Aprenda a Pensar Como un Programador

con Python

Aprenda a Pensar Como un Programador

con Python

Allen Downey
Jeffrey Elkner
Chris Meyers

Traducido por
Angel Arnal
I Juanes
Litza Amurrio
Efrain Andia

Green Tea Press

Wellesley, Massachusetts

Corregido por Shannon Turlington y Lisa Cutler.

Diseño de la cubierta por Rebecca Gimenez.

Green Tea Press
1 Grove St.
P.O. Box 812901
Wellesley, MA 02482

La forma original de este libro es LaTeX código fuente. Compilando este LaTeX fuente tiene el efecto de generar un dispositivo-independiente repesentacion de un libro de textos, cual puede ser convertido a otros formatos y imprimido.

El LaTeX fuente para este libro y mas información del Libro de Texto de Código Abierto proyecto esta disponible de

`http://www.thinkpython.com`

Este libro estaba compuesto tipo utilizando LaTeX y LyX. Los illustraciones estaban dibujados en xfig. Todos estes son gratis, programas de código abierto.

Historia de la impresion:

April 2002: Primera edición.

ISBN 0-9716775-0-6

Introducíon

Por David Beazley

Como un educador, investigador, y autor de libro, estoy encantado a ver el completación de este libro. Python es divertido y una extremadamente facil-para-usar leguaje de programación que ha ganado constantemente en renombre concluido los ultimos años. Desarrolladó hace diez años por Guido van Rossum, el sintaxis simple de Python y la sensación total se deriva en gran parte del ABC, un lenguaje de la enseñanza que desarrolló en los 1980s. Sin embargo, Python tambien fue creado para resolver problemas reales y presta una variedad amplia de características de los lenguajes de programaciónes como C++, Java, Modula-3, y Scheme. Debido a esto, uno de los características notables de Python es su amplia attracción a los reveladores profesionales de progamación, scientistas,investigadores, artistas, y educadores.

A pesar del attracción de Python a muchos diversos communidades, usted puede todavia preguntarse "porque Python?" o "porque enseñar programación con Python?" Respondiendo a estos preguntas no es una simple tarea— especialmente cuando el opinion popular esta en el lado de mas masoquista alternativas como C++ y Java. Sin embargo, pienso la respuesta mas directa es que la progrmación en Python es simplemente mas divertido y mas productiva.

Cuando enseño cursos de informática, yo quiero cubrir conceptos importantes y hacer el material interesante y enganchar a los estudiantes. Desafortunadamente, hay una tendencia para que los cursos de programación introductorios se centren demasiada atención en el abstracción matemático y para hacer que los estudiantes se frustran con problemas molestosas relacionadas a detalles bajos del sintaxis, compilación, y la applicación aparentemente de reglas arcanes. Aunque tal abstracion y formalismo son importantes a ingenieros profesionales de programación y estudiantes que planean a continuar sus estudios de informática, hablando de tal acercamiento en un curso introductor solo tiene éxito en haciendo informática banal. Cuando enseño un curso, yo no quiero tener un curato de estudiantes sin inspiración. Quisiera verlos intentando resolver

problemas interesantes en explorando ideas differentes, hablando acrecamientos no convencionales, rompiendo las reglas, y aprendiendo de sus errors. En haciendolo, yo no quiero perder la mitad del semestre tratando de clasificar problemas sintaxis obscuros, mensajes de error del compilador incomprensibles, olos varios cientos de maneras que una programa puede generar una proteción fallo general.

Una de las razones porque me gusta Python es que proporcion un equilibrio muy buena entre el práctico y el conceptual. Puesto que se interpreta Python, los principiantes pueden recoger el lenguaje y empesar haciendo cosas interesantes casi inmediato sin perdiendose el los problemas de compilaciòn y conexión. Además, Python viene con un biblioteca grande de módulos que pueden ser usados para hacer toda clase de tareas que se extienden de web-programming a gráficos. Teniendo un foco práctico es una gran manera de enganchar a estudiantes y permite que completan proyectos significativos. Sin embargo, Python también puede servir como una fundación excelente para intruducir conceptos importantes de la informática. Puesto que Python soporta completamente procedimientos y clases, los estudiantes pueden ser introducidos gradualmente a los asuntos como abstracción procesal, estructuras de datos, y programación orientada al objeto—cuáles son aplicables a cursos posteriores en Java or C++. Python incluso pide presta un número de características de lenguajes de programación funcionales y puede ser usado para intruducir conceptos que pudiera ser cubierta en mas detalles en coursos en Scheme y Lisp.

Leendo, el prefacio de Jeffrey, yo estoy sorprendido por sus comentarios que Python lo permita a ver un "más alto nivel de éxito y un nivel bajo de frustración" y que el podia "mover rapido con mejor resultados." Aunque estos comentarios refieren a su curos introductorio, yo al veces uso Python por estes mismos razones en los cursos de informática advancada en el Universidad de Chicago. En estes cursos, yo estoy constantemente enfrentando con la tarea desalentadora de cubriendo un monton de material difíl del curso en una cuarta de nueve semanas. Aunque es ciertamente posible para que inflija mucho dolor y sufrimiento usando un lenguaje como C++, he encontrado a menudo este manera para ser ineficaz—especialmente cuando el curso se trata de un asunto sin relación apenas a la "programación." Yo encuentro que usando Python me permita a más enfocar en el asunto actual a mano mientras dejando estudiantes complentar projectos substanciales.

Aunque Python siga es un lenguaje joven y desarollando, creo que tiene un futuro brillante en la edicación. Este libro is un paso importante en esa direcció.

David Beazley
Universidad de Chicago
Autor del *Python Essential Reference*

Prefacio

Por Jeff Elkner

Este libro debe su exsistencia al collaboración hecho posible por el Internet y el free software movement. Sus tres autores—un profesor de colegio, un profesor de secundaria, y un programador profesional—tienen todavía verse cara a cara, pero hemos podido a trabajar juntos y hemos sido ayudado por maravillosas personas quienes han donado su tiempo y energia a ayudar a hacer este libro verse mejor.

Nosotros pensamos que este libro es un testamento a los beneficios y futuras posibilidades de este clase de coloboración, el marco por el cual se ha puesto en lugar por Richard Stallman y el Free Software Foundation.

Cómo y porqué vino a utilizar Python

En 1999, el examen del College Board's Advanced Placement (AP) de Informática fue dado en C++ por primera vez. Como en muchas escuelas a través del país,la decisiós para cambiar el lenguaje tenía un impacto directo en el plan de estudios de la informt́ica en la escuela secudaria de Yorktown en Arlington, Virginia, donde enseño. Hasta este punto, Pascal era el lenguaje de instrucción en nuestros cursos del primer año y del AP. In keeping with past practice de dar a estudiantes dos años de exposición al mismo lenguaje, tomamos la decisión para cambiar a C++ en el curso del primer año por el año escolar 1997-98 de modo que fuéramos en el paso de progresión con el College Board's change para el curso del AP el año siguiente.

Dos anos despues, estoy convincido que C++ era un opcion no buena para utilizar para introducir a estudiantes la informatica. Mientras que ciertamente un lenguaje de programacion muy de gran alcance, es tambien un lenguaje extremadamente dif??icil a aprender y a ensenar. Me encontre constantemente peleando con la syntax difil de C++ y m??ultiples maneras de hacer cosas, y

estaba perdiendo muchos estudiantes innecesariamente como resultado. Convencido que habia que haber un opcion mejor para nuestros primer-ano clases, fui en busca por un alternativa a C++.

Necesiaba un lenguaje que podia correr en los macinas en nuestro laboratorio Linux tambien en los plataformas de Windows y Macintosh que muchos de los estudiantes tienen en casa. Quieria que sea fuente abierta, para que estudiantes lo podian usar en casa sin inportar su renta. Queria un lenguaje usado por programadores profesionales, y uno que tenia una comunidad activa alrededor de el. Tenia que soportar la programacion procesal y orientada al objecto. Y mas importante, tenia que ser facil para aprender y ensenar. Quando investige las opciones con estos metas en mente, Python salto como el mejor candidato al trabajo.

Pedí a uno de los estudiantes mas talentosos de Yorktown, Matt Ahrens, que diera a Python un intento. En dos meses el no solomente aprendio el lenguaje pero escribio una appicacion llamado pyTicket que permito?? a nuestro personal senalar problemas de la tecnologia via el Web. Sabia que Matt no podria teminar una aplicacion de ese escala en tan poco tiempo con C++, y este realizacion, combinada con el gravamen positivo de Matt sobre Python, sugerio que Python era la solucion que buscaba.

Encontrando un Libro de Textos

Al decidir a utilizar Python en ambas de mi clases informatica introductoria el año siguiente, el problema mas acuciante era la carencia de un libro de textos disponible.

Contenido gratis vino al rescate. A principios del año, Richard Stallman me introducio a Allen Downey. Los dos habiamos escirto a Richard expresando un interes en desarrollando un contenido gratis y educativo. Allen ya habia escrito un libro de textos del primer añ de la informática, Como Pensar como un Progrmador. Quando lei este libro, sabia inmediatamente que yo quieria usarlo en mi clase. Era el texto más claro y mas provechoso de la imformatica que habia visto. Acentuoá los procesos del pensamiento implicados en la programacion mas bien que las carateriásticas de un lenguaje particular. Leendolo inmediatamente me hiso sentir un profesor mejor. *Como Pensar como un Programador com Python no solo fue un libro excelente, pero habia sido released bajo un GNU licensia publica, qual significa que podria se utilizada libermente y ser modificada para resolver las necesidades de su utilizador. Una vez que decidiera utilizar Python, se me ocurriá que podria traducir la version original de Java de Allen del libro al nuevo lenguaje. Mientras que no podria escribir un libro de textos solo, teniendo el libro de Allen me hiso possible para que yo haga asi, en el*

mismo tiempo demonstrando que el modelo cooperativo del desarrollo usado en software logica podria tambien trabajar por contenido educativo.

Trabajando en este libro por los dos ultimos años ha sido un recompensa a mis estudiantes y a mi, y mis estudiantes jugaron un grande parte en el proceso.Puesto que podria realizar cambios inmediatos simpre que alguien encontrara un error de deletreo o un paso difícil, yo les animara a que buscaran errores en el libro dandoles una punta cada vez que hicieron una sugerencia que resulto en un cambio en el texto. Eso tenia la ventaja doble de animarles a que lean el texto mas cuidadosamente y de conseguir el texto repasado por sus cri??ticos mas importantes, estudiantes usa??ndolo para aprender la informatica.

Por el segundo parte del libro en orientada al object programacion, sabia que alguien con un mas experiencia con programacion que yo sea necesario para hacerlo correcto. El libro se sentó en un estado inacabado para la parte mejor de un año hasta que la comunidad abierta de la fuente proporciono de nuevo los medios necesarios para su terminación.

Recivi un email de Chris Meyers expresando interes en el libro. Chris es un programador profesional que empezo ensenado un curso de programacion el ano pasado usando Python en Lane Community College en Eugene, Oregon. La perspectiva de ensenar el curso llevo a Chirs al libro, y el comenzo ayudando con el inmediatamente. Antes de fin de ano escolar el habi?a creado un proyecto acompaniante sobre nuestro Webiste `http://www.ibiblio.org/obp` *titulado* Python for Fun *y estaba trabajando con algunos de mis estudiantes mas advanzados como un profesor principal, diriendolos mas alla donde podria tomarlos.*

Introduciendo Programacion con Python

El proceso de tranduccion y usando Como Pensar como un Programdor con Python por los ultimos dos anos han confirmado la conveniencia de Python para ensenar estudiantes principiantes. Python simplifica grandemente ejemplos de programacion y hace ideas de programacion importantes mas faciles a ensenar.

El primer ejemplo del texto illustra este punto. Es el tradicional “hola, mundo” programa, qual en el version de C++ version del libro se ve asi:

```
#include <iostream.h>

void main()
{
  cout << "Hola, mundo." << endl;
}
```

en el version Python version es:

```
print "Hola, Mundo!"
```

Aunque este es un ejemplo trivial, los aventajes de Python resalta. El curso de-Informtica I en Yorktown no tiene prerrequisitos, es por eso que muchos de los estudiantes viendo este ejemplo estn mirando a su primer programa. Algunos-de ellos son un poco nerviosos, escuchando que programacin de computacin es difcil a aprender. La versin C++ siempre me a forzado a escoger entre dos opciones insatisfechas: explicar el `#incluido`, `void main()`, {, y } declaraciones y arriesgar a confundir o intimidar algunos de los estudiantes en el principio, o decirles, "No te preocupes de todo eso ahora; Lo hablares mas tarde,τ arriesgar la misma cosa. Los objetivos educacionales en este momento en el curso son para introducir estudiantes a la idea de un declaraciones programacin y llevndoles a escribir sus primer programacin, asintroducindoles al la ambiente de programacin. El programacin Python tiene exactamente lo que necesito para hacer estas cosas, y nada ms.

Comparando el texto explicativo de este programa en cada versin del libro ilustra mas que significa este a los estudiantes principiantes. Hay trece prrafos de explicacin de "Hola, mundo!"en el versin C++; En el versin Python, solo hay dos. Ms importante, programacin de computacin pero con el minucias de sintaxis C++. Encontr esta misma cosa pasando por todo el libro. Prrafos enteros desapareciendo del versin Python del texto porque el sintaxis claro de Python los hace innecesario.

Usando un lenguaje de muy alto-nivel como Python, deja a un profesor posponer hablando de los detalles bajo-nivel de la maquina hasta los estudiantes tienen el background que ellos necesitan para entender los detalles. Crea la habilidada poner "primer cosas primero"pedagogically. Unos de los mejores ejemplos de este es la manera en cual Python maneja variables. En C++ un variable es un nombre para un lugar que agarra una cosa. Variables tienen que ser declarados con tipos por lo menos en parte porque el tamao del lugar a cual refieren tieneque ser predeterminado. As, la idea de un variable es vendada con el hardware de la maquina. El concepto poderoso y fundamental de un variable ya es difcil para estudiantes principiantes (ambos en la informtica y lgebra. Octetos y direcciones no ayudan la materia. En Python un variable en un nombre que refiere a una cosa. Este es mucho ms un concepto intuito para estudiantes principiantes y es mas cerca del significado de "variable"que aprendieron este ao que yo en el pasado, y yo tarde menos tiempo en ayudndolos con problemas usndolos.

Un otro ejemplo de cmo Python ayuda en la enseanza y aprenda de programacin es en su sintaxis por funciones. Mis estudiantes siempre han tendido un gran dificultad comprendiendo funciones. El problema principal se centra alrededor de la diferencia entro una funcin definicin y una funcin llamada, y la distincin relacionada entre un parmetro y un argumento. Python viene al rescate con sintaxis que es nada corto de bella. Funcin definicin empieza conla palabra clave `def`, y

yo simplemente digo a mis estudiantes, Çuando define una funcin, empiece con `def`, seguido del nombre de la funcin que estas definiendo; Cuando llamas una funcin, simplemente llama (tipo) su nombre."Parmetros van con definiciones; argumentos van con llamadas. No hay tipos retornes, tipos parmetros, o parmetro referencia y valor a conseguir de la manera, y ahorra yo puedo ensear funciones en la mitad de tiempo que antes, con mejor comprensin.

Usando Python hay mejorado la eficacia de nuestro programa informtica para todos los estudiantes. Veo un nivel general alto de xito y un nivel bajo de frustracin que yo he pasado durante los dos aos que ensee C++. Muevo ms rpido con mejor resultados. Mas estudiantes terminan el curso con la habilidada crear programas significativos y con la actitud positivo hacia la experiencia de programacin que este engendra.

Construyendo una Comunidad

He recibido un email de todos partes del mundo de personas usando este libro a aprender o ensear programacin. Una comunidad de utilizador ha comenzado a emerger, y muchas personas han contribuido al proyecto mandando materiales por el Website acompaante a `http://www.thinkpython.com`.

Con la publicacin de este libro en forma emprime, espero la crecimiento en la comunidad utilizador a continuar y acelerar. El emergenicia de este user comunidad utilizador y la posibilidad que sugiere para colaboracin similar entre educadores han sido los partes ms excitantes de trabajando en este proyecto para mi. Trabajando juntos, nosotros podemos increase la calidad de materias disponibles para nuestro uso y ahorrar tiempo valioso.

Yo les invito a nuestra comunidad y espero eschuchar de ustedes. Por favor escribe a los autores a `feedback@thinkpython.com`.

Jeffrey Elkner
Escuela Secundaria Yortown
Arlington, Virginia

Lista de los Contribuidores

Para parafrasear la filosifía del Free Software Foundation, este libro es gratis como libre discurso, pero no necesariamente gratis como pizza. Vino alrededor debido a una colaboracíon que no seria posible sin el GNU Free Documentation License. Quisieramos agradecer el Free Software Foundation por desarollando este licensia y, por supuesto, poniéndolo a nuestro disposición.

Nosotros quisieramos a agradecer a los mas de 100 sharp-eyed and thoughtful leedores quienes nos han mandado sugesiones y correciones durante los años pasados. En el espiritu de free software, dicidimos a expresar nuestro agradecimiento en la forma de un lista de contribuidores. Desafortunadamente, esta lista no esta completa, pero nosotros estamos haciendo nuestro mejor esfuerzo para maternerlo actulizado.

Si tiene un chance a leer la lista, deberia realizar que cada persona aqui le ha ha ahorrado a usted y todos leedores subsecuentes de la confusión de un error técnico o de una explicación menos-que-transparente, solo mandandonos una nota.

Imposible como puede parecerse después de tantas correcciones, puede todavía haber errores en este libro. Si vee uno, esperamos que toma un minuto para contactarnos. El correo electronico es `feedback@thinkpython.com`. Si hacemos un cambio debido a su sugerencias, apareceras en la siguiente version de la lista de contribuidores (al menos que usted pida ser omitido). Gracias!

- Lloyd Hugh Allen envió una corrección a la Sección 8.4.
- Yvon Boulianne envió una corrección de error semantico en Capítulo 5.
- Fred Bremmer sometió una corrección en Sección 2.1.
- Jonah Cohen escribió las escrituras de Perl para convertir la fuente LaTeX de este libro a bella HTML.

- Michael Conlon envió una corrección de gramatica en Capítulo 2 y un progreso en estilo en Capítulo 1, y el inició discusión en los aspectos técnicos de interpretadores.
- Benoit Girard envió una corrección a un error chistoso en Sección 5.6.
- Courtney Gleason y Katherine Smith escribió `horsebet.py`, que estaba usada como un caso de estudio en un version anterior de este libro. Su programa se puede encontrar en el website.
- Lee Harr sometió mas correcciones que tenemos campo para enumar aqui, y por supesto el deberia ser listado como uno de los editores principales del texto.
- James Kaylin es un estudiante usando el texto. El ha sometido numerosas correcciones.
- David Kershaw arrelgó las funcciones rotos `catTwice` en Seción 3.10.
- Eddie Lam ha enviado numerosas correccions a Capítulos 1, 2, y 3. El tambíen arregló el Makefile para que creara un índice la primera ves esta generado y nos ayudó a instala un esquema versioning.
- Man-Yong Lee envió una corrección al codigo ejemplo en Sección 2.4.

  in Chapter 1 needed to be changed to "subconsciously".
- Chris McAloon envió varias correcciones a Secciones 3.9 y 3.10.
- Matthew J. Moelter ha sido un contribuidor de mucho tiempo que envió numerosas correcciones y sugerencias al libro.
- Simon Dicon Montford reportó una definición función que faltaba y varios errores en Capítulo 3. El tambien encontró errores en el `increment` función en Capítulo 13.
- John Ouzts corregió la definición de "return value"en Capítulo 3.
- Kevin Parks envió comentas valuosas y sugerencias para mejorar la distrubuicion del libro.
- David Pool envió un error en el glosario de Capíltuo 1, tambien palabras de estímulo.
- Michael Schmitt envió una corrección al capítulo de files y exceptions.
- Robin Shaw punto un error en Sección 13.1, donde el función printTime estaba usada en un ejemplo sin estar definido.

- Paul Sleigh encontró un error en Capítulo 7 y un bug en la escritura de Perl de Jonah Cohen que genera HTML de LaTeX.
- Craig T. Snydal is testing the text in a course at Drew University. He has contributed several valuable suggestions and corrections.
- Ian Thomas and his students are using the text in a programming course. They are the first ones to test the chapters in the latter half of the book, and they have make numerous corrections and suggestions.
- Keith Verheyden sent in a correction in Chapter 3.
- Peter Winstanley let us know about a longstanding error in our Latin in Chapter 3.
- Chris Wrobel hizo correcciones al codigo en el capítulo sobre file I/O y exceptions.
- Moshe Zadka hizo comtribuciones inestimables a este proyecto. Además de escribi el primer bosquejo del capítulo sobre Diccionarios, él proporcionó a la dirección continua en los primeros tiempos del libro.
- Christoph Zwerschke envió varias correcciones y sugerencias pedagógicas, y explico la diferencia entre *gleich* y *selbe*.
- James Mayer nos envió una ciénaga entera de deletreo y de errores topgráficos, incluyendo dos en la lista de contribuidores.
- Hayden McAfee cogió una inconsistencia potencialmente confusa entre dos ejemplos.
- Angel Arnal es parte de un equipo internacional de los traductores que trabanjan en la versión españo del texto. Él también ha encontrado varios errores en la versión ingles.

Índice general

Capítulo 1

El Camino del Programa

El objetivo de este libro es de ensear al estudiante a pensar como lo hacen los cientficos informticos. Esta manera de pensar combina las mejores caracterstica de la matemtica, ingeniera, y las ciencias naturales. Como los matemticos, los cientficos informticos usan lenguas formales para designar ideas (especficamente, computaciones). Como los ingenieros, ellos disean cosas, ensamblando sistemas de componentes y evaluando ventajas y desventajas de cada una de las alternativas. Como los cientficos, ellos observan el comportamiento de sistemas complejos, forman hiptesis, y prueban sus predicciones.

La habilidad ms importante del cientfico informtico es **la solucin de problemas**. La solucin de problemas incluye poder formular problemas, pensar en la solucin de manera creativa, y expresar una solucin clara y precisamente. Como se ver, el proceso de aprender a programar es la oportunidad perfecta para desarrollar la habilidad de resolver problemas. Por esa razn este captulo se llama "El Camino del programa."

A cierto nivel, usted aprender a programar, lo cual es una habilidad muy til por s misma. A otro nivel, usted utilizar la programacin para obtener algn resultado. Ese resultado se vera mas claramente durante el proceso.

1.1. El lenguaje de programacin Python

El lenguaje de programacin que aprender es Python. Python es un ejemplar de un **lenguaje de alto nivel**; otros ejemplos de lenguajes de alto nivel son C, C++, Perl y Java.

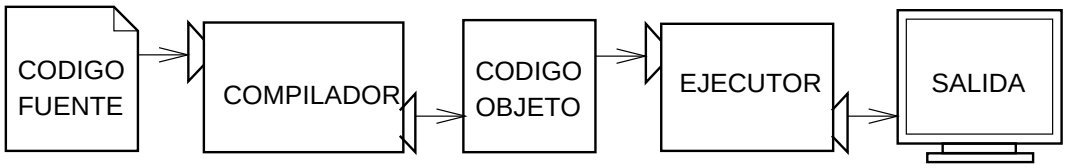
Como se puede deducir de la nomenclatura "lenguaje de alto nivel," tambin existen **lenguajes de bajo nivel**, que tambin se los califica como lenguaje de mquina o lenguaje de ensamblado. A propsito, las computadoras slo ejecutan programas escritos en lenguajes de bajo nivel. Los programas de alto nivel tienen que ser traducidos antes de ser ejecutados. Esta traduccin lleva tiempo, lo cual es una pequea desventaja de los lenguajes de alto nivel.

Aun as las ventajas son enormes. En primer lugar, la programacin en lenguajes de alto nivel es mucho ms fcil; escribir programas en un lenguaje de alto nivel toma menos tiempo, los programas son mas cortos y mas fciles de leer, y es mas probable que estos programas sean correctos. En segundo lugar, lenguajes de alto nivel son **porttiles**, lo que significa que pueden ser ejecutados en tipos diferentes de computadoras sin modificacin alguna o con pocas modificaciones. Programas escritos en lenguajes de bajo nivel solo pueden ser ejecutados en un tipo de computadora y deben ser re-escritos para ser ejecutados en otro.

Debido a estas ventajas, casi todo programa es escrito en un lenguaje de alto nivel. Los lenguajes de bajo nivel son slo usados para unas pocas aplicaciones especiales.

Hay dos tipos de programas que traducen lenguajes de alto nivel a lenguajes de bajo nivel: **interpretadoras** y **compiladores**. Una interpretadora lee un programa de alto nivel y lo ejecuta, lo que significa que lleva a cabo lo que indica el programa. Traduce el programa poco a poco, leyendo y ejecutando cada comando.

Un compilador lee el programa y lo traduce todo al mismo tiempo, antes de ejecutar alguno de los programas. A menudo se compila un programa como un paso a parte, y luego se ejecuta el cdigo compilado. En este caso, al programa de alto nivel se lo llama el **cdigo fuente**, y al programa traducido es llamado el **cdigo de objeto** o el **cdigo ejecutable**.



A Python se lo considera un lenguaje interpretado porque programas de Python son ejecutados por una interpretadora. Existen dos maneras de usar la interpretadora: modo de comando y modo de guin. En modo de comando se escriben sentencias en el lenguaje Python y la interpretadora muestra el resultado.

```
$ python
Python 1.5.2 (#1, Feb 1 2000, 16:32:16)
Copyright 1991-1995 Stichting Mathematish Centrum, Amsterdam
>>>print 1 + 1
2
```

La primera lnea de este ejemplo es el comando que pone en marcha a la interpretadora Python. Las dos lneas siguientes son mensajes de la interpretadora. La tercera linea comienza con `>>>`, lo que es la invitacin de la interpretadora para indicar que est listo. Escribimos `1+1` y la interpretadora contest `2`.

Alternativamente, se puede escribir el programa en un archivo y usar la interpretadora para ejecutar el contenido de dicho archivo. El archivo es, en este caso, llamado un **guin**. Por ejemplo, en un editor de texto se puede crear un archivo `latoya.py` que contenga esta lnea:

```
print 1 + 1
```

Por acuerdo unnime, los archivos que contienen programas de Python tienen nombres que terminan con `.py`.

Para ejecutar el programa, se le tiene que indicar el nombre del guin a la interpretadora.

```
$ python latoya.py
2
```

En otros entomos de desarrollo los detalles de la ejecucin de programas diferirn. Tambin, la mayora de programas son mas interesantes que el mencionado.

La mayora de ejemplos en este libro son ejecutados en la lnea de comando. La lnea de comando es ms conveniente para el desarrollo de programas y para pruebas rpidas porque se pueden pasar a mquina las instrucciones de Python y pueden ser ejecutados inmediatamente. Una vez que un programa esta completo, se lo puede archivar en un guin para ejecutarlo o modificarlo en el futuro.

1.2. Qu es un programa?

Un programa es una secuencia de instrucciones que especifican como ejecutar una computacin. La computacin puede ser algo matemtico, como solucionar un sistema de ecuaciones o determinar las races de un polinomio, pero tambin

pueden ser una computacin simblica, como buscar y reemplazar el texto de un documento o (aunque parezca raro) compilar un programa.

Las instrucciones (comandos, rdenes) tienen una apariencia diferente en lenguajes de programacin diferentes, pero existen algunas funciones bsicas que se presentan en casi todo lenguaje:

entrada: Recibir datos del teclado, o un archivo o otro aparato.

salida: Mostrar datos en el monitor o enviar datos a un archivo u otro aparato.

matemticas: Ejecutar operaciones bsicas de matemticas como la adicin y la multiplicacin.

operacin condicional: Probar la veracidad de alguna condicin y ejecutar una secuencia de instrucciones apropiada.

repeticin Ejecutar alguna accin repetidas veces, usualmente con alguna variacin.

Aunque sea difcil de creer, todos los programas en existencia son formulados exclusivamente de tales instrucciones. As, una manera de describir la programacin es: El proceso de romper una tarea en tareas cada vez ms pequeas hasta que estas tareas sean suficientemente simples para ser ejecutada con una de estas simples instrucciones.

Quizs esta descripcin es un poco ambigua. No se preocupe. Explicaremos con mas detalle con el tema de **algoritmos**.

1.3. Qu es la depuracin (debugging)?

La programacin es un proceso complejo y a veces este proceso lleva a **errores indefinidos**, tambin llamados **defectos** o **errores de programacin** (en ingls 'bugs') y el proceso de buscarlos y corregirlos es llamado **depuracin** (en ingls 'debugging').

Hay tres tipos de errores que pueden ocurrir en un programa. Es muy til distinguirlos para encontrarlos mas rpido.

1.3.1. Errores sintcticos

Python slo puede ejecutar un programa si el programa est correcto sintcticamente. Contrariamente, es decir si el programa no esta correcto sintcticamente, el proceso falla y devuelve un mensaje de error. El trmino **sintctica** se refiere a

la estructura de cualquier programa y a las reglas de esa estructura. Por ejemplo, en espaol la primera letra de toda oracin debe ser mayscula. esta oracin tiene un error sintctico. Esta oracin tambin

Para la mayora de lectores, unos pocos errores no impiden la comprensin de la poesa de e e cummings, la cual rompe muchas reglas de sintaxis. Sin embargo Python no es as. Si hay aunque sea un error sintctico en el programa, Python mostrar un mensaje de error y abortar la ejecucin del programa. Al principio usted pasar mucho tiempo buscando errores sintcticos, pero con el tiempo no tendr tantos errores y los encontrar mas rpido.

1.3.2. Errores de tiempo de ejecucin

El segundo tipo de error es un error de tiempo de ejecucin. Este error aparece slo cuando de ejecuta el programa. Estos errores tambin se llaman **excepciones** porque indican que algo excepcional ha ocurrido.

Con los programas que vamos a escribir al principio, errores de tiempo de ejecucin ocurrirn con poca frecuencia.

1.3.3. Errores semnticos

El tercer tipo de error es el **error semntico**. Si hay un error de lgica en su programa, el programa ser ejecutado sin ningn mensaje de error, pero el resultado no ser el deseado. El programa ejecutar la lgica que usted le dijo que ejecutara.

A veces ocurre que el programa escrito no es el programa que se tena en mente. El sentido o significado del programa no es correcto. Es difcil hallar errores de lgica. Eso requiere trabajar al revs, comenzando por la salida (output) para encontrar al problema.

1.3.4. Depuracin experimental

Una de las tcnicas ms importantes que usted aprender es la depuracin. Aunque a veces es frustrante, la depuracin es un proceso interesante, estimulante e intelectual.

La depuracin es una actividad parecida a la tarea de un investigador: se tienen que estudiar las claves para inducir los procesos y eventos the llevarn a los resultados previstos.

La depuracin tambin es una ciencia experimental. Una vez que se tiene la idea de el error, se modifica el programa y se intenta nuevamente. Si la hiptesis fue

la correcta se pueden predecir los resultados de la modificacin y estaremos mas cerca a un programa correcto. Si la hiptesis fue la errnea tendr que idearse otra hiptesis. Como dijo Sherlock Holmes, "Cuando se ha descartado lo imposible, lo que queda, no importa cuan inverosmil, debe ser la verdad." (A. Conan Doyle, *The Sign of Four*)

Para algunas personas, la programacin y la depuracin son lo mismo: la programacin es el proceso de depurar un programa gradualmente hasta que el programa tenga el resultado deseado. Esto quiere decir que el programa debe ser, desde un principio, un programa que funcione, aunque su funcin sea solo mnima. El programa es depurado mientras el crece y se desarrolla.

Por ejemplo, aunque el sistema operativo Linux contenga miles de lneas de instrucciones, Linus Torvalds lo comenz como un programa para explorar el microprocesador Intel 80836. Segn Larry Greenfield, "Uno de los proyectos tempranos de Linus fue un programa que intercambiara la impresin de AAAA con BBBB. Este programa se volvi en Linux" (de *The Linux Users' Guide* Versin Beta 1).

Otros captulos tratarn ms con el tema de depuracin y otras tcnicas de programacin.

1.4. Lenguajes formales y lenguajes naturales

Lenguajes naturales son los lenguajes hablados por seres humanos, como el espaol, el ingls y el francs. Estos no han sido diseados artificialmente (aunque se traten de imponer), pues se han desarrollado naturalmente.

Lenguajes formales son lenguajes que son diseados por humanos y que tienen aplicaciones especficas. La notacin matemtica, por ejemplo, es un lenguaje formal ya que se presta a la representacin de las relaciones entre nmeros y smbolos. Los qumicos utilizan un lenguaje formal para representar la estructura qumica de las molculas. Es necesario notar que:

> **Lenguajes de programacin son lenguajes formales que han sido desarrollados para expresar computaciones.**

Los lenguajes formales casi siempre tienen reglas sintcticas estrictas. Por ejemplo, $3 + 3 = 6$ es una expresin matemtica correcta, pero $3 = +6\$$ no lo es. De la misma manera, H_20 es una nomenclatura qumica correcta, pero ${}_2Zz$ no lo es.

Existen dos clases de reglas sintcticas, en cuanto a unidades y estructura. Las unidades son los elementos bsicos de un lenguaje, como lo son las palabras, los nmeros y los elementos qumicos. Por ejemplo, en `3=+6$`, `$` no es una unidad

matemtica aceptada. Similarmente, ${}_2Xx$ no es formal porque no hay ningn elemento con la abreviacin Zz.

La segunda clase de error sintctico est relacionado con la estructura de un elemento; mejor dicho, el orden de las unidades. La estructura de la sentencia `3=+6$` no es aceptada porque no se puede escribir el smbolo de igualdad seguido de un smbolo positivo. Similarmente, las frmulas moleculares tienen que mostrar el nmero de subndice despus del elemento, no antes.

> *A manera de prctica, trate de producir una oracin con estructura aceptada pero que es compuesta de unidades irreconocibles. Luego escriba otra oracin con unidades aceptables pero con estructura no vlida.*

Al leer una oracin, sea en un lenguaje natural o una sentencia en un lenguaje tcnico, se debe discernir la estructura de la oracin. En un lenguaje natural este proceso, llamado **anlisis sintctico** ocurre subconcientemente.

Por ejemplo cuando se escucha una oracin simple, se puede distinguir el sustantivo y el verbo. Cuando se ha analizado la oracin sintcticamente, se puede deducir el significado, o la semntica, de la oracin. Si se conoce el sustantivo y el verbo, se entender el significado de la oracin.

Aunque existen muchas cosas en comn entre los lenguajes naturales y los lenguajes formales—por ejemplo las unidades, la estructura, la sintctica y la semntica—pero tambin existen muchas diferencias.

ambigedad: Los lenguajes naturales tienen muchsimas ambigedades, las que se entienden usando claves contextuales y otra informacin. Lenguajes formales son diseados para ser completamente libres de ambigedades o tanto como sea posible, lo que quiere decir que cualquier sentencia tiene slo un significado, no importe el contexto.

redundancia: Para reducir la ambigedad y los malentendidos, las lenguas naturales utilizan bastante redundancia. Como resultado tienen una abundancia de posibilidades para expresarse. Lenguajes formales son menos redundantes y mas concisos.

calidad literal: Los lenguajes naturales tienen muchas metforas y frases hechas. El significado de un dicho, por ejemplo "Estirar la pata", es diferente al significado de sus sustantivos y verbos. En este ejemplo, la oracin no tiene nada que ver con un pie y significa 'morirse'. Los lenguajes formales no difieren de el significado literal.

Los que aprenden a hablar un lenguaje natural—es decir todo el mundo—muchas veces tienen dificultad en adaptarse a los lenguajes formales. A veces la diferencia entre los lenguajes formales y los naturales es comparable a la diferencia entre la prosa y la poesa:

Poesa: Se utiliza una palabra por su cualidad auditiva tanto como por su significado. El poema, en su totalidad, produce un efecto o reaccin emocional. La ambigedad no es solo comn sino utilizada a propsito.

Prosa: El significado literal de la palabra es mas importante y la estructura contribuye mas significado aun. La prosa se presta al anlisis mas que la poesa, pero todava contiene ambigedad.

Programas: El significado de un programa es inequvoco y literal, y es entendido en su totalidad analizando las unidades y la estructura.

E aqu unas sugerencias para la lectura de un programa (y de otros lenguajes formales). Primero, recuerde que los lenguajes formales son mucho mas densos que los lenguajes naturales, y por consecuente lleva mas tiempo para leerlos. Tambin, la estructura es muy importante, entonces no es una buena idea leerlo de pies a cabeza, de izquierda a derecha. En vez, aprenda a separar las diferentes partes en su mente, identificar las unidades y interpretando la estructura. Finalmente, ponga atencin en los detalles. La fallas de puntuacin y la ortografa afectarn negativamente la ejecucin de su programa.

1.5. El primer programa

Tradicionalmente el primer programa en un lenguaje nuevo se llama "Hola todo el mundo!" (Hello world!) porque solo muestra las palabras "Hola todo el mundo" (Hello world!). En el lenguaje Python es as:

```
print "Hola todo el mundo!"
```

Este es un ejemplo de una sentencia *print*, la cual no imprime nada en papel, mas bien muestra un valor. En este caso, el resultado es las palabras

```
Hola todo el mundo!
```

Las comillas sealan el comienzo y el final del valor; no aparecen en el resultado.

Alguna gente evala la calidad de un lenguaje de programacin por la simplicidad de el programa "Hola todo el mundo!". Si seguimos ese criterio, Python cumple con todas sus metas.

1.6. Glosario

solucin de problemas: El proceso de formular un problema, hallar la solucin y expresar la solucin.

lenguaje de alto nivel: Un lenguaje como Python que es diseado para ser fcil de leer y escribir para la gente.

lenguaje de bajo nivel: Un lenguaje de programacin que es diseado para ser fcil de ejecutar para una computadora; tambin se lo llama "lenguaje de maquina" o "lenguaje de ensamblado".

portabilidad: La cualidad de un programa que puede ser ejecutado en mas de un tipo de computadora.

interpretar: Ejecutar un programa escrito en un lenguaje de alto nivel traducindolo lnea por lnea

compilar: Traducir un programa escrito en un lenguaje de alto nivel a un lenguaje de bajo nivel todo al mismo tiempo, en preparacin para la ejecucin posterior.

cdigo fuente: Un programa escrito en un lenguaje de alto nivel antes de ser compilado.

cdigo de objeto: La salida del compilador una vez que el programa ha sido traducido.

programa ejecutable: Otro nombre para el cdigo de objeto que est listo para ser ejecutado.

guin: Un programa archivado (que va a ser interpretado).

programa: Un grupo de instrucciones que especifica una computacin.

algoritmo: Un proceso general para resolver una clase completa de problemas.

error (bug): Un error en un programa.

depuracin: El proceso de hallazgo y eliminacin de los tres tipos de errores de programacin.

sintaxis: La estructura de un programa.

error sintctico: Un error en un programa que hace que un programa sea imposible de analizar sintcticamente (e imposible de interpretar).

error de tiempo de ejecucin: Un error que no ocurre hasta que el programa ha comenzado a ejecutar e impide que el programa continue.

excepcin: Otro nombre para un error de tiempo de ejecucin.

error semntico: Un error en un programa que hace que ejecute algo que no era lo deseado.

semntica: El significado de un programa.

language natural: Cualquier lenguaje hablado que evolucion de forma natural.

lenguaje formal: Cualquier lenguaje diseado que tiene un propsito especfico, como la representacin de ideas matemticas o programas de computadoras; todos los lenguajes de programacin son lenguajes formales.

unidad: Uno de los elementos bsicos de la estructura sintctica de un programa, anlogo a una palabra en un lenguaje natural.

anlisis sintctico: La examinacin de un programa y el anlisis de su estructura sintctica.

sentencia print: Una instruccin que causa que la interpretadora Python muestre un valor en el monitor.

includechap02 clearemptydoublepage

Capítulo 2

Diccionarios

Los tipos compuestos que has visto hasta ahora (cadenas, listas y tuplas) usan enteros como ndices. Si intentas usar cualquier otro tipo como ndice provocars un error.

Los **diccionarios** son similares a otros tipos compuestos excepto en que pueden usar como ndice cualquier tipo inmutable. A modo de ejemplo, crearemos un diccionario que traduzca palabras inglesas al espaol. En este diccionario, los ndices son `strings` (cadenas).

Una forma de crear un diccionario es empezar con el diccionario vaco y aadir elementos. El diccionario vaco se expresa como {}:

```
>>> ing\_a\_esp = {}
>>> ing\_a\_esp['one'] = 'uno'
>>> ing\_a\_esp['two'] = 'dos'
```

La primera asignacin crea un diccionario llamado `ing_a_esp`; las otras asignaciones aaden nuevos elementos al diccionario. Podemos presentar el valor actual del diccionario del modo habitual:

```
>>> print ing\_a\_esp
{'one': 'uno', 'two': 'dos'}
```

Los elementos de un diccionario aparecen en una lista separada por comas. Cada entrada contiene un ndice y un valor separado por dos puntos (:). En un diccionario, los ndices se llaman **claves**, por eso los elementos se llaman **pares clave-valor**.

Otra forma de crear un diccionario es dando una lista de pares clave-valor con la misma sintaxis que la salida del ejemplo anterior:

```
>>> ing\_a\_esp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

Si volvemos a imprimir el valor de `ing_a_esp`, nos llevamos una sorpresa:

```
>>> print ing\_a\_esp
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

Los pares clave-valor no estn en orden! Afortunadamente, no necesitamos preocuparnos por el orden, ya que los elementos de un diccionario nunca se indexan con ndices enteros. En lugar de eso, usamos las claves para buscar los valores correspondientes:

```
>>> print ing\_a\_esp['two']
'dos'
```

La clave `'two'` nos da el valor `'dos'` aunque aparezca en el tercer par clave-valor.

2.1. Operaciones sobre diccionarios

La sentencia `del` elimina un par clave-valor de un diccionario. Por ejemplo, el diccionario siguiente contiene los nombres de varias frutas y el nmero de esas frutas en el almacn:

```
>>> inventario = {'manzanas': 430, 'bananas': 312, 'naranjas': 525, 'peras': 21
>>> print inventario
{'naranjas': 525, 'manzanas': 430, 'peras': 217, 'bananas': 312}
```

Si alguien compra todas las peras, podemos eliminar la entrada del diccionario:

```
>>> del inventario['peras']
>>> print inventario
{'naranjas': 525, 'manzanas': 430, 'bananas': 312}
```

O si esperamos recibir ms peras pronto, podemos simplemente cambiar el inventario asociado con las peras:

```
>>> inventario['peras'] = 0
>>> print inventario
{'naranajas': 525, 'manzanas': 430, 'peras': 0, 'bananas': 312}
```

La funcin `len` tambin funciona con diccionarios; devuelve el nmero de pares clave-valor:

```
>>> len(inventario)
4
```

2.2. Mtodos del diccionario

Un **mtodo** es similar a una funcin, acepta parmetros y devuelve un valor, pero la sintaxis es diferente. Por ejemplo, el mtodo `keys` acepta un diccionario y devuelve una lista con las claves que aparecen, pero en lugar de la sintaxis de la funcin `keys(ing_a_esp)`, usamos la sintaxis del mtodo `ing_a_esp.keys()`.

```
>>> ing\_a\_esp.values()
['uno', 'tres', 'dos']
```

Esta forma de notacin de punto especifica el nombre de la funcin, `keys`, y el nombre del objeto al que se va a aplicar la funcin, `ing_a_esp`. Los parntesis indican que este mtodo no admite parmetros.

La llamda a un mtodo se denomina **invocacin**; en este caso, diramos que estamos invocando `keys` sobre el objeto `ing_a_esp`.

El mtodo `values` es similar; devuelve una lista de los valores del diccionario:

```
>>> ing\_a\_esp.values()
['uno', 'tres', 'dos']
```

El mtodo `items` devuelve ambos, una lista de tuplas con los pares clave-valor del diccionario:

```
>>> ing\_a\_esp.items()
[('one','uno'), ('three', 'tres'), ('two', 'dos')]
```

La sintaxis nos proporciona informacin muy til acerca del tipo de datos. Los corchetes indican que es una lista. Los parntesis indican que los elementos de la lista son tuplas.

Si un mtodo acepta un argumento, usa la misma sintaxis que una llamada a una funcin. Por ejemplo, el mtodo `has_key` acepta una clave y devuelve verdadero (1) si la clave aparece en el diccionario:

```
>>> ing\_a\_esp.has_key('one')
1
>>> ing\_a\_esp.has_key('deux')
0
```

Si invocas un mtodo sin especificar un objeto, provocas un error. En este caso, el mensaje de error no es de mucha ayuda:

```
>>> has_key('one')
NameError: has_key
```

2.3. Aliasing and copying

Debes estar atento a los alias a causa de la mutabilidad de los diccionarios. Si dos variables se refieren al mismo objeto los cambios en una afectan a la otra.

Si quieres modificar un diccionario y mantener una copia del original, usa el mtodo `copy`. Por ejemplo, `opuestos` es un diccionario que contiene pares de opuestos:

```
>>> opuestos = {'arriba': 'abajo', 'derecho': 'torcido', 'verdadero': 'falso'}
>>> alias = opuestos
>>> copia = opuestos.copy()
```

`alias` y `opuestos` se refieren al mismo objeto; `copia` se refiere a una copia nueva del mismo diccionario. Si modificamos `alias`, `opuestos` tambin resulta cambiado:

```
>>> alias['derecho'] = 'sentado'
>>> opuestos['derecho']
'sentado'
```

Si modificamos `copia`, `opuestos` no vara:

```
>>> copia['derecho'] = 'privilegio'
>>> opuestos['derecho']
'sentado'
```

2.4. Matrices dispersas

En la Seccin **??** usamos una lista de listas para representar una matriz. Es una buena opcin para una matriz en la que la mayora de los valores es diferente de cero, pero piensa en una matriz como sta:

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \end{bmatrix}$$

La representacin de la lista contiene un montn de ceros:

```
matrix = [ [0,0,0,1,0],
           [0,0,0,0,0],
           [0,2,0,0,0],
```

```
            [0,0,0,0,0],
            [0,0,0,3,0] ]
```

Una posible alternativa es usar un diccionario. Como claves, podemos usar tuplas que contengan los nmeros de fila y columna. sta es la representacin de la misma matriz por medio de un diccionario:

```
matrix = {(0,3): 1, (2, 1): 2, (4, 3): 3}
```

Slo hay tres pares clave-valor, una para cada elemento de la matriz diferente de cero. Cada clave es una tupla, y cada valor es un entero.

Para acceder a un elemento de la matriz, podemos usar el operador `[]`:

```
matrix[0,3]
1
```

Fjate en que la sintaxis para la representacin por medio del diccionario no es la misma de la representacin por medio de la lista anidada. En lugar de dos ndices enteros, usamos un ndice que es una tupla de enteros.

Hay un porblema. Si apuntamos a un elemento que es cero, se produce un error porque en el diccionario no hay una entrada con esa clave:

```
>>> matrix[1,3]
KeyError: (1, 3)
```

El mtodo `get` soluciona este problema:

```
>>> matrix.get((0,3), 0)
1
```

El primer argumento es la clave; el segundo argumento es el valor que debe devolver `get` en caso de que la clave no est en el diccionario:

```
>>> matrix.get((1,3), 0)
0
```

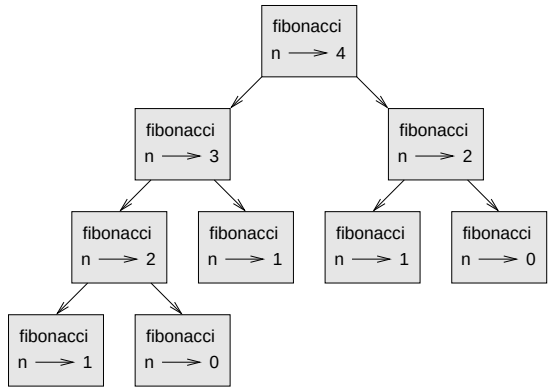
`get` mejora sensiblemente la semntica del acceso a una matriz dispersa. Lstima de sintaxis.

2.5. Pistas

Si estuviste jugando con la funcin `fibonacci` de la Seccin **??**, es posible que hayas notado que cuanto ms grande es el argumento que le das, ms tiempo le cuesta ejecutarse. Ms an, el tiempo de ejecucin aumenta muy rpidamente. En

nuestra mquina, `fibonacci(20)` acaba instantneamente, `fibonacci(30)` tarda ms o menos un segundo, y `fibonacci(40)` tarda una eternidad.

Para entender por qu, observa este **grfico de llamadas** de `fibonacci` con `n=4`:



Un grfico de llmadas muestra un conjunto de cajas de funcin con lneas que conectan cada caja con las cajas de las funciones a las que llama. En lo alto del grfico, `fibonacci` con `n=4` llama a `fibonacci` con `n=3` y `n=2`. A su vez, `fibonacci` con `n=3` llama a `fibonacci` con `n=2` y `n=1`. Y as sucesivamente.

Cuenta cuntas veces se llama a `fibonacci(0)` y `fibonacci(1)`. Es una solucin ineficaz al problema, y empeora mucho tal como crece el argumento.

Una buena solucin es llevar un registro de los valores que ya se han calculado almacenndolos en un diccionario. A un valor que ya ha sido calculado y almacenado para un uso posterior se le llama **pista**. Aqu hay una implementacin de `fibonacci` con pistas:

```
anteriores = {0:1, 1:1}

def fibonacci(n):
  if anteriores.has_key(n):
    return anteriores[n]
  else:
    nuevoValor = fibonacci(n-1) + fibonacci(n-2)
    anteriores[n] = nuevoValor
    return nuevoValor
```

El diccionario llamado `anteriores` mantiene un registro de los valores de Fibonacci que ya conocemos. El programa comienza con slo dos pares: 0 corresponde a 1 y 1 corresponde a 1.

Siempre que se llama a `fibonacci` comprueba si el diccionario contiene el resultado ya calculado. Si est ah, la funcin puede devolver el valor inmediatamente sin hacer ms llamadas recursivas. Si no, tiene que calcular el nuevo valor. El nuevo valor se aade al diccionario antes de que la funcin vuelva.

Con esta versin de `fibonacci`, nuestra mquina puede calcular `fibonacci(40)` en un abrir y cerrar de ojos. Pero cuando intentamos calcular `fibonacci(50)`, nos encontramos con otro problema:

```
>>> fibonacci(50)
OverflowError: integer addition
```

La respuesta, como vers en un momento, es 20.365.011.074. El problema es que este nmero es demasiado grande para caber en un entero de Python. Se **desborda**. Afortunadamente, hay una solucin fcil para este problema.

2.6. Enteros largos

Python proporciona un tipo llamado `long int` que puede manejar enteros de cualquier tamao. Hay dos formas de crear un valor `long int`. Una es escribir un entero con una *L* mayscula al final:

```
>>> type(1L)
<type 'long int'>
```

La otra es usar la funcin `long` para convertir un valor en `long int`. `long` acepta cualquier tipo numrico e incluso cadenas de dgitos:

```
>>> long(1)
1L
>>> long(3.9)
3L
>>> long('57')
57L
```

Todas las operaciones matemticas funcionan sobre los `long int`s, as que no tenemos que hacer mucho para adaptar `fibonacci`:

```
>>> previous = {0:1L, 1:1L}
>>> fibonacci(50)
20365011074L
```

Simplemente cambiando el contenido inicial de `anteriores` cambiamos el comportamiento de `fibonacci`. Los primeros dos nmeros de la secuencia son `long int`s, as que todos los nmeros subsiguientes lo sern tambin.

Como ejercicio, modifica `factorial` *de forma que produzca un* `long int` *como resultado.*

2.7. Contar letras

En el captulo **??** escribimos una funcin que contaba el nmero de apariciones de una letra en una cadena. Una versin ms genrica de este problema es crear un histograma de las letras de la cadena, o sea, cuntas veces aparece cada letra.

Ese histograma podra ser til es para comprimir un archivo de texto. Como las diferetnes letras aparecen con frecuencias distintas, podemos comprimir un archivo usando cdigos cortos para las letras ms habituales y cdigos ms largos para las que aparecen con menor frecuencia.

Los diccionarios facilitan una forma elegante de generar un histograma:

```
>>> cuentaLetras = {}
>>> for letra in "Mississippi":
...   cuentaLetras[letra] = cuentaLetras.get (letra, 0) + 1
...
>>> cuentaLetras
{'M': 1, 's': 4, 'p': 2, 'i': 4}
>>>
```

Inicialmente, tenemos un diccionario vaco. Para cada letra de la cadena, buscamos el recuento actual (posiblemente cero) y la incrementamos. Al final, el diccionario contiene pares de letras y sus frecuencias.

Puede ser ms atractivo mostrar el histograma en orden alfabtico. Podemos hacerlo con los mtodos `items` y `sort`:

```
>>> itemsLetras = cuentaLetras.items()
>>> itemsLetras.sort()
>>> print itemsLetras
[('M', 1), ('i', 4), ('p', 2), ('s', 4)]
```

Ya habas visto el mtodo `items`, pero `sort` es el primer mtodo aplicable a listas que hemos visto. Hay varios ms, como `append`, `extend`, y `reverse`. Consulta la documentacin de Python para ver los detalles.

2.8. Glosario

diccionario: Una coleccin de pares clave-valor que establece una correspondencia entre claves y valores. Las claves pueden ser de cualquier tipo inmutable, los valores pueden ser de cualquier tipo.

clave: Un valor que se usa para buscar una entrada en un diccionario.

par clave-valor: Uno de los elementos de un diccionario, tambin llamado .ªsociacin".

mtodo: Un tipo de funcin al que se llama con una sintaxis diferente y al que se invoca "sobreün objeto.

invocar: Llamar a un mtodo.

pista: Almacenamiento temporal de un valor precalculado para evitar clculos redundantes.

rebasamiento: Un resultado numrico que es demasiado grande para representarse en formato numrico.

Capítulo 3

Archivos y excepciones

Cuando un programa se est ejecutando, sus datos estn en la memoria. Cuando un programa termina, o se apaga el computador, los datos de la memoria desaparecen. Para almacenar los datos de forma permanente debes ponerlos en un **archivo**. Normalmente los archivos se guardan en un disco duro, disquete o CD-ROM.

Cuando hay un gran nmero de archivos, suelen estar organizados en **directorios** (tambin llamados çarpetas"). Cada archivo se identifica con un nombre nico, o una combinacin de nombre de archivo y nombre de directorio.

Leyendo y escribiendo archivos, los programas pueden intercambiar informacin entre ellos y generar formatos imprimibles como PDF.

Trabajar con archivos se parece mucho a trabajar con libros. Para usar un libro, tienes que abrirlo. Cuando has terminado, tienes que cerrarlo. Mientras el libro est abierto, puedes escribir en l o leer de l. En cualquier caso, sbes en qu lugar del libro te encuentras. Casi siempre lees el libro segn su orden natural, pero tambin puedes ir saltando de pgina en pgina.

Todo esto sirve tambin para los archivos. Para abrir un archivo, especificas su nombre e indicas si quieres leer o escribir.

La apertura de un archivo crea un objeto archivo. En este ejemplo, la variable `f` apunta al nuevo objeto archivo.

```
>>> f = open("test.dat","w")
>>> print f
<open file 'test.dat', mode 'w' at fe820>
```

La funcin open toma dos argumentos. El primero es el nombre del archivo, y el segundo es el modo. El modo `"w"` significa que lo estamos abriendo para escribir.

Si no hay un archivo llamado `test.dat` se crear. Si ya hay uno, el archivo que estamos escribiendo lo reemplazar.

Al imprimir el objeto archivo, vemos el nombre del archivo, el modo y la localizacin del objeto.

Para meter datos en el archivo invocamos al mtodo `write` sobre el objeto archivo:

```
>>> f.write("Ya es hora")
>>> f.write("de cerrar el archivo")
```

El cierre del archivo le dice al sistema que hemos terminado de escribir y deja el archivo listo para leer:

```
>>> f.close()
```

Ya podemos abrir el archivo de nuevo, esta vez para lectura, y poner su contenido en una cadena. Esta vez el argumento de modo es `"r"` para lectura:

```
>>> f = open("test.dat","r")
```

Si intentamos abrir un archivo que no existe, recibimos un mensaje de error:

```
>>> f = open("test.cat","r")
IOError: [Errno 2] No such file or directory: 'test.cat'
```

Como era de esperar, el mtodo `read` lee datos del archivo. Sin artumentos, lee el archivo completo:

```
>>> text = f.read()
>>> print text
Ya es horade cerrar el archivo
```

No hay un espacion entre "horaτ "de"porque no escribimos un espacio entre las cadenas.

`read` tambin puede aceptar un argumento que le indica cuntos caracteres leer:

```
>>> f = open("test.dat","r")
>>> print f.read(7)
Ya es h
```

Si no quedan suficientes caracteres en el archivo, `read` devuelve los que haya. Cuando llegamos al final del archivo, `read` devuelve una cadena vaca:

```
>>> print f.read(1000006)
orade cerrar el archivo
>>> print f.read()

>>>
```

La siguiente funcin copia un archivo, leyendo y escribiendo los caracteres de cincuenta en cincuenta. El primer argumento es el nombre del archivo original; el segundo es el nombre del archivo nuevo:

```
def copiaArchivo(archViejo, archNuevo):
  f1 = open(archViejo, "r")
  f2 = open(archNuevo, "w")
  while 1:
    texto = f1.read(50)
    if texto == "":
      break
    f2.write(texto)
  f1.close()
  f2.close()
  return
```

La sentencia `break` es nueva. Su ejecucin interrumpe el bucle; el flujo de la ejecucin pasa a la primera sentencia tras el bucle.

En este ejmplo, el bucle `while` es infinito porque el valor `1` siempre es verdadero. La *nica* forma de salir del bucle es ejecutar `break`, lo que sucede cuando `texto` es una cadena vaca, lo que sucede cuando llegamos al final del archivo.

3.1. Archivos de texto

Un **archivo de texto** es un archivo que contiene caracteres imprimibles y espacios organizados en lneas separadas por caracteres de salto de lnea. Como Python est diseado especficamente para procesar archivos de texto, proporciona mtodos que facilitan la tarea.

Para hacer una demostracin, crearemos un archivo de texto con tres lneas de texto separadas por saltos de lnea:

```
>>> f = open("test.dat","w")
>>> f.write("lnea uno\nlnea dos\nlnea tres\n")
>>> f.close()
```

El mtodo `readline` lee todos los caracteres hasta e incluyendo el siguiente salto de lnea:

```
>>> f = open("test.dat","r")
>>> print f.readline()
lnea uno

>>>
```

`readlines` devuelve todas las lneas que queden como una lista de cadenas:

```
>>> print f.readlines()
['lnea dos\012', 'lnea tres\012']
```

En este caso, la salida est en forma de lista, lo que significa que las cadenas aparecen con comillas y el carcter de salto de lnea aparece como la secuencia de escape
012.

Al final del archivo, `readline` devuelve una cadena vaca y `readlines` devuelve una lista vaca:

```
>>> print f.readline()

>>> print f.readlines()
[]
```

Lo que sigue es un ejemplo de un programa de proceso de lneas. `filtraArchivo` hace una copia de `archViejo`, omitiendo las lneas que comienzan por `#`:

```
def filtraArchivo(archViejo, archNuevo):
  f1 = open(archViejo, "r")
  f2 = open(archNuevo, "w")
  while 1:
    texto = f1.readline()
    if texto == "":
      break
    if texto[0] == '#':
      continue
    f2.write(texto)
  f1.close()
  f2.close()
  return
```

La sentencia `continue` termina la iteracin actual del bucle, pero sigue haciendo bucles. El flujo de ejecucin pasa al principio del bucle, comprueba la condicin y contina consecuentemente.

As, si `texto` es una cadena vaca, el bucle termina. Si el primer carcter de `texto` es una almohadilla, el flujo de ejecucin va al principio del bucle. Slo si ambas condiciones fallan copiamos `texto` en el archivo nuevo.

3.2. Escribir variables

El argumento de `write` debe ser una cadena, as que si queremos poner otros valores en un archivo, tenemos que convertirlos ante en cadenas. La forma ms fcil de hacerlo es con la funcin `str`:

```
>>> x = 52
>>> f.write (str(x))
```

Una alternativa es usar el **operador de formato** `%`. Cuando aplica a enteros, `%` es el operador de mdulo. Pero cuando el primer operando es una cadena, `%` es el operador de formato.

El primer operando es la **cadena de formato**, y el segundo operando es una tupla de expresiones. El resultado es una cadena que contiene los valores de las expresiones, formateados de acuerdo a la cadena de formato.

A modo de ejemplo simple, la **secuencia de formato** `"\%d"` significa que la primera expresin de la tupla debera formatearse como un entero. Aqu la letra *d* quiere decir "decimal":

```
>>> motos = 52
>>> "%d" % motos
'52'
```

El resultado es la cadena `'52'`, que no debe confundirse con el valor entero `52`.

Una secuencia de formato puede aparecer en cualquier lugar de la cadena de formato, de modo que podemos incrustar un valor en una frase:

```
>>> motos = 52
>>> "En julio vendimos %d motos." % motos
'En julio vendimo 52 motos.'
```

La secuencia de formato `"\%f"` formatea el siguiente elemento de la tupla como un nmero en coma flotante, y `"\%s"` formatea el siguiente elemento como una cadena:

```
>>> "En %d das ingresamos %f millones de %s." % (34,6.1,'dlares')
'En 34 das ingresamose 6.100000 miliones de dlares.'
```

Por defecto, el formato de coma flotante imprime seis decimales.

El nmero de expresiones en la tupla tiene que coincidir con el nmero de secuencias de formato de la cadena. Igualmente, los tipos de las expresiones deben coincidir con las secuencias de formato:

```
>>> "%d %d %d" % (1,2)
TypeError: not enough arguments for format string
>>> "%d" % 'dlares'
TypeError: illegal argument type for built-in operation
```

En el primer ejemplo, no hay suficientes expresiones; en el segundo, la expresin es de un tipo incorrecto.

Para tener ms control sobre el formato de los nmeros, podemos detallar el nmero de dgitos como parte de la secuencia de formato:

```
>>> "%6d" % 62
'    62'
>>> "%12f" % 6.1
'    6.100000'
```

El nmero tras el signo de porcentaje es el nmero mnimo de espacios que ocupar el nmero. Si el valor necesita menos dgitos, se aaden espacios en blanco delante del nmero. Si el nmero de espacios es negativo, se aaden los espacios tras el nmero:

```
>>> "%-6d" % 62
'62    '
```

Tambin podemos especificar el nmero de decimales para los nmeros en coma flotante:

```
>>> "%12.2f" % 6.1
'        6.10'
```

En este ejemplo, el resultado ocupa doce espacios e incluye dos dgitos tras la coma. Este formato es til para imprimir cantidades de dinero con las comas alineadas.

Imagina, por ejemplo, un diccionario que contiene los nombres de los estudiantes como clave y las tarifas horarias como valores. He aqu una funcin que imprime el contenido del diccionario como de un informe formateado:

```
def informe (tarifas) :
  estudiantes = tarifas.keys()
  estudiantes.sort()
```

```
  for estudiante in estudiantes :
    print "%-20s %12.02f" % (estudiante, tarifas[estudiante])
```

Para probar lafuncin, crearemos un pequeo diccionario e imprimiremos el contenido:

```
>>> tarifas = {'mara': 6.23, 'jos': 5.45, 'jess': 4.25}
>>> informe (tarifas)
jos                          5.45
jess                         4.25
mara                         6.23
```

Controlando la anchura de cada valor nos aseguramos de que las columnas van a quedar alineadas, siempre que los nombre tengan menos de veintin caracteres y las tarifas sean menos de mil millones la hora.

3.3. Directorios

Cuando creas un archivo nuevo abrindolo y escribiendo, el nuevo archivo va al directorio en uso (aqul en el que etuvieses al ejecutar el programa). Del mismo modo, cuando abres un archivo para leerlo, Python lo busca en el directorio en uso.

Si queires abrir un archivo de cualquier otro sitio, tienes que especificar la **ruta** del archivo, que es el nombre del direcotrio (o carpeta) donde se encuentra ste:

```
>>>   f = open("/usr/share/dict/words","r")
>>>   print f.readline()
Aarhus
```

Este ejemplo abre un archivo llamado `words` que est en un directorio llamado `dict`, que est en `share`, que est en `usr`, que est en el directorio de nivel superior del sistema, llamado `/`.

No puedes usar `/` como parte del nombre de un archivo; est reservado como delimitador entre nombres de archivo y directorios.

El archivo `/usr/share/dict/words` contiene una lista de palabras en orden alfabtico, la primera de las cuales es el nombre de una universidad danesa.

3.4. Encurtido

Para poner valores en un archivo, debes convertirlos en cadenas. Ya has visto cmo hacerlo con `str`:

```
>>> f.write (str(12.3))
>>> f.write (str([1,2,3]))
```

El problema es que cuando vuelves a leer el valor, obtienes una cadena. Has perdido la informacin del tipo de dato original. En realidad, no puedes distinguir dnde termina un valor y comienza el siguiente:

```
>>>   f.readline()
'12.3[1, 2, 3]'
```

La solucin es el **encurtido**, llamado as porque çonserva"estructuras de datos. El mdulo `pickle` contiene las rdenes necesarias. Para usarlo, importa `pickle` y luego abre el archivo de la forma habitual:

```
>>> import pickle
>>> f = open("test.pck","w")
```

Para almacenar una estructura de datos, usa el mtido`dump` y luego cierra el archivo de la forma habitual:

```
>>> pickle.dump(12.3, f)
>>> pickle.dump([1,2,3], f)
>>> f.close()
```

Ahora podemos abrir el archivo para leer y cargar las estructuras de datos que volcamos ah:

```
>>> f = open("test.pck","r")
>>> x = pickle.load(f)
>>> x
12.3
>>> type(x)
<type 'float'>
>>> y = pickle.load(f)
>>> y
[1, 2, 3]
>>> type(y)
<type 'list'>
```

Cada vez que invocamos `load` obtenemos un valor del archivo, completo con su tipo original.

3.5. Excepciones

Siempre que ocurre un error en tiempo de ejecucin, crea una **exception**. Normalmente el programa se para y Pythton presenta un mensaje de error.

Por ejemplo, la divisin por cero crea una excepcin:

```
>>> print 55/0
ZeroDivisionError: integer division or modulo
```

Un elemento no existente en una lista hace lo mismo:

```
>>> a = []
>>> print a[5]
IndexError: list index out of range
```

O el acceso a una clave que no est en el diccionario:

```
>>> b = {}
>>> print b['qu']
KeyError: qu
```

En cada caso, el mensaje de error tiene dos partes: el tipo de error antes de los dos puntos y detalles sobre el error depus de los dos puntos. Normalmente Python tambin imprime una traza de dnde se encontraba el programa, pero la hemos omitido en los ejemplos.

A veces queremos realizar una operacin que podra provocar una excepcin, pero no queremos que se pare el programa. Podemos **manejar** la excepcin usando las sentencias `try` y `except`.

Por ejemplo, podemos preguntar al usuario por el nombre de un archivo y luego intentar abrirlo. Si el archivo no existe, no queremos que el programa casque; queremos manejar la excepcin.

```
nombreArch = raw_input('Introduce un nombre de archivo: ')
try:
  f = open (nombreArch, "r")
except:
  print 'No hay ningn archivo que se llame', nombreArch
```

La sentencia `try` ejecuta las sentencias del primer bloque. Si no se produce ninguna excepcin, pasa por alto la sentencia `except`. Si ocurre cualquier excepcin, ejecuta las sentencias de la rama `except` y despus continua.

Podemos encapsular esta capacidad en una funcin: `existe` acepta un nombre de archivo y devuelve verdadero si el archivo existe y falso si no:

```
def existe(nombreArch):
  try:
    f = open(nombreArch)
    f.close()
```

```
    return 1
  except:
    return 0
```

Puedes usar mltiples bloques `except` para manejar diferentes tipos de excepciones. El *Manual de Referencia de Python* contiene los detalles.

Si tu programa detecta una condicin de error, puedes hacer que lance (**raise** en ingls) una excepcin. Aqu tienes un ejemplo que acepta una entrada del usuario y comprueba si es 17. Suponiendo que 17 no es una entrada vlida por cualquier razn, lanzamos una excepcin.

```
def tomaNumero () :                  # Recuerda, los acentos estn prohibidos
  x = input ('Elige un nmero: ')   # en los nombres de funciones y variables!
  if x == 17 :
    raise 'ErrorNmeroMalo', '17 es un mal nmero'
  return x
```

La sentencia `raise` acepta dos argumentos: el tipo de excepcin e informacin especfica acerca del error. `ErrorNmeroMalo` es un nuevo tipo de excepcin que hemos inventado para esta aplicacin.

Si la funcin llamada `tomaNumero` maneja el error, el programa puede continuar; en caso contrario, Python imprime el mensaje de error y sale:

```
>>> tomaNumero ()
Elige un nmero: 17
ErrorNmeroMalo: 17 es un mal nmero
```

El mensaje de error incluye el tipo de excepcin y la informacin adicional que proporcionaste.

> *Como ejercicio, escribe una funcin que use* `tomaNumero` *para leer un nmero del teclado y que maneje la excepcin* `ErrorNmeroMalo`.

3.6. Glosario

archivo: Una entidad con nombre, normalmente almacenada en un disco duro, disquete o CD-ROM, que contiene una secuencia de caracteres.

directorio: Una coleccin, con nombre, de archivos, tambin llamado carpeta.

ruta: Una secuencia de nombres de directorio que especifica la localizacin exacta de un archivo.

archivo de texto: Un archivo que contiene caracteres imprimibles organizados en lneas separadas por caracteres de salto de lnea.

sentencia break: Una sentencia que provoca que el flujo de ejecucin salga de un bucle.

sentencia continue: Una sentencia que provoca que termine la iteracin actual de un bucle. El flujo de la ejecucin va al principio del bucle, evala la condicin, y procede en consecuencia.

operador de formato: El operador `%` toma una cadena de formato y una tupla de expresiones y entrega una cadena que incluye las expresiones, formateadas de acuerdo con la cadena de formato.

cadena de formato: Una cadena que contiene caracteres imprimibles y secuencias de formato que indican cmo formatear valores.

secuencia de formato: Una secuencia de caracteres que comienza con `%` e indica cmo formatear un valor.

encurtir: Escribir el valor de un dato en un archivo junto con la informacin sobre su tipo de forma que pueda ser reconstituido ms tarde.

excepcin: Un error que ocurre en tiempo de ejecucin.

manejar: Impedir que una excepcin detenga un programa utilizando las sentencias `try` y `except`.

lanzar: Sealar una excepcin usando la sentencia `raise`.

Capítulo 4

Clases y objetos

4.1. Tipos compuestos definidos por el usuario

Una vez utilizados algunos de los tipos internos de Python, estamos listos para crear un tipo definido por el usuario: el `Punto`.

Piensa en el concepto de un punto matemtico. En dos dimensiones, un punto es dos nmeros (coordenadas) que se tratan colectivamente como un solo objeto. En notacin matemtica, los puntos suelen escribirse entre parntesis con una coma separando las coordenadas. Por ejemplo, $(0, 0)$ representa el origen, y (x, y) representa el punto x unidades a la derecha e y unidades hacia arriba desde el origen.

Una forma natural de representar un punto en Python es con dos valores en coma flotante. La cuestin es, entonces, cmo agrupar esos dos valores en un objeto compuesto. La solucin rpida y burda es utilizar una lista o tupla, y para algunas aplicaciones esa podra ser la mejor opcin.

Una alternativa es que el usuario defina un nuevo tipo compuesto, tambin llamado una **class**. Esta aproximacin exige un poco ms de esfuerzo, pero tiene sus ventajas que pronto se harn evidentes.

Una definicin de clase se parece a esto:

```
class Punto:
  pass
```

Las definiciones de clase pueden aparecer en cualquier lugar de un programa, pero normalmente estn al principio (tras las sentencias `import`). Las reglas sintcticas de la definicin de clases son las mismas que para cualesquiera otras sentencias compuestas. (ver la Seccin **??**).

Esta definicin crea una nueva clase llamada `Punto`. La sentencia **pass** no tiene efectos; slo es necesaria porque una sentencia compuesta debe tener algo en su cuerpo.

Al crear la clase `Punto` hemos creado un nuevo tipo, que tambin se llama `Punto`. Los miembros de este tipo se llaman **instancias** del tipo u **objetos**. La creacin de una nueva instancia se llama **instanciacin**. Para instanciar un objeto `Punto` ejecutamos una funcin que se llama (lo has adivinado) `Punto`:

```
limpio = Punto()
```

A la variable `limpio` se le asigna una referencia a un nuevo objeto `Punto`. A una funcin como `Punto` que crea un objeto nuevo se le llama **constructor**.

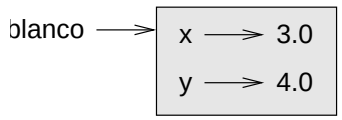
4.2. Atributos

Podemos aadir nuevos datos a una instancia utilizando la notacin de punto:

```
>>> limpio.x = 3.0
>>> limpio.y = 4.0
```

Esta sintaxis es similar a la sintaxis para seleccionar una variable de un mdulo, como `math.pi` o `string.uppercase`. En este caso, sin embargo, estamos seleccionando un dato de una instancia. Estos temes con nombre se llaman **atributos**.

El diagrama de estados que sigue muestra el resultado de esas asignaciones:



La variable `limpio` apunta a un objeto Punto, que contiene dos atributos. Cada atributo apunta a un nmero en coma flotante.

Podemos leer el valor de un atributo utilizando la misma sintaxis:

```
>>> print limpio.y
4.0
>>> x = limpio.x
>>> print x
3.0
```

La expresin `limpio.x` significa, “ve al objeto al que apunta `limpio` y toma el valor de `x`.” En este caso, asignamos ese valor a una variable llamada `x`. No hay

conflicte entre la variable `x` y el atributo `x`. El propsito de la notacin de punto es identificar de forma inequvoca a qu variable te refieres.

Puedes usar la notacin de punto como parte de cualquier expresin. As, las sentencias que siguen son correctas:

```
print '(' + str(limpio.x) + ', ' + str(limpio.y) + ')'
distanciaAlCuadrado = limpio.x * limpio.x + limpio.y * limpio.y
```

La primera lnea presenta `(3.0, 4.0)`; la segunda lnea calcula el valor 25.0.

Puede tentarte imprimir el propio valor de `limpio`:

```
>>> print limpio
<__main__.Point instance at 80f8e70>
```

El resultado indica que `limpio` es una instancia de la clase `Punto` que se defini en `__main__`. `80f8e70` es el identificador nico de este objeto, escrito en hexadecimal. Probablemente no es esta la manera ms clara de mostrar un objeto `Punto`. En breve vers cmo cambiarlo.

> *Como ejercicio, crea e imprime un objeto* `Punto` *y luego usa* `id` *para imprimir el identificador nico del objeto. Traduce la el nmero hexadecimal a decimal y asegrate de que coinciden.*

4.3. Instancias como parmetro

Puedes pasar una instancia como parmetro de la forma habitual. Por ejemplo:

```
def imprimePunto(p):
  print '(' + str(p.x) + ', ' + str(p.y) + ')'
```

`imprimePunto` acepta un punto como argumento y lo muestra en formato estndar. Si llamas a `imprimePunto(limpio)`, el resultado es `(3.0, 4.0)`.

> *Como ejercicio, reescribe la funcin* `distancia` *de la Seccin* **??** *de forma que acepte dos* `Punto`*s como parmetros en lugar de cuatro nmeros.*

4.4. Mismidad

El significado de la palabra "mismo"parece totalmente claro hasta que te paras un poco a pensarlo, y entonces te das cuenta de que hay algo ms de lo que suponas.

Por ejemplo, si dices "Pepe y yo tenemos la misma moto", lo que quieres decir es que su moto y la tuya son de la misma marca y modelo, pero que son dos motos distintas. Si dices "Pepe y yo tenemos la misma madre", quieres decir que su madre y la tuya son la misma persona[1]. As que la idea de "identidad"es diferente segn el contexto.

Cuando hablas de objetos, hay una ambigedad parecida. Por ejemplo, si dos `Puntos` son el mismo, significa que contienen los mismos datos (coordenadas) o que son de verdad el mismo objeto?

Para averiguar si dos referencias se refieren al mismo objeto, utiliza el operador `==`. Por ejemplo:

```
>>> p1 = Punto()
>>> p1.x = 3
>>> p1.y = 4
>>> p2 = Puto()
>>> p2.x = 3
>>> p2.y = 4
>>> p1 == p2
0
```

Aunque `p1` y `p2` contienen las mismas coordenadas, no son el mismo objeto. Si asignamos `p1` a `p2`, las dos variables son alias del mismo objeto:

```
>>> p2 = p1
>>> p1 == p2
1
```

Este tipo de igualdad se llama **igualdad superficial** porque slo compara las referencias, pero no el contenido de los objetos.

Para comparar los contenidos de los objetos (**igualdad profunda**) podemos escribir una funcin llamada `mismoPunto`:

```
def mismoPunto(p1, p2) :
  return (p1.x == p2.x) and (p1.y == p2.y)
```

Si ahora creamos dos objetos diferentes que contienen los mismos datos podremos usar `mismoPunto` para averiguar si representan el mismo punto:

```
>>> p1 = Punto()
>>> p1.x = 3
```

[1] No todas las lenguas tienen el mismo problema. Por ejemplo, el alemn tienen palabras diferentes para los diferentes tipos de identidad. "Misma moto"en este contexto sera "gleiche Motorradτ "misma madre"sera "selbe Mutter".

```
>>> p1.y = 4
>>> p2 = Punto()
>>> p2.x = 3
>>> p2.y = 4
>>> mismoPunto(p1, p2)
1
```

Por supuesto, si las dos variables apuntan al mismo objeto `mismoPunto` devuelve verdadero.

4.5. Rectngulos

Digamos que queremos una clase que represente un rectngulo. La pregunta es, qu informacin tenemos que proporcionar para definir un rectngulo? Para simplificar las cosas, supongamos que el rectngulo est orientado vertical u horizontalmente, nunca en diagonal.

Tenemos varias posibilidades: podemos sealar el centro del rectngulo (dos coordenadas) y su tamao (anchura y altura); o podemos sealar una de las esquinas y el tamao; o podemos sealar dos esquinas opuestas. Un modo convencional es sealar la esquina superior izquierda del rectngulo y el tamao.

De nuevo, definiremos una nueva clase:

```
class Rectangulo: # Prohibidos los acentos fuera de las cadenas!
  pass
```

Y la instanciaremos:

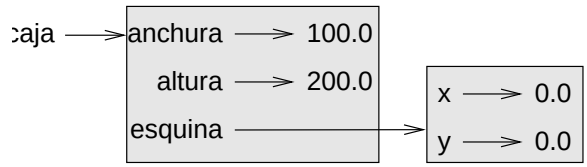
```
caja = Rectangulo()
caja.anchura = 100.0
caja.altura = 200.0
```

Este cdigo crea un nuevo objeto `Rectangulo` con dos atributos en coma flotante. Para sealar la esquina superior izquierda podemos incrustar un objeto dentro de otro!

```
caja.esquina = Punto()
caja.esquina.x = 0.0;
caja.esquina.y = 0.0;
```

El operador punto compone. La expresin `caja.esquina.x` significa "ve al objeto al que se refiere `caja` y selecciona el atributo llamado `esquina`; entonces ve a ese objeto y selecciona el atributo llamado `x`."

La figura muestra el estado de este objeto:



4.6. Instancias como valores de retorno

Las funciones pueden devolver instancias. Por ejemplo, `encuentraCentro` acepta un `Rectangulo` como argumento y devuelve un `Punto` que contiene las coordenadas del centro del `Rectangulo`:

```
def encuentraCentro(caja):
  p = Punto()
  p.x = caja.esquina.x + caja.anchura/2.0
  p.y = caja.esquina.y + caja.altura/2.0
  return p
```

Para llamar a esta fucnin, pasa `caja` como argumento y asigna el resultado a una variable:

```
>>> centro = encuentraCentro(caja)
>>> imprimePunto(centro)
(50.0, 100.0)
```

4.7. Los objetos son mudables

Podemos cambiar el estado de un objeto efectuando una asignacin sobre uno de sus atributos. Por ejemplo, para cambiar el tamao de un rectngulo sin cambiar su posicin, podemos cambiar los valores de `anchura` y `altura`:

```
caja.anchura = caja.anchura + 50
caja.altura = caja.altura + 100
```

Podemos encapsular este cdigo en un mtodo y generalizarlo para agrandar el rectngulo en cualquier cantidad:

```
def agrandarRect(caja, danchura, daltura) :
  caja.anchura = caja.anchura + danchura
  caja.altura = caja.altura + daltura
```

Las variables `danchura` y `daltura` indican cunto debe agrandarse el rectngulo en cada direccin. Invocar este mtodo tiene el efecto de modificar el `Rectangulo` que se pasa como argumento.

Por ejemplo, podemos crear un nuevo `Rectangulo` llamado `bob` y pasrselo a `agrandaRect`:

```
>>> bob = Rectangulo()
>>> bob.anchura = 100.0
>>> bob.altura = 200.0
>>> bob.esquina = Punto()
>>> bob.esquina.x = 0.0;
>>> bob.esquina.y = 0.0;
>>> agrandaRect(bob, 50, 100)
```

Mientras `agrandaRect` se est ejecutando, el parmetro `caja` es un alias de `bob`. Cualquier cambio que hagas a `caja` afectar tambin a `bob`.

> *A modo de ejercicio, escribe una funcin llamada* `mueveRect` *que tome un* `Rectangulo` *y dos parmetros llamados* `dx` *y* `dy`*. Tiene que cambiar la posicin del rectngulo aadiendo* `dx` *a la coordenada* `x` *de* `esquina` *y aadiendo* `dy` *a la coordenada* `y` *de* `esquina`*.*

4.8. Copiado

El uso de alias puede hacer que un programa sea difcil de leer, porque los cambios hechos en un lugar pueden tener efectos inesperados en otro lugar. Es difcil estar al tanto de todas las variables a las que puede apuntar un objeto dado.

Copiar un objeto es, muchas veces, una alternativa a la creacin de un alias. El mdulo `copy` contiene una funcin llamada `copy` que puede duplicar cualquier objeto:

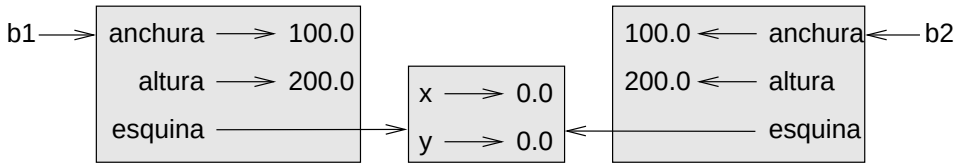
```
>>> import copy
>>> p1 = Punto()
>>> p1.x = 3
>>> p1.y = 4
>>> p2 = copy.copy(p1)
>>> p1 == p2
0
>>> mismoPunto(p1, p2)
1
```

Una vez que hemos importado el mdulo `copy`, podemos usar el mtodo `copy` para hacer un nuevo `Punto`. `p1` y `p2` no son el mismo punto, pero contienen los mismos datos.

Para copiar un objeto simple como un `Punto`, que no contiene objetos incrustados, `copy` es suficiente. Esto se llama **copiado superficial**.

Para algo como un `Rectangulo`, que contiene una referencia a un `Punto`, `copy` no lo hace del todo bien. Copia la referencia al objeto `Punto`, de modo que tanto el `Rectangulo` viejo como el nuevo apuntan a un nico `Punto`.

Si creamos una caja, `b1`, de la forma habitual y entonces hacemos una copia, `b2`, usando `copy`, el diagrama de estados resultante se ve as:



Es casi seguro que esto no es lo que queremos. En este caso, la invocacin de `agrandaRect` sobre uno de los `Rectangulo`s no afectara al otro, pero la invocacin de `mueveRect` sobre cualquiera afectaria a ambos! Este comportamiento es confuso y propicia los errores.

Afortunadamente, el mdulo `copy` contiene un mtodo llamado `deepcopy` que copia no slo el objeto sino tambin cualesquiera objetos incrustados. No te sorprender saber que esta operacin se llama **copia profunda** (deep copy).

```
>>> b2 = copy.deepcopy(b1)
```

Ahora `b1` y `b2` son objetos totalmente independientes.

Podemos usar `deepcopy` para reescribir `agrandaRect` de modo que en lugar de modificar un `Rectangulo` existente, cree un nuevo `Rectangulo` que tiene la misma localizacin que el viejo pero nuevas dimensiones:

```
def agrandaRect(caja, danchura, daltura) :
  import copy
  nuevaCaja = copy.deepcopy(caja)
  nuevaCaja.anchura = neuvaCaja.anchura + danchura
  nuevaCaja.altura = nuevaCaja.altura + daltura
  return nuevaCaja
```

Como ejercicio, resscribe `mueveRect` *de modo que cree y devuelva un nuevo* `Rectangulo` *en lugar de modificar el viejo.*

4.9. Glosario

clase: Un tipo compuesto definido por el usuario. Tambin se puede pensar en una clase como una plantilla para los objetos que son instancias de la misma.

instanciar: Crear una instancia de una clase.

instancia: Un objeto que pertenece a una clase.

objeto: Un tipo de dato compuesto que suele usarse para representar una cosa o concepto del mundo real.

constructor: Un mtodo usado para crear nuevos objetos.

atributo: Uno de los elementos de datos con nombre que constituyen una instancia.

igualdad superficial: Igualdad de referencias, o dos referencias que apuntan al mismo objeto.

igualdad profunda: Igualdad de valores, o dos referencias que apuntan a objetos que tienen el mismo valor.

copia superficial: Copiar el contenido de un objeto, incluyendo cualquier referencia a objetos incrustados; implementada por la funcin `copy` del mdulo `copy`.

copia profunda: Copiar el contenido de un objeto as como cualesquiera objetos incrustados, y los incrustados en estos, y as; implementada por la funcin `deepcopy` del mdulo `copy`.

Índice alfabético