luego, si su tipo coincide con la excepción nombrada luego de la palabra reservada `except`, se ejecuta el *bloque except*, y la ejecución continúa luego de la declaración `try`.
- Si ocurre una excepción que no coincide con la excepción nombrada en el `except`, esta se pasa a declaraciones `try` de más afuera; si no se encuentra nada que la maneje, es una *excepción no manejada*, y la ejecución se frena con un mensaje como los mostrado arriba.

Una declaración `try` puede tener más de un `except`, para especificar manejadores para distintas excepciones. A lo sumo un manejador será ejecutado. Sólo se manejan excepciones que ocurren en el correspondiente `try`, no en otros manejadores del mismo `try`. Un `except` puede nombrar múltiples excepciones usando paréntesis, por ejemplo:

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

El último `except` puede omitir nombrar qué excepción captura, para servir como comodín. Usá esto con extremo cuidado, ya que de esta manera es fácil ocultar un error real de programación. También puede usarse para mostrar un mensaje de error y luego re-generar la excepción (permitiéndole al que llama, manejar también la excepción):

```
import sys

try:
    f = open('miarchivo.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as (errno, strerror):
    print "Error E/S ({0}): {1}".format(errno, strerror)
except ValueError:
    print "No pude convertir el dato a un entero."
except:
    print "Error inesperado:", sys.exc_info()[0]
    raise
```

Las declaraciones `try` ... `except` tienen un *bloque else* opcional, el cual, cuando está presente, debe seguir a los except. Es útil para aquel código que debe ejecutarse si el *bloque try* no genera una excepción. Por ejemplo:

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

El uso de `else` es mejor que agregar código adicional en el `try` porque evita capturar accidentalmente una excepción que no fue generada por el código que está protegido por la declaración `try` ... `except`.

Cuando ocurre una excepción, puede tener un valor asociado, también conocido como el *argumento* de la excepción. La presencia y el tipo de argumento depende del tipo de excepción.

El `except` puede especificar una variable luego del nombre (o tupla) de excepción(es). La variable se vincula a una instancia de excepción con los argumentos almacenados en `instance.args`. Por conveniencia, la instancia de excepción define `__getitem__()` y `__str__()` para que se pueda acceder o mostrar los argumentos directamente, sin necesidad de hacer referencia a `.args`.

Pero se recomienda no usar `.args`. En cambio, el uso preferido es pasar un único argumento a la excepción (que puede ser una tupla se se necesitan varios argumentos) y vincularlo al atributo `message`. Uno también puede instanciar una excepción antes de generarla, y agregarle cualquier atributo que uno desee:

```
>>> try:
...     raise Exception('carne', 'huevos')
... except Exception as inst:
...     print type(inst)     # la instancia de excepción
...     print inst.args      # argumentos guardados en .args
...     print inst           # __str__ permite imprimir args directamente
...     x, y = inst          # __getitem__ permite usar args directamente
...     print 'x =', x
...     print 'y =', y
...
<type 'exceptions.Exception'>
('carne', 'huevos')
('carne', 'huevos')
x = carne
y = huevos
```

Si una excepción tiene un argumento, este se imprime como la última parte (el 'detalle') del mensaje para las excepciones que no están manejadas.

Los manejadores de excpeciones no manejan solamente las excepciones que ocurren en el *bloque try*, también manejan las excepciones que ocurren dentro de las funciones que se llaman (inclusive indirectamente) dentro del *bloque try*. Por ejemplo:

```
>>> def esto_falla():
...     x = 1/0
...
>>> try:
...     esto_falla()
... except ZeroDivisionError as detail:
...     print 'Manejando error en tiempo de ejecucion:', detail
...
Manejando error en tiempo de ejecucion: integer division or modulo by zero
```

Lanzando Excepciones

La declaración `raise` permite al programador forzar a que ocurra una excepción espe-

cifica. Por ejemplo:

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

El primer argumento de `raise` nombra la excepción que se lanzará. El segundo (opcional) especifíca el argumento de la excepción. También se podría haber escrito como `raise NameError('Hola')`. Amabas formas funcionan bien, pero parece haber una creciente preferencia de estilo por la anterior.

Si necesitas determinar cuando una excepción fue lanzada pero no intentas manejarla, una forma simplificada de la instrucción `raise` te permite relanzarla:

```
>>> try:
...     raise NameError, 'Hola'
... except NameError:
...     print 'Volo una excepcion!'
...     raise
...
Volo una excpecion!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
NameError: Hola
```

Excepciones Definidas por el Usuario

Los programas pueden nombrar sus propias excepciones creando una nueva calse excecpción. Las excepciones, típicamente, deberán derivar 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 MiError(2*2)
... except MyError as e:
...     print 'Ocurrio mi excepcion, valor:', e.valor
...
Ocurrio mi excepcion, valor: 4
>>> raise MiError, 'oops!'
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MiError: 'oops!'
```

En este ejemplo, el método `__init__()` de `Exception` fue sobrescrito. El nuevo comportamiento simplememente crea el atributo *valor*. Esto reemplaza el comportamiento por defecto de crear el atributo *args*.

Las clases de Excepciones pueden ser definidas de la misma forma que cualquier otra clase, pero usualmente se mantienen simples, a menudo solo ofreciendo un número de atributos con información sobre el error que leerán los manejadores de la excepción. Al crear un módulo que pueden lanzar varios errores distintos, una práctica común es crear una clase base para excepciones definidas en ese módulo y extenderla para crear clases excepciones específicas para distintas condiciones de error:

```
class Error(Exception):
    """Clas base para excepciones en el modulo."""
    pass

class EntradaError(Error):
    """Excepcion lanzada por errores en las entradas.

    Atributos:
        expresion -- expresion de entrada en la que ocurre el error
        mensaje -- explicacion del error
    """

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

class TransicionError(Error):
    """Lanzada cuando una operacion intenta una transicion de estado no
    permitida.

    Atributos:
        previo -- estado al principio de la transicion
        siguiente -- nuevo estado intentado
        mensaje -- explicacion de porque la transicion no esta permitida
    """

    def __init__(self, previo, siguiente, mensaje):
        self.previo = previo
        self.siguiente = siguiente
        self.mensaje = mensaje
```

La mayoría de las excepciones son definidas con nombres que terminan en "Error", similares a los nombres de las excepciones estándar.

Muchos módulos estándar definen sus propias excepciones para reportar errores que pueden ocurrir en funciones propias. Se puede encontrar más información sobre clases en el capítulo *tut-classes*.

Definiendo Acciones de Limpieza

La declaración `try` tiene otra cláusula opcional que intenta definir acciones de limpieza que deben ser ejecutadas bajo ciertas circunstancias. Por ejemplo:

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

Una *cláusula finally* siempre es ejecutada antes de salir de la declaración `try`, ya sea que una excepción haya ocurrido o no. Cuando ocurre una excepción en la cláusula `try` y no fue manejada por una cláusula `except` (o ocurrió en una cláusula `except` o `else`), es relanzada luego de que se ejecuta la cláusula `finally`. `finally` es también ejecutada "a la salida" cuando cualquier otra cláusula de la declaración the `try` es dejada via `break`, `continue` or `return`. Un ejemplo más complicado (cláusulas `except` y `finally` en la misma declaración `try` como funciona en Python 2.5):

```
>>> def dividir(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print "¡division por cero!"
...     else:
...         print "el resultado es", result
...     finally:
...         print "ejecutando la clausula finally"
...
>>> dividir(2, 1)
el resultado es 2
ejecutando la clausula finally
>>> dividir(2, 0)
¡division por cero!
ejecutando la clausula finally
>>> divide("2", "1")
```

```
ejecutando la clausula finally
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Como podés ver, la cláusula `finally` es ejecutada siempre. La excepción `TypeError` lanzada al dividir dos cadenas de texto no es manejado por la cláusula `except` y por lo tanto re lanzado luego de que se ejecuta la cláusula `finally`.

En aplicaciones reales, la cláusula `finally` es útil para liberar recursos externos (como archivos o conexiones de red), sin importar si el uso del recurso fue exitoso.

Acciones Pre-definidas de Limpieza

Algunos objetos definen acciones de limpieza estándar que llevar a cabo cuando el objeto no es más necesitado, independientemente de que las operaciones sobre el objeto hayan sido exitosas o no. Mirá el siguiente ejemplo, que intenta abrir un archivo e imprimir su contenido en la pantalla.:

```
for linea in open("miarchivo.txt"):
    print linea
```

El problema con este código es que deja el archivo abierto por un periodo de tiempo indeterminado luego de que termine de ejecutarse. Esto no es un problema en scripts simples, pero puede ser un problema en aplicaciones más grandes. La declaración `with` permite que objetos como archivos sean usados de una forma que asegure que siempre se los libera rápido y en forma correcta.

```
with open("miarchivo.txt") as f:
    for linea in f:
        print linea
```

Luego de que la declaración sea ejecutada, el archivo *f* siempre es cerrado, incluso si se encuentra un problema al procesar las líneas. Otros objetos que

Clases

El mecanismo de clases de Python agrega clases al lenguaje con un mínimo de nuevas sintáxis y semánticas. Es una mezcla de los mecanismos de clase encontrados en C++ y Modula-3. Como es cierto para los módulos, las clases en Python no ponen una barrera absoluta entre la definición y el usuario, sino que más bien se apoya en la cortesía del usuario de no "forzar la definición". Sin embargo, se mantiene el poder completo de las características más importantes de las clases: el mecanismo de la herencia de clases permite múltiples clases base, una clase derivada puede sobreescribir cualquier método de su(s) clase(s) base, y un método puede llamar al método de la clase base con el mismo nombre. Los objetos pueden tener una cantidad arbitraria de datos privados.

En terminología de C++, todos los miembros de las clases (incluyendo los miembros de datos), son *públicos*, y todas las funciones miembro son *virtuales*. No hay constructores o destructores especiales. Como en Modula-3, no hay atajos para hacer referencia a los miembros del objeto desde sus métodos: la función método se declara con un primer argumento explícito que representa al objeto, el cual se provee implícitamente por la llamada. Como en Smalltalk, las clases mismas son objetos, aunque en un más amplio sentido de la palabra: en Python, todos los tipos de datos son objetos. Esto provee una semántica para importar y renombrar. A diferencia de C++ y Modula-3, los tipos de datos integrados pueden usarse como clases base para que el usuario los extienda. También, como en C++ pero a diferencia de Modula-3, la mayoría de los operadores integrados con sintáxis especial (operadores aritméticos, de subíndice, etc.) pueden ser redefinidos por instancias de la clase.

Unas palabras sobre terminología

Sin haber una terminología universalmente aceptada sobre clases, haré uso ocasional de términos de Smalltalk y C++. (Usaría términos de Modula-3, ya que su semántica orientada a objetos es más cercanas a Python que C++, pero no espero que muchos lectores hayan escuchado hablar de él).

Los objetos tienen individualidad, y múltiples nombres (en muchos ámbitos) pueden vincularse al mismo objeto. Esto se conoce como *aliasing* en otros lenguajes. Normalmente no se aprecia esto a primera vista en Python, y puede ignorarse sin problemas cuando se maneja tipos básicos inmutables (números, cadenas, tuplas). Sin embargo, el *aliasing*, o renombrado, tiene un efecto (intencional!) sobre la semántica de código Python que involucra objetos mutables como listas, diccionarios, y la mayoría de tipos que representan entidades afuera del programa (archivos, ventanas, etc.). Esto se usa normalmente para beneficio del programa, ya que los renombres funcionan como punteros en algunos aspectos. Por ejemplo, pasar un objeto es barato ya que la implementación solamente pasa el puntero; y si una función modifica el objeto que fue pasado, el que la llama verá el cambio; esto elimina la necesidad de tener dos formas diferentes de pasar argumentos,

como en Pascal.

Alcances y espacios de nombres en Python

Antes de ver clases, primero debo decirte algo acerca de las reglas de alcance de Python. Las definiciones de clases hacen unos lindos trucos con los espacios de nombres, y necesitás saber cómo funcionan los alcances y espacios de nombres para entender por completo cómo es la cosa. De paso, los conocimientos en este tema son útiles para cualquier programador Python avanzado.

Comenzemos con unas definiciones.

Un *espacio de nombres* es un mapeo de nombres a objetos. Muchos espacios de nombres están implementados en este momento como diccionarios de Python, pero eso no se nota para nada (excepto por el desempeño), y puede cambiar en el futuro. Como ejemplos de espacios de nombres tenés: el conjunto de nombres incluídos (funciones como `abs()`, y los nombres de excepciones integradas); los nombres globales en un módulo; y los nombres locales en la invocación a una función. Lo que es importante saber de los espacios de nombres es que no hay relación en absoluto entre los nombres de espacios de nombres distintos; por ejemplo, dos módulos diferentes pueden tener definidos los dos una función "maximizar" sin confusión -- los usuarios de los módulos deben usar el nombre del módulo como prefijo.

Por cierto, yo uso la palabra *atributo* para cualquier cosa después de un punto --- por ejemplo, en la expresión `z.real`, `real` es un atributo del objeto `z`. Estrictamente hablando, las referencias a nombres en módulos son referencias a atributos: en la expresión `modulo.funcion`, `modulo` es un objeto módulo y `funcion` es un atributo de éste. En este caso hay un mapeo directo entre los atributos del módulo y los nombres globales definidos en el módulo: ¡están compartiendo el mismo espacio de nombres! [5]

Los atributos pueden ser de sólo lectura, o de escritura. En el último caso es posible la asignación a atributos. Los atributos de módulo pueden escribirse: podés escribir `modulo.la_respuesta = 42`. Los atributos de escritura se pueden borrar también con la instrucción `del`. Por ejemplo, `del modulo.la_respuesta` va a eliminar el atributo `the_answer` del objeto con nombre `modulo`.

Los espacios de nombres se crean en diferentes momentos y con diferentes tiempos de vida. El espacio de nombres que contiene los nombres incluidos se crea cuando se inicia el intérprete, y nunca se borra. El espacio de nombres global de un módulo se crea cuando se lee la definición de un módulo; normalmente, los espacios de nombres de módulos también duran hasta que el intérprete finaliza. Las instrucciones ejecutadas en el nivel de llamadas superior del intérprete, ya sea desde un script o interactivamente, se consideran parte del módulo llamado `__main__`, por lo tanto tienen su propio espacio de nombres global. (Los nombres incluídos en realidad también viven en un módulo; este se llama `__builtin__`.)

El espacio de nombres local a una función se crea cuando la función es llamada, y se elimina cuando la función retorna o lanza una excepción que no se maneje dentro de la función. (Podríamos decir que lo que pasa en realidad es que ese espacio de nombres se "olvida".) Por supuesto, las llamadas recursivas tienen cada una su propio espacio de nombres local.

Un *alcance* es una región textual de un programa en Python donde un espacio de nombres es accesible directamente. "Accesible directamente" significa que una referencia sin calificar a un nombre intenta encontrar dicho nombre dentro del espacio de nombres.

Aunque los alcances se determinan estáticamente, se usan dinámicamente. En cualquier momento durante la ejecución hay por lo menos tres alcances anidados cuyos espacios de nombres son directamente accesibles: el ámbito interno, donde se busca primero, contiene los nombres locales; los espacios de nombres de las funciones anexas, en las cuales se busca empezando por el alcance adjunto más cercano; el alcance intermedio, donde se busca a continuación, contiene el módulo de nombres globales actual; y el alcance exterior (donde se busca al final) es el espacio de nombres que contiene los nombres incluidos.

Si un nombre se declara como global, entonces todas las referencias y asignaciones al mismo van directo al alcance intermedio que contiene los nombres globales del módulo. De otra manera, todas las variables que se encuentren fuera del alcance interno son de sólo escritura (un intento de escribir a esas variables simplemente crea una *nueva* variable en el alcance interno, dejando intacta la variable externa del mismo nombre).

Habitualmente, el alcance local referencia los nombres locales de la función actual. Fuera de una función, el alcance local referncia al mismo espacio de nombres que el alcance global: el espacio de nombres del módulo. Las definiciones de clases crean un espacio de nombres más en el alcance local.

Es importante notar que los alcances se determinan textualmente: el alcance global de una función definida en un módulo es el espacio de nombres de ese módulo, no importa desde dónde o con qué alias se llame a la función. Por otro lado, la búsqueda de nombres se hace dinámicamente, en tiempo de ejecución --- sin embargo, la definición del lenguaje está evolucionando a hacer resolución de nombres estáticamente, en tiempo de "compilación", ¡así que no te confíes de la resolución de nombres dinámica! (De hecho, las variables locales ya se determinan estáticamente.)

Una peculiaridad especial de Python es que -- si no hay una declaración `global` o `nonlocal` en efecto -- las asignaciones a nombres siempre van al alcance interno. Las asignaciones no copian datos --- solamente asocian nombres a objetos. Lo mismo cuando se borra: la instrucción `del x` quita la asociación de x del espacio de nombres referenciado por el alcance local. De hecho, todas las operaciones que introducen nuevos nombres usan el alcance local: en particular, las instrucciones import y las definiciones de funciones asocian el módulo o nombre de la función al espacio de nombres en el alcance local. (La instrucción `global` puede usarse para indicar que ciertas variables viven en el alcance global.)

Un Primer Vistazo a las Clases

Las clases agregan un poco de sintaxis nueva, tres nuevos tipos de objetos y algo de semántica nueva.

Syntaxis de Definición de Clases

La forma más sencilla de definición de clase se ve así:

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

Las definiciones de clases, tal como las definiciones de funciones (sentencias `def`) deben ser ejecutadas antes de que tengan algún efecto. (Si quisieras podrías colocar una definición de clase en un bloque de una sentencia `if`, o dentro de una función.)

En la práctica, las sentencias dentro de una definición de clase generalmente serán definiciones de funciones, pero se permiten otras sentencias y a veces son útiles --- veremos esto más adelante. Las definiciones de funciones dentro de una clase normalmente tienen una forma peculiar de lista de argumentos, que está dada por las convenciones de llamadas para los métodos --- esto también será explicado luego.

Cuando se ingresa a una definición de clase, se crea un nuevo espacio de nombres, y es usado como el ámbito local --- por lo tanto, todas las asignaciones a variables locales quedan dentro de este nuevo espacio de nombres. En particular, las definiciones de funciones enlazan el nombre de la nueva función aquí.

Cuando se sale normalmente de la definición de una clase (por el final), se crea un *objeto clase*. Esto es básicamente un envoltorio de los contenidos del espacio de nombres creado por la definición de clase; veremos más sobre los objetos clase en la próxima sección. El ámbito local original (el que estaba en efecto justo antes de ingresar a la definición de clase) se restaura, y el objeto clase se enlaza aquí al nombre de la clase dado en el encabezado de la definición de clase (`NombreDeClase` en el ejemplo).

Objetos Clase

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

Para *referenciar atributos* se usa la sintaxis estandard que es usada para todas las referencias a atributos en Python: `objeto.nombre`. Nombres válidos de atributos son todos los nombres que estaban en el espacio de nombre de la clase en el momento que el

objeto clase fue creado. Por lo tanto, si la definición de clase fuera así:

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

entonces `MiClase.i` y `MiClase.f` son referencias de atributos válidas, que devuelven un entero y un objeto función, respectivamente. También se le puede asignar a los atributos de clase, asi que podés cambiar el valor de `MiClase.i` mediante la asignación. `__doc__` es también un atributo válido, que devuelve el docstring que pertenece a la clase: `"Una clase de ejemplo simple"`.

La *instanciación* de clase usa notación de función. Tan solo hacé de cuenta que el objeto clase es una función que no recibe parámetros, y que devuelve una nueva instancia de la clase. Por ejemplo (asumiendo la clase anterior):

```
x = MiClase()
```

crea un nueva *instancia* de la clase y asigna este objeto a la variable local `x`.

La operación de instanciación ("llamar" a un objeto clase) crea un objeto vacío. Muchas clases desean crear objetos con instancias personalizadas con un estado inicial específico. Por lo tanto la clase puede definir un método especial llamado `__init__()`, de esta forma:

```
def __init__(self):
    self.datos = []
```

Cuando una clase define un método `__init__()`, la instanciación de clase automáticamente invoca a `__init__()` para la instancia de clase recién creada. Entonces en este ejemplo, una instancia nueva e inicializada puede ser obtenida así:

```
x = MiClase()
```

Por supuesto, el método `__init__()` puede tener argumentos para mayor flexibilidad. En tal caso, los argumentos dados al operador de instanciación de clase son pasados a su vez a `__init__()`. Por ejemplo,

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

Objetos Instancia

Ahora, ¿Qué podemos hacer con los objetos instancia? La única operación que es entendida por los objetos instancia es la referencia de atributos. Hay dos tipos de nombres de atributos válidos, atributos de datos y métodos.

Los *atributos de datos* se corresponden con las "variables de instancia" en Smalltalk, y con las "variables miembro" en C++. Los atributos de datos no necesitan ser declarados; tal como las variables locales son creados la primera vez que se les asigna algo. Por ejemplo, si `x` es la instancia de `MiClase` creada más arriba, el siguiente pedazo de código va a imprimir el valor `16`, sin dejar ningún rastro:

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

El otro tipo de referenciar atributos de instancia es el *método*. Un método es una función que "pertenece a" un objeto. (En Python, el término método no está limitado a instancias de clase: otros tipos de objetos pueden tener métodos también. Por ejemplo, los objetos lista tienen métodos llamados append, insert, remove, sort, y así sucesivamente. Pero, en la siguiente explicación, usaremos el término método para referirnos exclusivamente a métodos de objetos instancia de clase, a menos que se especifique explícitamente lo contrario).

Los nombres válidos de métodos de un objeto instancia dependen de su clase. Por definición, todos los atributos de clase que son objetos funciones definen metodos correspondientes de sus instancias. Entonces, en nuestro ejemplo, `x.f` es una referencia a un método válido, dado que `MiClase.f` es una función, pero `x.i` no lo es, dado que `MiClase.i` no lo es. Pero `x.f` no es la misma cosa que `MiClase.f` --- es un *objeto método*, no un objeto función.

Objetos Método

Generalmente, un método es llamado luego de ser enlazado:

```
x.f()
```

En el ejemplo `MiClase`, esto devuelve la cadena `'hola mundo'`. Pero no es necesario llamar al método justo en ese momento: `x.f` es un objeto método, y puede ser guardado y llamado más tarde. Por ejemplo:

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

continuará imprimiendo `hola mundo` hasta el fín de los días.

¿Que sucede exactamente cuando un método es llamado? Debes haber notado que `x.f()` fue llamado más arriba sin ningún argumento, a pesar de que la definición de función de `f()` especificaba un argumento. ¿Que pasó con ese argumento? Por supuesto que Python levanta una excepción cuando una función que requiere un argumento es llamada sin ninguno --- aún si el argumento no es utilizado...

De hecho, tal vez hayas adivinado la respuesta: lo que tienen de especial los métodos es que el objeto es pasado como el primer argumento de la función. En nuestro ejemplo, la llamada `x.f()` es exáctamente equivalente a `MiClase.f(x)`. En general, llamar a un método con una lista de *n* argumentos es equivalente a llamar a la función correspondiente con una lista de argumentos que es creada insertando el objeto del método antes del primer argumento.

Si aún no comprendes como funcionan los métodos, un vistazo a la implementación puede ayudar a clarificar este tema. Cuando un atributo de instancia es referenciado y no es un atributo de datos, se busca dentro de su clase. Si el nombre denota un atributo de clase válido que es un objeto función, un método objeto es creado juntando (punteros a) el objeto instancia y el objeto función que ha sido encontrado. Este objeto abstracto creado de la junta es el objeto método. Cuando el objeto método es llamado con una lista de argumentos, es nuevamente desempacado, una lista de argumentos nueva es construida a partir del objeto instancia y la lista de argumentos original, y el objeto función es llamado con esta nueva lista de argumentos.

Random Remarks

Data attributes override method attributes with the same name; to avoid accidental name conflicts, which may cause hard-to-find bugs in large programs, it is wise to use some kind of convention that minimizes the chance of conflicts. Possible conventions include capitalizing method names, prefixing data attribute names with a small unique string (perhaps just an underscore), or using verbs for methods and nouns for data attributes.

Data attributes may be referenced by methods as well as by ordinary users ("clients") of an object. In other words, classes are not usable to implement pure abstract data types. In fact, nothing in Python makes it possible to enforce data hiding --- it is all based upon convention. (On the other hand, the Python implementation, written in C, can completely hide implementation details and control access to an object if necessary; this can be used by extensions to Python written in C.)

Clients should use data attributes with care --- clients may mess up invariants maintained by the methods by stamping on their data attributes. Note that clients may add data attributes of their own to an instance object without affecting the validity of the methods, as long as name conflicts are avoided --- again, a naming convention can save a lot of headaches here.

There is no shorthand for referencing data attributes (or other methods!) from within methods. I find that this actually increases the readability of methods: there is no chance of confusing local variables and instance variables when glancing through a method.

Often, the first argument of a method is called `self`. This is nothing more than a convention: the name `self` has absolutely no special meaning to Python. (Note, however, that by not following the convention your code may be less readable to other Python programmers, and it is also conceivable that a *class browser* program might be written that relies upon such a convention.)

Any function object that is a class attribute defines a method for instances of that class. It is not necessary that the function definition is textually enclosed in the class definition: assigning a function object to a local variable in the class is also ok. For example:

```
# Function defined outside the class
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1
    def g(self):
        return 'hello world'
    h = g
```

Now `f`, `g` and `h` are all attributes of class `C` that refer to function objects, and consequently they are all methods of instances of `C` --- `h` being exactly equivalent to `g`. Note that this practice usually only serves to confuse the reader of a program.

Methods may call other methods by using method attributes of the `self` argument:

```
class Bag:
    def __init__(self):
        self.data = []
    def add(self, x):
        self.data.append(x)
    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

Methods may reference global names in the same way as ordinary functions. The global scope associated with a method is the module containing the class definition. (The class itself is never used as a global scope!) While one rarely encounters a good reason for using global data in a method, there are many legitimate uses of the global scope: for one thing, functions and modules imported into the global scope can be used by methods, as well as functions and classes defined in it. Usually, the class containing the method is itself defined in this global scope, and in the next section we'll find some good reasons why a method would want to reference its own class!

Each value is an object, and therefore has a *class* (also called its *type*). It is stored as `object.__class__`.

Inheritance

Of course, a language feature would not be worthy of the name "class" without supporting inheritance. The syntax for a derived class definition looks like this:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

The name `BaseClassName` must be defined in a scope containing the derived class definition. In place of a base class name, other arbitrary expressions are also allowed. This can be useful, for example, when the base class is defined in another module:

```
class DerivedClassName(modname.BaseClassName):
```

Execution of a derived class definition proceeds the same as for a base class. When the class object is constructed, the base class is remembered. This is used for resolving attribute references: if a requested attribute is not found in the class, the search proceeds to look in the base class. This rule is applied recursively if the base class itself is derived from some other class.

There's nothing special about instantiation of derived classes: `DerivedClassName()` creates a new instance of the class. Method references are resolved as follows: the corresponding class attribute is searched, descending down the chain of base classes if necessary, and the method reference is valid if this yields a function object.

Derived classes may override methods of their base classes. Because methods have no special privileges when calling other methods of the same object, a method of a base class that calls another method defined in the same base class may end up calling a method of a derived class that overrides it. (For C++ programmers: all methods in Python are effectively `virtual`.)

An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name. There is a simple way to call the base class method directly: just call `BaseClassName.methodname(self, arguments)`. This is occasionally useful to clients as well. (Note that this only works if the base class is defined or imported directly in the global scope.)

Python has two builtin functions that work with inheritance:

- Use `isinstance()` to check an object's type: `isinstance(obj, int)` will be `True` only if `obj.__class__` is `int` or some class derived from `int`.
- Use `issubclass()` to check class inheritance: `issubclass(bool, int)` is `True` since `bool` is a subclass of `int`. However, `issubclass(unicode, str)` is `False` since `unicode` is not a subclass of `str` (they only share a common ancestor, `basestring`).

Multiple Inheritance

Python supports a limited form of multiple inheritance as well. A class definition with multiple base classes looks like this:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

For old-style classes, the only rule is depth-first, left-to-right. Thus, if an attribute is not found in `DerivedClassName`, it is searched in `Base1`, then (recursively) in the base classes of `Base1`, and only if it is not found there, it is searched in `Base2`, and so on.

(To some people breadth first --- searching `Base2` and `Base3` before the base classes of `Base1` --- looks more natural. However, this would require you to know whether a particular attribute of `Base1` is actually defined in `Base1` or in one of its base classes before you can figure out the consequences of a name conflict with an attribute of `Base2`. The depth-first rule makes no differences between direct and inherited attributes of `Base1`.)

For *new-style class*es, the method resolution order changes dynamically to support cooperative calls to `super()`. This approach is known in some other multiple-inheritance languages as call-next-method and is more powerful than the super call found in single-inheritance languages.

With new-style classes, dynamic ordering is necessary because all cases of multiple inheritance exhibit one or more diamond relationships (where one at least one of the parent classes can be accessed through multiple paths from the bottommost class). For example, all new-style classes inherit from `object`, so any case of multiple inheritance provides more than one path to reach `object`. To keep the base classes from being accessed more than once, the dynamic algorithm linearizes the search order in a way that preserves the left-to-right ordering specified in each class, that calls each parent only once, and that is monotonic (meaning that a class can be subclassed without affecting the precedence order of its parents). Taken together, these properties make it possible to design reliable and extensible classes with multiple inheritance. For more detail, see http://www.python.org/-download/releases/2.3/mro/.

Private Variables

There is limited support for class-private identifiers. Any identifier of the form `__spam` (at least two leading underscores, at most one trailing underscore) is textually replaced with `_classname__spam`, where `classname` is the current class name with leading underscore(s) stripped. This mangling is done without regard to the syntactic position of the identifier, so it can be used to define class-private instance and class variables, methods, variables stored in globals, and even variables stored in instances. private to this class on instances of *other* classes. Truncation may occur when the mangled name would be longer than 255 characters. Outside classes, or when the class name consists of only underscores, no mangling occurs.

Name mangling is intended to give classes an easy way to define "private" instance variables and methods, without having to worry about instance variables defined by derived classes, or mucking with instance variables by code outside the class. Note that the mangling rules are designed mostly to avoid accidents; it still is possible for a determined soul to access or modify a variable that is considered private. This can even be useful in special circumstances, such as in the debugger, and that's one reason why this loophole is not closed. (Buglet: derivation of a class with the same name as the base class makes use of private variables of the base class possible.)

Notice that code passed to `exec`, `eval()` or `execfile()` does not consider the classname of the invoking class to be the current class; this is similar to the effect of the `global` statement, the effect of which is likewise restricted to code that is byte-compiled together. The same restriction applies to `getattr()`, `setattr()` and `delattr()`, as well as when referencing `__dict__` directly.

Odds and Ends

Sometimes it is useful to have a data type similar to the Pascal "record" or C "struct", bundling together a few named data items. An empty class definition will do nicely:

```
class Employee:
    pass

john = Employee() # Create an empty employee record

# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

A piece of Python code that expects a particular abstract data type can often be passed a class that emulates the methods of that data type instead. For instance, if you have a

function that formats some data from a file object, you can define a class with methods `read()` and `readline()` that get the data from a string buffer instead, and pass it as an argument.

Instance method objects have attributes, too: `m.im_self` is the instance object with the method `m()`, and `m.im_func` is the function object corresponding to the method.

Exceptions Are Classes Too

User-defined exceptions are identified by classes as well. Using this mechanism it is possible to create extensible hierarchies of exceptions.

There are two new valid (semantic) forms for the raise statement:

```
raise Class, instance

raise instance
```

In the first form, `instance` must be an instance of `Class` or of a class derived from it. The second form is a shorthand for:

```
raise instance.__class__, instance
```

A class in an except clause is compatible with an exception if it is the same class or a base class thereof (but not the other way around --- an except clause listing a derived class is not compatible with a base class). For example, the following code will print B, C, D in that order:

```
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"
```

Note that if the except clauses were reversed (with `except B` first), it would have printed

B, B, B --- the first matching except clause is triggered.

When an error message is printed for an unhandled exception, the exception's class name is printed, then a colon and a space, and finally the instance converted to a string using the built-in function `str()`.

Iterators

By now you have probably noticed that most container objects can be looped over using a `for` statement:

```
for element in [1, 2, 3]:
    print element
for element in (1, 2, 3):
    print element
for key in {'one':1, 'two':2}:
    print key
for char in "123":
    print char
for line in open("myfile.txt"):
    print line
```

This style of access is clear, concise, and convenient. The use of iterators pervades and unifies Python. Behind the scenes, the `for` statement calls `iter()` on the container object. The function returns an iterator object that defines the method `next()` which accesses elements in the container one at a time. When there are no more elements, `next()` raises a `StopIteration` exception which tells the `for` loop to terminate. This example shows how it all works:

```
>>> 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 "<stdin>", line 1, in ?
    it.next()
StopIteration
```

Having seen the mechanics behind the iterator protocol, it is easy to add iterator behavior to your classes. Define a `__iter__()` method which returns an object with a `next()` method. If the class defines `next()`, then `__iter__()` can just return `self`:

```
class Reverse:
    "Iterator for looping over a sequence backwards"
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def next(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]

>>> for char in Reverse('spam'):
...     print char
...
m
a
p
s
```

Generators

Generators are a simple and powerful tool for creating iterators. They are written like regular functions but use the `yield` statement whenever they want to return data. Each time `next()` is called, the generator resumes where it left-off (it remembers all the data values and which statement was last executed). An example shows that generators can be trivially easy to create:

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]

>>> for char in reverse('golf'):
...     print char
...
f
l
o
```

```
g
```

Anything that can be done with generators can also be done with class based iterators as described in the previous section. What makes generators so compact is that the `__iter__()` and `next()` methods are created automatically.

Another key feature is that the local variables and execution state are automatically saved between calls. This made the function easier to write and much more clear than an approach using instance variables like `self.index` and `self.data`.

In addition to automatic method creation and saving program state, when generators terminate, they automatically raise `StopIteration`. In combination, these features make it easy to create iterators with no more effort than writing a regular function.

Expresiones Generadoras

Algunos generadores simples pueden ser codificados concisamente como expresiones usando una sintaxis similar a las listas por comprensión pero con paréntesis en vez de corchetes. Estas expresiones son designadas para situaciones donde el generador es usado inmediatamente por una función que lo contiene. Las expresiones generadoras son más compactas pero menos versátiles que definiciones completas de generadores, y tienden a utilizar menos memoria que las listas por comprensión equivalentes.

Ejemplos:

```
>>> sum(i*i for i in range(10))                 # suma de 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_de_senos = dict((x, sin(x*pi/180)) for x in range(0, 91))

>>> palabras_unicas = set(word  for line in page  for word in line.split())

>>> mejor_promedio = max((estudiante.promedio, estudiante.nombre) for estudiante in graduados)

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

Notas al Pie

5 Excepto una cosita. Los objetos módulo tienen un atributo secreto de solo lectura llamado `__dict__` que devuelve el diccionario usado para implementar el espacio de nombres del módulo; el nombre `__dict__` es un atributo, pero no es un nombre global. Obviamente, esto viola la abstracción de la implementación de espacios de nombres, y debe ser restringido a cosas tales como depuradores post-mortem.

Pequeño paseo por la Biblioteca Estándar

Interfaz al sistema operativo

El módulo `os` provee docenas de funciones para interactuar con el sistema operativo:

```
>>> import os
>>> os.system('time 0:02')
0
>>> os.getcwd()      # devuelve el directorio de trabajo actual
'C:\\Python26'
>>> os.chdir('/server/accesslogs')
```

Asegurate de usar el estilo `import os` en lugar de `from os import *`. Esto evitará que `os.open()` oculte a la función integrada `open()`, que trabaja bastante diferente.

Las funciones integradas `dir()` y `help()` son útiles como ayudas interactivas para trabajar con módulos grandes como `os`:

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

Para tareas diarias de administración de archivos y directorios, el módulo `shutil` provee una interfaz de más alto nivel que es más fácil de usar:

```
>>> import shutil
>>> shutil.copyfile('datos.db', 'archivo.db')
>>> shutil.move('/build/executables', 'dir_instalac')
```

Comodines de archivos

El módulo `glob` provee una función para hacer listas de archivos a partir de búsquedas con comodines en directorios:

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

Argumentos de linea de órdenes

Los programas frecuentemente necesitan procesar argumentos de linea de órdenes. Estos argumentos se almacenan en el atributo *argv* del módulo `sys` como una lista. Por ejemplo, la siguiente salida resulta de ejecutar `python demo.py uno dos tres` en la línea de órdenes:

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

El módulo `getopt` procesa *sys.argv* usando las convenciones de la función de Unix `getopt()`. El módulo `optparse` provee un procesamiento más flexible de la linea de órdenes.

Redirección de la salida de error y finalización del programa

El módulo `sys` también tiene atributos para *stdin*, *stdout*, y *stderr*. Este último es útil para emitir mensajes de alerta y error para que se vean incluso cuando se haya redireccionado *stdout*:

```
>>> sys.stderr.write('Alerta, archivo de log no encontrado\n')
Alerta, archivo de log no encontrado
```

La forma más directa de terminar un programa es usar `sys.exit()`.

Coincidencia en patrones de cadenas

El módulo `re` provee herramientas de expresiones regulares para un procesamiento avanzado de cadenas. Para manipulación y coincidencias complejas, las expresiones regulares ofrecen soluciones concisas y optimizadas:

```
>>> import re
>>> re.findall(r'\bt[a-z]*', 'tres felices tigres comen trigo')
['tres', 'tigres', 'trigo']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'gato en el el sombrero')
'gato en el sombrero'
```

Cuando se necesita algo más sencillo solamente, se prefieren los métodos de las cadenas porque son más fáciles de leer y depurar.

```
>>> 'te para tos'.replace('tos', 'dos')
'te para dos'
```

Matemática

El módulo `math` permite el acceso a las funciones de la biblioteca C subyacente para la matemática de punto flotante:

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

El módulo `random` provee herramientas para realizar selecciones al azar:

```
>>> import random
>>> random.choice(['manzana', 'pera', 'banana'])
'manzana'
>>> random.sample(xrange(100), 10)   # elección sin reemplazo
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # un float al azar
0.17970987693706186
>>> random.randrange(6)    # un entero al azar tomado de range(6)
4
```

Acceso a Internet

Hay varios módulos para acceder a internet y procesar sus protocolos. Dos de los más simples son `urllib2` para traer data de URLs y `smtplib` para mandar correos:

```
>>> import urllib2
>>> for line in urllib2.urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl'):
...     if 'EST' in line or 'EDT' in line:  # buscamos la hora del este
...         print line

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

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@ejemplo.org', 'jcaesar@ejemplo.org',
... """To: jcaesar@ejemplo.org
... From: soothsayer@ejemplo.org
...
... Ojo al piojo.
... """)
>>> server.quit()
```

(Notá que el segundo ejemplo necesita un servidor de correo corriendo en la máquina local)

---------- revisado hasta acá!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! .. _tut-dates-and-times:

Fechas y tiempos

El módulo `datetime` ofrece clases para manejar fechas y tiempos tanto de manera simple como compleja. Aunque se soporta aritmética sobre fechas y tiempos, el foco de la implementación es en la eficiente extracción de partes para manejarlas o formatear la salida. El módulo también soporta objetos que son conscientes de la zona horaria.

```
# las fechas son fácilmente construidas y formateadas
>>> from datetime import date
>>> hoy = date.today()
>>> hoy
datetime.date(2009, 7, 19)

# nos aseguramos de tener la info de localización correcta
>>> locale.setlocale(locale.LC_ALL, locale.getdefaultlocale())
'es_ES.UTF8'
>>> hoy.strftime("%m-%d-%y. %d %b %Y es %A. hoy es %d de %B.")
'07-19-09. 19 jul 2009 es domingo. hoy es 19 de julio.'

# las fechas soportan aritmética de calendario
>>> nacimiento = date(1964, 7, 31)
>>> edad = hoy - nacimiento
>>> edad.days
14368
```

Compresión de datos

Los formatos para archivar y comprimir datos se soportan directamente con los módulos: `zlib`, `gzip`, `bz2`, `zipfile` y `tarfile`.

```
>>> import zlib
>>> s = 'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

Medición de rendimiento

Algunos usuarios de Python desarrollan un profundo interés en saber el rendimiento relativo de las diferentes soluciones al mismo problema. Python provee una herramienta de medición que responde esas preguntas inmediatamente.

Por ejemplo, puede ser tentador usar la característica de empaquetamiento y desempaquetamiento de las tuplas en lugar de la solución tradicional para intercambiar argumentos. El módulo `timeit` muestra rapidamente una modesta 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
```

En contraste con el fino nivel de granularidad del módulo `timeit`, los módulos `profile` y `pstats` proveen herramientas para identificar secciones críticas de tiempo en bloques de código más grandes.

Control de calidad

Una forma para desarrollar software de alta calidad es escribir pruebas para cada función mientras se la desarrolla, y correr esas pruebas frecuentemente durante el proceso de desarrollo.

El módulo `doctest` provee una herramienta para revisar un módulo y validar las pruebas integradas en las cadenas de documentación (o *docstring*) del programa. La construcción de las pruebas es tan sencillo como cortar y pegar una ejecución típica junto con sus resultados en los docstrings. Esto mejora la documentación al proveer al usuario un ejemplo y permite que el módulo `doctest` se asegure que el código permanece fiel a la documentación:

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

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

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

El módulo `unittest` necesita más esfuerzo que el módulo `doctest`, pero permite que

se mantenga en un archivo separado un conjunto más comprensivo de pruebas:

```
import unittest

class TestFuncionesEstadisticas(unittest.TestCase):

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

unittest.main() # llamarlo de la linea de comandos ejecuta todas las pruebas
```

Las pilas incluidas

Python tiene una filosofía de "pilas incluidas". Esto se ve mejor en las capacidades robustas y sofisticadas de sus paquetes más grandes. Por ejemplo:

- Los módulos `xmlrpclib` y `SimpleXMLRPCServer` hacen que implementar llamadas a procedimientos remotos sea una tarea trivial. A pesar de los nombres de los módulos, no se necesita conocimiento directo o manejo de XML.
- El paquete `email` es una biblioteca para manejar mensajes de mail, incluyendo MIME y otros mensajes basados en RFC 2822. Al contrario de `smtplib` y `poplib` que en realidad envían y reciben mensajes, el paquete `email` tiene un conjunto de herramientas completo para construir y decodificar estructuras complejas de mensajes (incluyendo adjuntos) y para implementar protocolos de cabecera y codificación de Internet).
- Los paquetes `xml.dom` y `xml.sax` proveen un robusto soporte para analizar este popular formato de intercambio de datos. Asimismo, el módulo `csv` soporta lecturas y escrituras directas en un formato común de base de datos. Juntos, estos módulos y paquetes simplifican enormemente el intercambio de datos entre aplicaciones Python y otras herramientas.
- Se soporta la internacionalización a través de varios módulos, incluyendo `gettext`, `locale`, y el paquete `codecs`.

■.. _tut-brieftourtwo:

Pequeño paseo por la Biblioteca Estándar - Parte II

Este segundo paseo cubre módulos más avanzados que facilitan necesidades de programación avanzadas. Estos módulos raramente se usan en scripts cortos.

Formato de salida

El módulo `repr` provee una versión de `repr()` ajustada para mostrar contenedores grandes o profundamente anidados, en forma abreviada:

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

El módulo `pprint` ofrece un control más sofisticado de la forma en que se imprimen tanto los objetos predefinidos como los objetos definidos por el usuario, de manera que sean legibles por el intérprete. Cuando el resultado ocupa más de una línea, el generador de "impresiones lindas" agrega saltos de línea e indentación para mostrar la estructura de los datos más claramente:

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

El módulo `textwrap` formatea párrafos de texto para que quepan dentro de cierto ancho de pantalla:

```
>>> import textwrap
>>> doc = """El método wrap() es como fill(), excepto que devuelve
... una lista de strings en lugar de una gran string con saltos de
... línea como separadores."""
>>> print textwrap.fill(doc, width=40)
El método wrap() es como fill(), excepto
que devuelve una lista de strings en
lugar de una gran string con saltos de
línea como separadores.
```

El módulo `locale` accede a una base de datos de formatos específicos de una cultura. El

atributo :var:`grouping` de la función :function:`format` permite una forma directa de formatear números con separadores de grupo:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, '')
'Spanish_Argentina.1252'
>>> conv = locale.localeconv()      # obtener un mapeo de convenciones
>>> x = 1234567.8
>>> locale.format("%d", x, grouping=True)
'1.234.567'
>>> locale.format("%s%.*f", (conv['currency_symbol'],
...         conv['frac_digits'], x), grouping=True)
'$1.234.567,80'
```

Plantillas

El módulo `string` incluye una clase versátil `Template` (plantilla) on una sintaxis simplificada apta para ser editada por usuarios finales. Esto permite que los usuarios personalicen sus aplicaciones sin necesidad de modificar la aplicación en sí.

El formato usa marcadores cuyos nombres se forman con $ seguido de identificadores Python válidos (caracteres alfanuméricos y guión de subrayado). Si se los encierra entre llaves, pueden seguir más caracteres alfanuméricos sin necesidad de dejar espacios en blanco. $$ genera un $:

```
>>> from string import Template
>>> t = Template('${village}folk send $$10 to $cause.')
>>> t.substitute(village='Nottingham', cause='the ditch fund')
'Nottinghamfolk send $10 to the ditch fund.'
```

El método `substitute()` lanza `KeyError` cuando no se suministra ningún valor para un marcador mediante un diccionario o argumento por nombre. Para aplicaciones estilo "combinación de correo", los datos suministrados por el usuario puede ser incompletos y el método `safe_substitute()` puede ser más apropiado --- deja los marcadores inalterados cuando hay datos faltantes:

```
>>> t = Template('Return the $item to $owner.')
>>> d = dict(item='unladen swallow')
>>> t.substitute(d)
Traceback (most recent call last):
  . . .
KeyError: 'owner'
>>> t.safe_substitute(d)
'Return the unladen swallow to $owner.'
```

Las subclases de Template pueden especificar un delimitador propio. Por ejemplo, una

utilidad de renombrado por lotes para un visualizador de fotos puede escoger usar signos de porcentaje para los marcadores tales como la fecha actual, el número de secuencia de la imagen, o el formato de archivo:

```
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Template):
...     delimiter = '%'
...
>>> fmt = raw_input('Enter rename style (%d-date %n-seqnum %f-format):  ')
Enter rename style (%d-date %n-seqnum %f-format):  Ashley_%n%f
>>> t = BatchRename(fmt)
>>> date = time.strftime('%d%b%y')
>>> for i, filename in enumerate(photofiles):
...     base, ext = os.path.splitext(filename)
...     newname = t.substitute(d=date, n=i, f=ext)
...     print '{0} --> {1}'.format(filename, newname)
...
img_1074.jpg --> Ashley_0.jpg
img_1076.jpg --> Ashley_1.jpg
img_1077.jpg --> Ashley_2.jpg
```

Las plantillas también pueden ser usadas para separar la lógica del programa de los detalles de múltiples formatos de salida. Esto permite sustituir plantillas específicas para archivos XML, reportes en texto plano, y reportes web en HTML.

Trabajar con registros estructurados conteniendo datos binarios

El módulo `struct` provee las funciones `pack()` y `unpack()` para trabajar con formatos de registros binarios de longitud variable. El siguiente ejemplo muestra cómo recorrer la información de encabezado en un archivo ZIP sin usar el módulo `zipfile`. Los códigos `"H"` e `"I"` representan números sin signo de dos y cuatro bytes respectivamente. El `"<"` indica que son de tamaño estándar y los bytes tienen ordenamiento *little-endian*:

```
import struct

datos = open('miarchivo.zip', 'rb').read()
inicio = 0
for i in range(3):                      # mostrar los 3 primeros encabezados
    inicio += 14
    campos = struct.unpack('<IIIHH', datos[inicio:inicio+16])
    crc32, tam_comp, tam_descomp, tam_nomarch, tam_extra = fields

    inicio += 16
    nomarch = datos[inicio:inicio+tam_nomarch]
```

```
    inicio += tam_nomarch
    extra = datos[inicio:inicio+tam_extra]
    print nomarch, hex(crc32), tam_comp, tam_descomp

    inicio += tam_extra + tam_comp     # saltear hasta el próximo encabezado
```

Multihilo

La técnica de multihilos permite desacoplar tareas que no tienen dependencia secuencial. Los hilos se pueden usar para mejorar el grado de reacción de las aplicaciones que aceptan entradas del usuario mientras otras tareas se ejecutan en segundo plano. Un caso de uso relacionado es ejecutar E/S en paralelo con cálculos en otro hilo.

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

```
import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, arch_ent, arch_sal):
        threading.Thread.__init__(self)
        self.arch_ent = arch_ent
        self.arch_sal = arch_sal
    def run(self):
        f = zipfile.ZipFile(self.arch_sal, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.arch_ent)
        f.close()
        print 'Terminó zip en segundo plano de: ', self.arch_ent

seg_plano = AsyncZip('misdatos.txt', 'miarchivo.zip')
seg_plano.start()
print 'El programa principal continúa la ejecución en primer plano.'

seg_plano.join()    # esperar que termine la tarea en segundo plano
print 'El programa principal esperó hasta que el segundo plano estuviera \
       listo.'
```

El desafío principal de las aplicaciones multihilo es la coordinación entre los hilos que comparten datos u otros recursos. A ese fin, el módulo threading provee una serie de primitivas de sincronización que incluyen locks, eventos, variables de condición, y semáforos.

Aún cuando esas herramientas son poderosas, pequeños errores de diseño pueden resultar en problemas difíciles de reproducir. La forma preferida de coordinar tareas es concentrar todos los accesos a un recurso en un único hilo y después usar el módulo `Queue` para alimentar dicho hilo con pedidos desde otros hilos. Las aplicaciones que usan objetos `Queue.Queue` para comunicación y coordinación entre hilos son más fáciles de

diseñar, más legibles, y más confiables.

Registro

El módulo `logging` ofrece un sistema de registro (traza) completo y flexible. En su forma más simple, los mensajes de registro se envían a un archivo o a `sys.stderr`:

```
import logging
logging.debug('Información de depuración')
logging.info('Mensaje informativo')
logging.warning('Atención: archivo de configuración %s no se encuentra',
                'server.conf')
logging.error('Ocurrió un error')
logging.critical('Error crítico -- cerrando')
```

Ésta es la salida obtenida:

```
WARNING:root:Atención: archivo de configuración server.conf no se encuentra
ERROR:root:Ocurrió un error
CRITICAL:root:Error crítico -- cerrando
```

De forma predeterminada, los mensajes de depuración e informativos se suprimen, y la salida se envía al error estándar. Otras opciones de salida incluyen mensajes de ruteo a través de correo electrónico, datagramas, sockets, o un servidor HTTP. Nuevos filtros pueden seleccionar diferentes rutas basadas en la prioridad del mensaje: `DEBUG`, `INFO`, `WARNING`, `ERROR`, and `CRITICAL` (Depuración, Informativo, Atención, Error y Crítico respectivamente)

El sistema de registro puede configurarse directamente desde Python o puede cargarse la configuración desde un archivo editable por el usuario para personalizar el registro sin alterar la aplicación.

Referencias débiles

Python realiza administración de memoria automática (cuenta de referencias para la mayoría de los objetos, y *garbage collection* (recolección de basura) para eliminar ciclos). La memoria se libera poco después de que la última referencia a la misma haya sido eliminada.

Esta estrategia funciona bien para la mayoría de las aplicaciones, pero ocasionalmente aparece la necesidad de hacer un seguimiento de objetos sólo mientras están siendo usados por alguien más. Desafortunadamente, el sólo hecho de seguirlos crea una referencia que los hace permanentes.

El módulo `weakref` provee herramientas para seguimiento de objetos que no crean una referencia. Cuando el objeto no se necesita más, es eliminado automáticamente de una tabla de referencias débiles y se dispara una retrollamada (*callback*). Comúnmente se usa

para mantener una *cache* de objetos que son caros de crear:

```
>>> import weakref, gc
>>> class A:
...     def __init__(self, value):
...             self.value = value
...     def __repr__(self):
...             return str(self.value)
...
>>> a = A(10)                    # crear una referencia
>>> d = weakref.WeakValueDictionary()
>>> d['primaria'] = a            # no crea una referencia
>>> d['primaria']                # traer el objeto si aún está vivo
10
>>> del a                        # eliminar la única referencia
>>> gc.collect()                 # recolección de basura justo ahora
0
>>> d['primaria']                # la entrada fue automáticamente eliminada
Traceback (most recent call last):
  . . .
KeyError: 'primaria'
```

Herramientas para trabajar con listas

Muchas necesidades de estructuras de datos pueden ser satisfechas con el tipo lista integrado. Sin embargo, a veces se hacen necesarias implementaciones alternativas con rendimientos distintos.

El módulo `array` provee un objeto `array()` (vector) que es como una lista que almacena sólo datos homogéneos y de una manera más compacta. Los ejemplos a continuación muestran un vector de números guardados como dos números binarios sin signo de dos bytes (código de tipo `"H"`) en lugar de los 16 bytes por elemento habituales en listas de objetos int 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` provee un objeto `deque()` que es como una lista más rápida para agregar y quitar elementos por el lado izquierdo pero búsquedas más lentas por el medio. Estos objetos son adecuados para implementar colas y árboles de búsqueda a lo ancho:

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

no_visitado = deque([nodo_inicial])
def busqueda_a_lo_ancho(no_visitado):
    nodo = no_visitado.popleft()
    for m in gen_moves(nodo):
        if is_goal(m):
            return m
        no_visitado.append(m)
```

Además de las implementaciones alternativas de listas, la biblioteca ofrece otras herramientas como el módulo `bisect` con funciones para manipular listas ordenadas:

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

El módulo `heapq` provee funciones para implementar heaps basados en listas comunes. El menor valor ingresado se mantiene en la posición cero. Esto es útil para aplicaciones que acceden a menudo al elemento más chico pero no quieren hacer un orden completo de la lista:

```
>>> from heapq import heapify, heappop, heappush
>>> datos = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(datos)                      # acomodamos la lista a orden de heap
>>> heappush(datos, -5)                 # agregamos un elemento
>>> [heappop(datos) for i in range(3)] # traemos los tres elementos menores
[-5, 0, 1]
```

Aritmética de punto flotante decimal

El módulo `decimal` provee un tipo de dato `Decimal` para soportar aritmética de punto flotante decimal. Comparado con `float`, la implementación de punto flotante binario incluida, la nueva clase es muy útil especialmente para aplicaciones financieras y para cualquier uso que requiera una representación decimal exacta, control de la precisión, control del redondeo para satisfacer requerimientos legales o reglamentarios, seguimiento de cifras significativas, o para aplicaciones donde el usuario espera que los resultados coincidan con cálculos hechos a mano.

Por ejemplo, calcular un impuesto del 5% de una tarifa telefónica de 70 centavos da resultados distintos con punto flotante decimal y punto flotante binario. La diferencia se vuelve significativa si los resultados se redondean al centavo más próximo:

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

El resultado con `Decimal` conserva un cero al final, calculando automáticamente cuatro cifras significativas a partir de los multiplicandos con dos cifras significativas. Decimal reproduce la matemática como se la hace a mano, y evita problemas que pueden surgir cuando el punto flotante binario no puede representar exactamente cantidades decimales.

La representación exacta permite a la clase `Decimal` hacer cálculos de modulo y pruebas de igualdad que son inadecuadas para punto flotante binario:

```
>>> 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` provee aritmética con tanta precisión como haga falta:

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

¿Y ahora qué?

Leer este tutorial probablemente reforzó tu interés por usar Python, deberías estar ansioso por aplicar Python a la resolución de tus problemas reales. ¿A dónde deberías ir para aprender más?

Este tutorial forma parte del juego de documentación de Python. Algunos otros documentos que encontrarás en este juego son:

- *library-index*:

  Deberías hojear este manual, que tiene material de referencia completo (si bien breve) sobre tipos, funciones y módulos de la biblioteca estándar. La distribución de Python estándar incluye un *montón* de código adicional. Hay módulos para leer archivos de correo de Unix, obtener documentos vía HTTP, generar números aleatorios, interpretar opciones de línea de comandos, escribir programas CGI, comprimir datos, y muchas otras tareas. Un vistazo por la Referencia de Biblioteca te dará una idea de lo que hay disponible.
- *install-index* explica cómo instalar módulos externos escritos por otros usuarios de Python.
- *reference-index*: Una descripción en detalle de la sintaxis y semántica de Python. Es una lectura pesada, pero útil como guía completa al lenguaje en si.

Más recursos sobre Python:

- http://www.python.org: El sitio web principal sobre Python. Contiene código, documentación, y referencias a páginas relacionadas con Python en la Web. Este sitio web tiene copias espejo en varios lugares del mundo cómo Europa, Japón y Australia; una copia espejo puede funcionar más rápido que el sitio principal, dependiendo de tu ubicación geográfica.
- http://docs.python.org: Acceso rápido a la documentación de Python.
- http://pypi.python.org: El Índice de Paquetes de Python, antes también apodado "El Negocio de Quesos", es un listado de módulos de Python disponibles para descargar hechos por otros usuarios. Cuándo comiences a publicar código, puedes registrarlo aquí así los demás pueden encontrarlo.
- http://aspn.activestate.com/ASPN/Python/Cookbook/: El Recetario de Python es una colección de tamaño considerable de ejemplos de código, módulos más grandes, y programas útiles. Las contribuciones particularmente notorias están recolectadas en un libro también titulado Recetario de Python (O'Reilly & Associates, ISBN 0-596-00797-3.)

Para preguntas relacionadas con Python y reportes de problemas puedes escribir al grupo de noticias *comp.lang.python*, o enviarlas a la lista de correo que hay en python-list@python.org. El grupo de noticias y la lista de correo están interconectadas, por lo que los mensajes enviados a uno serán retransmitidos al otro. Hay alrededor de 120 mensajes diarios (con picos de hasta varios cientos), haciendo (y respondiendo) preguntas, sugiriendo nuevas características, y anunciando nuevos módulos. Antes de escribir, asegúrate

de haber revisado la lista de Preguntas Frecuentes (también llamado el FAQ), o buscalo en el directorio `Misc/` de la distribución de código fuente de Python. Hay archivos de la lista de correo disponibles en http://mail.python.org/pipermail/. El FAQ responde a muchas de las preguntas que aparecen una y otra vez, y puede que ya contenga la solución a tu problema.

Edición de Entrada Interactiva y Sustitución de Historial

Algunas versiones del intérprete de Python permiten editar la línea de entrada actual, y sustituir en base al historial, de forma similar a las capacidades del intérprete de comandos Korn y el GNU bash. Esto se implementa con la biblioteca *GNU Readline*, que soporta edición al estilo de Emacs y al estilo de vi. Esta biblioteca tiene su propia documentación que no duplicaré aquí; pero la funcionalidad básica es fácil de explicar. La edición interactiva y el historial aquí descriptos están disponibles como opcionales en las versiones para Unix y Cygwin del intérprete.

Este capítulo *no* documenta las capacidades de edición del paquete PythonWin de Mark Hammond, ni del entorno IDLE basado en Tk que se distribuye con Python. El historial de línea de comandos que funciona en pantallas de DOS en NT y algunas otras variantes de DOS y Windows es también una criatura diferente.

Edición de Línea

De estar soportada, la edición de línea de entrada se activa en cuanto el intérprete muestra un símbolo de espera de ordenes primario o secundario. La línea activa puede editarse usando los caracteres de control convencionales de Emacs. De estos, los más importantes son: `C-A` (Ctrl-A) mueve el cursor al comienzo de la línea, `C-E` al final, `C-B` lo mueve una posición a la izquierda, `C-F` a la derecha. La tecla de retroceso (Backspace) borra el caracter a la izquierda del cursor, `C-D` el caracter a su derecha. `C-K` corta el resto de la línea a la derecha del cursor, `C-Y` pega de vuelta la última cadena cortada. `C-underscore` deshace el último cambio hecho; puede repetirse para obtener un efecto acumulativo.

Sustitución de historial

La sustitución de historial funciona de la siguiente manera: todas las líneas ingresadas y no vacías se almacenan en una memoria intermedia, y cuando se te pide una nueva línea, estás posicionado en una linea nueva al final de esta memoria. `C-P` se mueve una línea hacia arriba (es decir, hacia atrás) en el historial, `C-N` se mueve una línea hacia abajo. Cualquier línea en el historial puede editarse; aparecerá un asterisco adelante del indicador de entrada para marcar una línea como editada. Presionando la tecla `Return` (Intro) se pasa la línea activa al intérprete. `C-R` inicia una búsqueda incremental hacia atrás, `C-S` inicia una búsqueda hacia adelante.

Atajos de teclado

Los atajos de teclado y algunos otros parámetros de la biblioteca Readline se pueden personalizar poniendo comandos en un archivo de inicialización llamado `~/.inputrc`. Los atajos de teclado tienen la forma

```
nombre-de-tecla: nombre-de-función
```

o

```
"cadena": nombre-de-función
```

y se pueden configurar opciones con

```
set nombre-opción valor
```

Por ejemplo:

```
# Prefiero edición al estilo vi:
set editing-mode vi

# Editar usando sólo un renglón:
set horizontal-scroll-mode On

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

Observa que la asociación por omisión para la tecla `Tab` en Python es insertar un caracter `Tab` (tabulación horizontal) en vez de la función por defecto de Readline de completar nombres de archivo. Si insistes, puedes redefinir esto poniendo

```
Tab: complete
```

en tu `~/.inputrc`. (Desde luego, esto hace más difícil escribir líneas de continuación indentadas si estás acostumbrado a usar `Tab` para tal propósito.)

Hay disponible opcionalmente completado automático de variables y nombres de módulos. Para activarlo en el modo interactivo del intérprete, agrega lo siguiente a tu archivo de arranque: [6]

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

Esto asocia la tecla `Tab` a la función de completado, con lo cual presionar la tecla `Tab` dos veces sugerirá valores para completar; se fija en nombres de instrucciones Python, las variables locales del momento, y los nombres de módulos disponibles. Para expresiones con puntos como `string.a`, evaluará la expresión hasta el último `'.'` y luego sugerirá opciones a completar de los atributos de el objeto resultante. Tenga en cuenta que esto puede ejecutar código definido por la aplicación si un objeto con un método

`__getattr__()` forma parte de la expresión.

Un archivo de inicialización con más capacidades podría ser como este ejemplo. Observa que éste borra los nombres que crea una vez que no se necesitan más; esto se hace debido a que el archivo de inicialización se ejecuta en el mismo espacio de nombres que los comandos interactivos, y borrar los nombres evita que se produzcan efectos colaterales en el entorno interactivo. Tal vez te resulte cómodo mantener algunos de los módulos importados, tales como `os`, que usualmente acaban siendo necesarios en la mayoría de las sesiones con el intérprete.

```
# Añadir auto-completado y almacenamiento de archivo de histórico a tu
# intérprete de Python interactivo. Requiere Python 2.0+, y readline.
# El autocompletado esta ligado a la tecla Esc por defecto (puedes
# modificarlo - lee la documentación de readline).
#
# Guarda este archivo en ~/.pystartup, y configura una variable de inicio
# para que lo apunte: en bash "export PYTHONSTARTUP=/home/usuario/.pystartup".
#
# Ten en cuenta que PYTHONSTARTUP *no* expande "~", así que debes poner
# la ruta completa a tu directorio personal.

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
```

Comentario

Esta funcionalidad es un paso enorme hacia adelante comparado con versiones anteriores del interprete; de todos modos, quedan pendientes algunos deseos: sería bueno si la indentación correcta se sugiriera en las lineas de continuación (el parser sabe si se requiere una indentación a continuación). El mecanismo de completado podría usar la tabla de símbolos del intérprete. Un comando para verificar (o incluso sugerir) coincidencia de paréntesis, comillas, etc. también sería útil.

Notas

6	Python ejecutará el contenido de un archivo indicado por la variable de entorno **PYTHONSTARTUP** cuando inicies un intérprete interactivo.

Aritmética de Punto Flotante: Problemas y Limitaciones

Los números de punto flotante se representan en el hardware de la computadora en fracciones en base 2 (binario). Por ejemplo, la fracción decimal

```
0.125
```

...tiene el valor 1/10 + 2/100 + 5/1000, y de la misma manera la fracción binaria

```
0.001
```

...tiene el valor 0/2 + 0/4 + 1/8. Estas dos fracciones tienen valores idénticos, la única diferencia real es que la primera está escrita en notación fraccional en base 10 y la segunda en base 2.

Desafortunadamente, la mayoría de las fracciones decimales no pueden representarse exactamente como fracciones binarias. Como consecuencia, en general los números de punto flotante decimal que ingresás en la computadora son sólo aproximados por los números de punto flotante binario que realmente se guardan en la máquina.

El problema es más fácil de entender primero en base 10. Considerá la fracción 1/3. Podés aproximarla como una fracción de base 10

```
0.3
```

...o, mejor,

```
0.33
```

...o, mejor,

```
0.333
```

...y así. No importa cuantos dígitos desees escribir, el resultado nunca será exactamente 1/3, pero será una aproximación cada vez mejor de 1/3.

De la misma manera, no importa cuantos dígitos en base 2 quieras usar, el valor decimal 0.1 no puede representarse exactamente como una fracción en base 2. En base 2, 1/10 es la siguiente fracción que se repite infinitamente:

```
0.0001100110011001100110011001100110011001100110011...
```

Frená en cualquier número finito de bits, y tendrás una aproximación. Es por esto que ves cosas como:

```
>>> 0.1
0.10000000000000001
```

En la mayoría de las máquinas de hoy en día, eso es lo que verás si ingresás 0.1 en un prompt de Python. Quizás no, sin embargo, porque la cantidad de bits usados por el hardware para almacenar valores de punto flotante puede variar en las distintas máquinas, y Python sólo muestra una aproximación del valor decimal verdadero de la aproximación binaria guardada por la máquina. En la mayoría de las máquinas, si Python fuera a mostrar el verdadero valor decimal de la aproximación almacenada por 0.1, tendría que mostrar sin embargo

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

El prompt de Python usa la función integrada `repr()` para obtener una versión en cadena de caracteres de todo lo que muestra. Para flotantes, `repr(float)` redondea el valor decimal verdadero a 17 dígitos significativos, dando

```
0.10000000000000001
```

`repr(float)` produce 17 dígitos significativos porque esto es suficiente (en la mayoría de las máquinas) para que se cumpla `eval(repr(x)) == x` exactamente para todos los flotantes finitos *X*, pero redondeando a 16 dígitos no es suficiente para que sea verdadero.

Notá que esta es la verdadera naturaleza del punto flotante binario: no es un error de Python, y tampoco es un error en tu código. Verás lo mismo en todos los lenguajes que soportan la aritmética de punto flotante de tu hardware (a pesar de que en algunos lenguajes por omisión no *muestren* la diferencia, o no lo hagan en todos los modos de salida).

La función integrada :func: *str* de Python produce sólo 12 dígitos significativos, y quizás quieras usar esa. Normalmente `eval(str(x))` no reproducirá *x*, pero la salida quizás sea más placentera de ver:

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

Es importante darse cuenta de que esto es, realmente, una ilusión: el valor en la máquina no es exactamente 1/10, simplemente estás redondeando el valor que se *muestra* del valor verdadero de la máquina.

A esta se siguen otras sorpresas. Por ejemplo, luego de ver:

```
>>> 0.1
0.10000000000000001
```

...quizás estés tentado de usar la función `round()` para recortar el resultado al dígito que esperabas. Pero es lo mismo:

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

El problema es que el valor de punto flotante binario almacenado para "0.1" ya era la mejor aproximación binaria posible de 1/10, de manera que intentar redondearla nuevamente no puede mejorarla: ya era la mejor posible.

Otra consecuencia es que como 0.1 no es exactamente 1/10, sumar diez valores de 0.1 quizás tampoco dé exactamente 1.0:

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

La aritmética de punto flotante binaria tiene varias sorpresas como esta. El problema con "0.1" es explicado con detalle abajo, en la sección "Error de Representación". Mirá los Peligros del Punto Flotante (en inglés, The Perils of Floating Point) para una más completa recopilación de otras sorpresas normales.

Como dice cerca del final, "no hay respuestas fáciles". A pesar de eso, ¡no le tengas mucho miedo al punto flotante! Los errores en las operaciones flotantes de Python se heredan del hardware de punto flotante, y en la mayoría de las máquinas están en el orden de no más de una 1 parte en 2**53 por operación. Eso es más que adecuado para la mayoría de las tareas, pero necesitás tener en cuenta que no es aritmética decimal, y que cada operación de punto flotante sufre un nuevo error de redondeo.

A pesar de que existen casos patológicos, para la mayoría de usos casuales de la aritmética de punto flotante al final verás el resultado que esperás si simplemente redondeás lo que mostrás de tus resultados finales al número de dígitos decimales que esperás. `str()` es normalmente suficiente, y para un control más fino mirá los parámetros del método de formateo `str.format()` en *formatstrings*.

Error de Representación

Esta sección explica el ejemplo "0.1" en detalle, y muestra como en la mayoría de los casos vos mismo podés realizar un análisis exacto como este. Se asume un conocimiento básico de la representación de punto flotante binario.

Error de representación se refiere al hecho de que algunas (la mayoría) de las fracciones decimales no pueden representarse exactamente como fracciones binarias (en base 2). Esta es la razón principal de por qué Python (o Perl, C, C++, Java, Fortran, y tantos otros) frecuentemente no mostrarán el número decimal exacto que esperás:

```
>>> 0.1
0.10000000000000001
```

¿Por qué es eso? 1/10 no es representable exactamente como una fracción binaria. Casi

todas las máquinas de hoy en día (Noviembre del 2000) usan aritmética de punto flotante IEEE-754, y casi todas las plataformas mapean los flotantes de Python al "doble precisión" de IEEE-754. Estos "dobles" tienen 53 bits de precisión, por lo tanto en la entrada la computadora intenta convertir 0.1 a la fracción más cercana que puede de la forma *J*/2***N* donde *J* es un entero que contiene exactamente 53 bits. Reescribiendo

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

...como

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

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

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

O sea, 56 es el único valor para *N* que deja *J* con exactamente 53 bits. El mejor valor posible para *J* es entonces el cociente redondeado:

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

Ya que el resto es más que la mitad de 10, la mejor aproximación se obtiene redondeándolo:

```
>>> q+1
7205759403792794L
```

Por lo tanto la mejor aproximación a 1/10 en doble precisión 754 es eso sobre 2**56, o

```
7205759403792794 / 72057594037927936
```

Notá que como lo redondeamos, esto es un poquito más grande que 1/10; si no lo hubiéramos redondeado, el cociente hubiese sido un poquito menor que 1/10. ¡Pero no hay caso en que sea *exactamente* 1/10!

Entonces la computadora nunca "ve" 1/10: lo que ve es la fracción exacta de arriba, la mejor aproximación al flotante doble de 754 que puede obtener:

```
>>> .1 * 2**56
7205759403792794.0
```

Si multiplicamos esa fracción por 10**30, podemos ver el valor (truncado) de sus 30 dígitos

más significativos:

```
>>> 7205759403792794 * 10**30 / 2**56
100000000000000005551115123125L
```

...lo que significa que el valor exacto almacenado en la computadora es aproximadamente igual al valor decimal 0.100000000000000005551115123125. Redondeando eso a 17 dígitos significativos da el 0.10000000000000001 que Python muestra (bueno, mostraría en cualquier plataforma que cumpla con 754 cuya biblioteca en C haga la mejor conversión posible en entrada y salida... ¡la tuya quizás no!).

Contenido

Índice

Symbols

_

B

C

D

F

H

M

O

P

R

S

U

V