Python para todos

Explorando la información con Python 3

Charles R. Severance

Créditos

Soporte Editorial: Elliott Hauser, Sue Blumenberg
Diseño de portada: Aimee Andrion

Historial de impresión

- 05-Jul-2016 Primera versión completa de Python 3.0
- 20-Dic-2015 Borrador inicial de conversión a Python 3.0

Detalles de Copyright

Copyright ~2009- Charles Severance.

Este trabajo está registrado bajo una Licencia Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License. Este licencia está disponible en

http://creativecommons.org/licenses/by-nc-sa/3.0/

Puedes ver lo que el autor considera usos comerciales y no-comerciales de este material, así como las exenciones de licencia en el Apéndice titulado “Detalles de Copyright”.

Prólogo

Remezclando un Libro Libre

Se suele decir de los académicos deben "publicar o perecer" continuamente, de modo que es bastante normal que siempre quieran empezar algo desde cero, para que sea su propia y flamante creación. Este libro es un experimento, ya que no parte desde cero, sino que en vez de eso "remezcla" el libro titulado *Think Python: How to Think Like a Computer Scientist* (Piensa en Python: Cómo pensar como un científico de la computación), escrito por Allen B. Bowney, Jeff Elkner, y otros.

En Diciembre de 2009, yo me estaba preparando para enseñar *SI502 - Programación en Red* en la Universidad de Michigan por quinto semestre consecutivo, y decidí que ya era hora de escribir un libro de texto sobre Python que se centrase en la exploración de datos en lugar de en explicar algoritmos y abstracciones. Mi objetivo en SI502 es enseñar a la gente habilidades permanentes para el manejo de datos usando Python. Pocos de mis estudiantes pretenden llegar a ser programadores de computadoras profesionales. En vez de eso, quieren ser bibliotecarios, gerentes, abogados, biólogos, economistas, etc., que tal vez quieran aplicar el uso de la tecnología en sus respectivos campos.

Parecía que no podría encontrar el libro perfecto para mi curso, que estuviera orientado al manejo de datos en Python, de modo que decidí empezar a escribirlo por mi mismo. Por suerte, en una reunión de profesores tres semanas antes de las vacaciones, que era la fecha en que tenía planeado empezar a escribir mi libro desde cero, el Dr. Atul Prakash me mostró el libro *Think Python* (Piensa en Python), que él había utilizado para impartir su curso de Python ese semestre. Se trata de un texto de Ciencias de la Computación bien escrito, con un enfoque breve, explicaciones directas y fácil de aprender.

La estructura principal del libro se ha cambiado, para empezar a realizar problemas de análisis de datos lo antes posible, y para tener una serie de ejemplos funcionales y de ejercicios sobre el análisis de datos desde el principio.

Los capítulos 2-10 son similares a los del libro *Think Python*, pero ha habido cambios importantes. Los ejemplos orientados a números y los ejercicios se han reemplazado por otros orientados a datos. Los temas se presentan en el orden necesario para ir creando soluciones de análisis de datos cuya complejidad aumente progresivamente. Algunos temas como `try` y `except` (manejo de excepciones) se han adelantado, y se presentan como parte del capítulo de los condicionales. Las funciones se tratan muy por encima hasta que son necesarias para manejar programas complejos, en lugar de introducirlas como abstracción en las primeras lecciones. Casi todas las funciones definidas por el usuario se han eliminado del código de los ejemplos y de los ejercicios excepto en el capítulo 4. La palabra "recursión"[1] no aparece en todo el libro.

Todo el contenido del capítulo 1 y del 11 al 16 es nuevo, centrado en aplicaciones para el mundo real y en ejemplos simples del uso de Python para el análisis de datos, incluyendo expresiones regulares para búsqueda y análisis, automatización de tareas en la computadora, descarga de datos a través de la red, escaneo de páginas web para recuperar datos, programación orientada a objetos, uso de servicios

[1] Excepto, por supuesto, en esa línea.

web, análisis de datos en formato XML y JSON, creación y uso de bases de datos usando el Lenguaje de Consultas Estructurado (SQL), y la visualización de datos.

El objetivo final de todos estos cambios es variar la orientación, desde una dirigida a las Ciencias de la Computación hacia otra puramente informática, que trate sólo temas adecuados para una clase de tecnología para principiantes, que puedan resultarles útiles incluso si eligen no ser programadores profesionales.

Los estudiantes que encuentren este libro interesante y quieran ir más allá, deberían echar un vistazo al libro *Think Python* de Allen B. Downey's. Como ambos libros comparten un montón de materia, los estudiantes adquirirán rápidamente habilidades en las áreas adicionales de la programación técnica y pensamiento algorítmico que se tratan en *Think Python*. Y dado que ambos libros comparten un estilo de escritura similar, deberían ser capaces de avanzar rápidamente a través del contenido de *Think Python* con un esfuerzo mínimo.

Como propietario del copyright de *Think Python*, Allen me ha dado permiso para cambiar la licencia del contenido de su libro que se utiliza en éste, y que originalmente poseía una *GNU Free Documentation License* a otra más actual, Creative Commons Attribution — Share Alike license. Así se sigue una tendencia general en las licencias de documentación abierta, que están pasando desde la GFDL a la CC-BY-SA (por ejemplo, Wikipedia). El uso de la licencia CC-BY-SA mantiene la arraigada tradición *copyleft* del libro, a la vez que hacen más sencillo para los autores nuevos la reutilización de ese material a su conveniencia.

Personalmente creo que este libro sirve como ejemplo de por qué los contenidos libres son tan importantes para el futuro de la educación, y quiero agradecer a Allen B. Downey y a la *Cambridge University Press* por su amplitud de miras a la hora de distribuir el libro bajo un copyright abierto. Espero que se sientan satisfechos con el resultado de mis esfuerzos y deseo que tú como lector también te sientas satisfecho de *nuestros* esfuerzos colectivos.

Quiero agradecer a Allen B. Downey y Lauren Cowles su ayuda, paciencia y orientación a la hora de tratar y resolver los problemas de copyright referentes a este libro.

Charles Severance
www.dr-chuck.com
Ann Arbor, MI, USA
9 de Septiembre, 2013

Charles Severance es Profesor Clínico Adjunto en la Escuela de Información (*School of Information*) de la Universidad de Michigan.

Contents

Chapter 1

¿Por qué deberías aprender a escribir programas?

Escribir programas (o programar) es una actividad muy creativa y gratificante. Puedes escribir programas por muchas razones, que pueden ir desde mantenerte activo resolviendo un problema de análisis de datos complejo hasta hacerlo por pura diversión ayudando a otros a resolver un enigma. Este libro asume que *todo el mundo* necesita saber programar, y que una vez que aprendas a programar ya encontrarás qué quieres hacer con esas habilidades recién adquiridas.

En nuestra vida diaria estamos rodeados de computadoras, desde equipos portátiles (laptops) hasta teléfonos móviles (celulares). Podemos pensar en esas computadoras como nuestros "asistentes personales", que pueden ocuparse de muchas tareas por nosotros. El hardware en los equipos que usamos cada día está diseñado esencialmente para hacernos la misma pregunta de forma constante, "¿Qué quieres que haga ahora?"

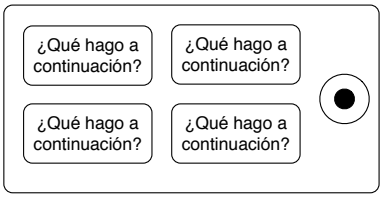

Figure 1.1: Personal Digital Assistant

Los programadores suelen añadir un sistema operativo y un conjunto de aplicaciones al hardware y así nos proporcionan un Asistente Digital Personal que es bastante útil y capaz de ayudarnos a realizar una gran variedad de tareas.

Nuestros equipos son rápidos y tienen grandes cantidades de memoria. Podrían resultarnos muy útiles si tan solo supiéramos qué idioma utilizar para explicarle a la computadora qué es lo que queremos que "haga ahora". Si conociéramos ese idioma, podríamos pedirle al aparato que realizase en nuestro lugar, por ejemplo, tareas repetitivas. Precisamente el tipo de cosas que las computadoras saben hacer mejor suelen ser el tipo de cosas que las personas encontramos pesadas y aburridas.

Por ejemplo, mira los primeros tres párrafos de este capítulos y dime cuál es la palabra que más se repite, y cuántas veces se ha utilizado. Aunque seas capaz de leer y comprender las palabras en pocos segundos, contarlas te resultará casi doloroso, porque la mente humana no fue diseñada para resolver ese tipo de problemas. Para una computadora es justo al revés, leer y comprender texto de un trozo de papel le sería difícil, pero contar las palabras y decirte cuántas veces se ha repetido la más utilizada le resulta muy sencillo:

```
python words.py
Enter file:words.txt
to 16
```

Nuestro "asistente de análisis de información personal" nos dirá enseguida que la palabra "que" se usó nueve veces en los primeros tres párrafos de este capítulo.

El hecho de que los computadores sean buenos en aquellas cosas en las que los humanos no lo son es el motivo por el que necesitas aprender a hablar el "idioma de las computadoras". Una vez que aprendas este nuevo lenguaje, podrás delegar tareas mundanas a tu compañero (la computadora), lo que te dejará más tiempo para ocuparte de las cosas para las que sólo tú estás capacitado. Tú pondrás la creatividad, intuición y el ingenio en esa alianza.

1.1 Creatividad y motivación

A pesar de que este libro no va dirigido a los programadores profesionales, la programación a nivel profesional puede ser un trabajo muy gratificante, tanto a nivel financiero como personal. Crear programas útiles, elegantes e inteligentes para que los usen otros, es una actividad muy creativa. Tu computadora o Asistente Digital Personal (PDA[1]), normalmente contiene muchos programas diferentes pertenecientes a distintos grupos de programadores, cada uno de ellos compitiendo por tu atención e interés. Todos ellos hacen su mejor esfuerzo por adaptarse a tus necesidades y proporcionarte una experiencia de usuario satisfactoria. En ocasiones, cuando elijes un software determinado, sus programadores son directamente recompensados gracias a tu elección.

Si pensamos en los programas como el producto de la creatividad de los programadores, tal vez la figura siguiente sea una versión más acertada de nuestra PDA:

Figure 1.2: Programadores Dirigiéndose a Ti

Por ahora, nuestra principal motivación no es conseguir dinero ni complacer a los usuarios finales, sino simplemente conseguir ser más productivos a nivel personal

[1] Personal Digital Assistant en inglés (N. del T.).

en el manejo de datos e información que encontremos en nuestras vidas. Cuando se empieza por primera vez, uno es a la vez programador y usuario final de sus propios programas. A medida que se gana habilidad como programador, y la programación se hace más creativa para uno mismo, se puede empezar a pensar en desarrollar programas para los demás.

1.2 Arquitectura hardware de las computadoras

Antes de que empecemos a aprender el lenguaje que deberemos hablar para darle instrucciones a las computadoras para desarrollar software, tendremos que aprender un poco acerca de cómo están construidos esas máquinas. Si desmontaras tu computadora o *smartphone* y mirases dentro con atención, encontrarías los siguientes componentes:

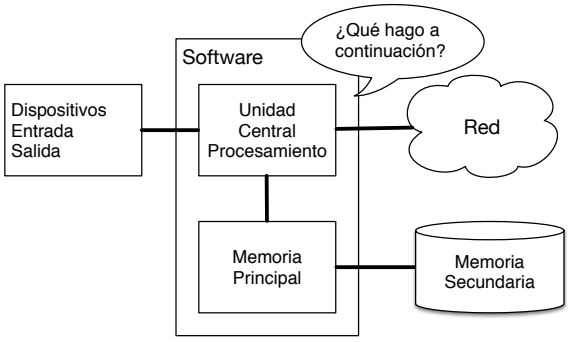

Figure 1.3: Arquitectura Hardware

Las definiciones de alto nivel de esos componentes son las siguientes:

- La *Unidad Central de Procesamiento* (o CPU[2]) es el componente de la computadora diseñado para estar obsesionado con el "¿qué hago ahora?". Si tu equipo está dentro de la clasificación de 3.0 Gigahercios, significa que la CPU preguntará "¿Qué hago ahora?" tres mil millones de veces por segundo. Vas a tener que aprender a hablar muy rápido para mantener el ritmo de la CPU.

- La *Memoria Principal* se usa para almacenar la información que la CPU necesita de forma inmediata. La memoria principal es casi tan rápida como la CPU. Pero la información almacenada en la memoria principal desaparece cuando se apaga el equipo.

- La *Memoria Secundaria* también se utiliza para almacenar información, pero es mucho más lenta que la memoria principal. La ventaja de la memoria secundaria es que puede almacenar la información incluso cuando el equipo está apagado. Algunos ejemplos de memoria secundaria serían las unidades de disco o las memorias flash (que suelen encontrarse en los *pendrives* USB y en los reproductores de música portátiles).

[2]Central Processing Unit en inglés (N. del T.).

- Los *Dispositivos de Entrada y Salida* son simplemente la pantalla, teclado, ratón, micrófono, altavoz, *touchpad*, etc. Incluyen cualquier modo de interactuar con una computadora.
- Actualmente, casi todos los equipos tienen una *Conexión de Red* para recibir información dentro de una red. Podemos pensar en una red como en un lugar donde almacenar y recuperar datos de forma muy lenta, que puede no estar siempre "activo". Así que, en cierto sentido, la red no es más que un tipo de *Memoria Secundaria* más lenta y a veces poco fiable.

Aunque la mayoría de los detalles acerca de cómo funcionan estos componentes es mejor dejársela a los constructores de equipos, resulta útil disponer de cierta terminología para poder referirnos a ellos a la hora de escribir nuestros programas.

Como programador, tu trabajo es usar y orquestar cada uno de esos recursos para resolver el problema del que tengas que ocuparte y analizar los datos de los que dispongas para encontrar la solución. Como programador estarás casi siempre "hablando" con la CPU y diciéndole qué es lo siguiente que debe hacer. A veces le tendrás que pedir a la CPU que use la memoria principal, la secundaria, la red, o los dispositivos de entrada/salida.

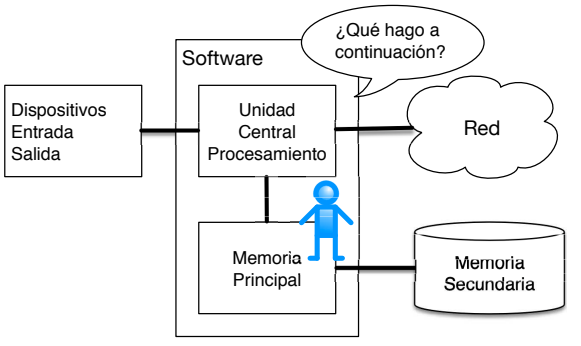

Figure 1.4: ¿Dónde estás?

Tú deberás ser la persona que responda a la pregunta "¿Qué hago ahora?" de la CPU. Pero sería muy incómodo encogerse uno mismo hasta los 5 mm. de altura e introducirse dentro de la computadora sólo para poder dar una orden tres mil millones de veces por segundo. Así que en vez de eso, tendrás que escribir las instrucciones por adelantado. Esas instrucciones almacenadas reciben el nombre de *programa* y el acto de escribirlas y encontrar cuáles son las instrucciones adecuadas, *programar*.

1.3 Comprendiendo la programación

En el resto de este libro, intentaremos convertirte en una persona experta en el arte de programar. Al terminar, te habrás convertido en un *programador* - tal vez no uno profesional, pero al menos tendrás la capacidad de encarar un problema de análisis de datos/información y desarrollar un programa para resolverlo.

En cierto modo, necesitas dos capacidades para ser programador:

- Primero, necesitas saber un lenguaje de programación (Python) - debes conocer su vocabulario y su gramática. Debes ser capaz de deletrear correctamente las palabras en ese nuevo lenguaje y saber construir "frases" bien formadas.
- Segundo, debes "contar una historia". Al escribir un relato, combinas palabras y frases para comunicar una idea al lector. Hay una cierta técnica y arte en la construcción de un relato, y la habilidad para escribir relatos mejora escribiendo y recibiendo cierta respuesta. En programación, nuestro programa es el "relato" y el problema que estás tratando de resolver es la "idea".

Una vez que aprendas un lenguaje de programación como Python, encontrarás mucho más fácil aprender un segundo lenguaje como JavaScript o C++. Cada nuevo lenguaje tiene un vocabulario y gramática muy diferentes, pero la técnica de resolución de problemas será la misma en todos ellos.

Aprenderás el "vocabulario" y "frases" de Python bastante rápido. Te llevará más tiempo el ser capaz de escribir un programa coherente para resolver un problema totalmente nuevo. Se enseña programación de forma muy similar a como se enseña a escribir. Se empieza leyendo y explicando programas, luego se escriben programas sencillos, y a continuación se van escribiendo programas progresivamente más complejos con el tiempo. En algún momento "encuentras tu musa", empiezas a descubrir los patrones por ti mismo y empiezas a ver casi de forma instintiva cómo abordar un problema y escribir un programa para resolverlo. Y una vez alcanzado ese punto, la programación se convierte en un proceso muy placentero y creativo.

Comenzaremos con el vocabulario y la estructura de los programas en Python. Ten paciencia si la simplicidad de los ejemplos te recuerda a cuando aprendiste a leer.

1.4 Palabras y frases

A diferencia de los lenguajes humanos, el vocabulario de Python es en realidad bastante reducido. Llamamos a este "vocabulario" las "palabras reservadas". Se trata de palabras que tienen un significado muy especial para Python. Cuando Python se encuentra estas palabras en un programa, sabe que sólo tienen un único significado para él. Más adelante, cuando escribas programas, podrás usar tus propias palabras con significado, que reciben el nombre de *variables*. Tendrás gran libertad a la hora de elegir los nombres para tus variables, pero no podrás utilizar ninguna de las palabras reservadas de Python como nombre de una variable.

Cuando se entrena a un perro, se utilizan palabras especiales como "siéntate", "quieto" y "tráelo". Cuando te diriges a un perro y no usas ninguna de las palabras reservadas, lo único que consigues es que se te quede mirando con cara extrañada, hasta que le dices una de las palabras que reconoce. Por ejemplo, si dices, "Me gustaría que más gente saliera a caminar para mejorar su salud general", lo que la mayoría de los perros oirían es: "bla bla bla *caminar* bla bla bla bla.". Eso se debe a que "caminar" es una palabra reservada en el lenguaje del perro. Seguramente habrá quien apunte que el lenguaje entre humanos y gatos no dispone de palabras reservadas[3].

[3] http://xkcd.com/231/

Las palabras reservadas en el lenguaje que utilizan los humanos para hablar con Python son, entre otras, las siguientes:

```
and       del       global      not       with
as        elif      if          or        yield
assert    else      import      pass
break     except    in          raise
class     finally   is          return
continue  for       lambda      try
def       from      nonlocal    while
```

Es decir, a diferencia de un perro, Python ya está completamente entrenado. Cada vez le digas "inténtalo", Python lo intentará una vez tras otra sin desfallecer[4].

Aprenderemos cuáles son las palabras reservadas y cómo utilizarlas en su momento, pero por ahora nos centraremos en el equivalente en Python de "habla" (en el lenguaje humano-perro). Lo bueno de pedirle a Python que hable es que podemos incluso indicarle lo que debe decir, pasándole un mensaje entre comillas:

```
print('¡Hola, mundo!')
```

Y ya acabamos de escribir nuestra primera oración sintácticamente correcta en Python. La frase comienza con la función *print* seguida de la cadena de texto que hayamos elegido dentro de comillas simples. Las comillas simples y dobles cumplen la misma función; la mayoría de las personas usan las comillas simples, excepto cuando la cadena de texto contiene también una comilla simple (que puede ser un apóstrofo).

1.5 Conversando con Python

Ahora que ya conocemos una palabra y sabemos cómo crear una frase sencilla en Python, necesitamos aprender a iniciar una conversación con él para comprobar nuestras nuevas capacidades con el lenguaje.

Antes de que puedas conversar con Python, deberás instalar el software necesario en tu computadora y aprender a iniciar Python en ella. En este capítulo no entraremos en detalles sobre cómo hacerlo, pero te sugiero que consultes https://es.py4e.com/ , donde encontrarás instrucciones detalladas y capturas sobre cómo configurar e iniciar Python en sistemas Macintosh y Windows. Si sigues los pasos, llegará un momento en que te encuentres ante una ventana de comandos o terminal. Si escribes entonces *python*, el intérprete de Python empezará a ejecutarse en modo interactivo, y aparecerá algo como esto:

```
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  6 2015, 01:54:25)
[MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

[4] En inglés "inténtalo" es "try", que es también una palabra reservada dentro del lenguaje Python (N. del T.).

El indicador >>> es el modo que tiene el intérprete de Python de preguntarte, "¿Qué quieres que haga ahora?". Python está ya preparado para mantener una conversación contigo. Todo lo que tienes que saber es cómo hablar en su idioma.

Supongamos por ejemplo que aún no conoces ni las palabras ni frases más sencillas de Python. Puede que quieras utilizar el método clásico de los astronautas cuando aterrizan en un planeta lejano e intentan hablar con los habitantes de ese mundo:

```
>>> Vengo en son de paz, por favor llévame ante tu líder
  File "<stdin>", line 1
    Vengo en son de paz, por favor llévame ante tu líder
          ^
SyntaxError: invalid syntax
>>>
```

Esto no se ve bien. A menos que pienses en algo rápidamente, los habitantes del planeta sacarán sus lanzas, te ensartarán, te asarán sobre el fuego y al final les servirás de cena.

Por suerte compraste una copia de este libro durante tus viajes, así que lo abres precisamente por esta página y pruebas de nuevo:

```
>>> print('¡Hola, mundo!')
¡Hola, mundo!
```

Esto tiene mejor aspecto, de modo que intentas comunicarte un poco más:

```
>>> print('Usted debe ser el dios legendario que viene del cielo')
Usted debe ser el dios legendario que viene del cielo
>>> print('Hemos estado esperándole durante mucho tiempo')
Hemos estado esperándole durante mucho tiempo
>>> print('La leyenda dice que debe estar usted muy rico con mostaza')
La leyenda dice que debe estar usted muy rico con mostaza
>>> print 'Tendremos un festín esta noche a menos que diga
  File "<stdin>", line 1
    print 'Tendremos un festín esta noche a menos que diga
                                                         ^
SyntaxError: Missing parentheses in call to 'print'
>>>
```

La conversación fue bien durante un rato, pero en cuanto cometiste el más mínimo fallo al utilizar el lenguaje Python, Python volvió a sacar las lanzas.

En este momento, te habrás dado cuenta que a pesar de que Python es tremendamente complejo y poderoso, y muy estricto en cuanto a la sintaxis que debes usar para comunicarte con él, Python *no* es inteligente. En realidad estás solamente manteniendo una conversación contigo mismo; eso sí, usando una sintaxis adecuada.

En cierto modo, cuando utilizas un programa escrito por otra persona, la conversación se mantiene entre tú y el programador, con Python actuando meramente de

intermediario. Python es una herramienta que permite a los creadores de programas expresar el modo en que la conversación supuestamente debe fluir. Y dentro de unos pocos capítulos más, serás uno de esos programadores que utilizan Python para hablar con los usuarios de tu programa.

Antes de que abandonemos nuestra primera conversación con el intérprete de Python, deberías aprender cual es el modo correcto de decir "adiós" al interactuar con los habitantes del Planeta Python:

```
>>> adiós
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'adiós' is not defined
>>> if you don't mind, I need to leave\footnote{si no te importa, tengo que marcha
  File "<stdin>", line 1
    if you don't mind, I need to leave
              ^
SyntaxError: invalid syntax
>>> quit()
```

Te habrás fijado en que el error es diferente en cada uno de los dos primeros intentos. El segundo error es diferente porque *if* es una palabra reservada, y cuando Python la ve, cree que estamos intentando decirle algo, pero encuentra la sintaxis de la frase incorrecta.

La forma correcta de decirle "adiós" a Python es introducir *quit()* en el símbolo indicador del sistema >>>. Seguramente te hubiera llevado un buen rato adivinarlo, así que tener este libro a mano probablemente te haya resultado útil.

1.6 Terminología: intérprete y compilador

Python es un lenguaje *de alto nivel*, pensado para ser relativamente sencillo de leer y escribir para las personas, y fácil de leer y procesar para las máquinas. Otros lenguajes de alto nivel son Java, C++, PHP, Ruby, Basic, Perl, JavaScript, y muchos más. El hardware real que está dentro de la Unidad Central de Procesamiento (CPU), no entiende ninguno de esos lenguajes de alto nivel.

La CPU entiende únicamente un lenguaje llamado *lenguaje de máquina* o *código máquina*. El código máquina es muy simple y francamente muy pesado de escribir, ya que está representado en su totalidad por solamente ceros y unos:

```
00101000111010010010101010000001111
11100110000011101010010101101101
...
```

El código máquina parece bastante sencillo a simple vista, dado que sólo contiene ceros y unos, pero su sintaxis es incluso más compleja y mucho más enrevesada que la de Python, razón por la cual muy pocos programadores escriben en código máquina. En vez de eso, se han creado varios programas traductores para permitir a los programadores escribir en lenguajes de alto nivel como Python o Javascript,

y son esos traductores quienes convierten los programas a código máquina, que es lo que ejecuta en realidad la CPU.

Dado que el código máquina está ligado al hardware de la máquina que lo ejecuta, ese código no es *portable* (trasladable) entre equipos de diferente tipo. Los programas escritos en lenguajes de alto nivel pueden ser trasladados entre distintas máquinas usando un intérprete diferente en cada una de ellas, o recompilando el código para crear una versión diferente del código máquina del programa para cada uno de los tipos de equipo.

Esos traductores de lenguajes de programación forman dos categorías generales: (1) intérpretes y (2) compiladores.

Un *intérprete* lee el código fuente de los programas tal y como ha sido escrito por el programador, lo analiza, e interpreta sus instrucciones sobre la marcha. Python es un intérprete y cuando lo estamos ejecutando de forma interactiva, podemos escribir una línea de Python (una frase), y este la procesa de forma inmediata, quedando listo para que podamos escribir otra línea.

Algunas de esas líneas le indican a Python que tú quieres que recuerde cierto valor para utilizarlo más tarde. Tenemos que escoger un nombre para que ese valor sea recordado y usaremos ese nombre simbólico para recuperar el valor más tarde. Utilizamos el término *variable* para denominar las etiquetas que usamos para referirnos a esos datos almacenados.

```
>>> x = 6
>>> print(x)
6
>>> y = x * 7
>>> print(y)
42
>>>
```

En este ejemplo, le pedimos a Python que recuerde el valor seis y use la etiqueta x para que podamos recuperar el valor más tarde. Comprobamos que Python ha guardado de verdad el valor usando *print*. Luego le pedimos a Python que recupere x, lo multiplique por siete y guarde el valor calculado en y. Finalmente, le pedimos a Python que escriba el valor actual de y.

A pesar de que estamos escribiendo estos comandos en Python línea a línea, Python los está tratando como una secuencia ordenada de sentencias, en la cual las últimas frases son capaces de obtener datos creados en las anteriores. Estamos, por tanto, escribiendo nuestro primer párrafo sencillo con cuatro frases en un orden lógico y útil.

La esencia de un *intérprete* consiste en ser capaz de mantener una conversación interactiva como la mostrada más arriba. Un *compilador* necesita que le entreguen el programa completo en un fichero, y luego ejecuta un proceso para traducir el código fuente de alto nivel a código máquina, tras lo cual coloca ese código máquina resultante dentro de otro fichero para su ejecución posterior.

En sistemas Windows, a menudo esos ejecutables en código máquina tienen un sufijo o extensión como ".exe" o ".dll", que significan "ejecutable" y "librería de

enlace dinámico" (*dynamic link library*) respectivamente. En Linux y Macintosh, no existe un sufijo que identifique de manera exclusiva a un fichero como ejecutable.

Si abrieras un fichero ejecutable con un editor de texto, verías algo complemente disparatado e ilegible:

```
^?ELF^A^A^A^@^@^@^@^@^@^@^@^B^@^C^@^A^@^@^@\xa0\x82
^D^H4^@^@^@\x90^]^@^@^@^@^@^@4^@ ^@^G^@(^@$^@!^@^F^@
^@^@4^@^@^@4\x80^D^H4\x80^D^H\xe0^@^@^@\xe0^@^@^@^E
^@^@^@^D^@^@^@^C^@^@^@^T^A^@^@^T\x81^D^H^T\x81^D^H^S
^@^@^@^S^@^@^@^D^@^@^@^A^@^@^@^A\^D^HQVhT\x83^D^H\xe8
....
```

No resulta fácil leer o escribir código máquina, pero afortunadamente disponemos de *intérpretes* y *compiladores* que nos permiten escribir en lenguajes de alto nivel, como Python o C.

Llegados a este punto en la explicación acerca de los compiladores e intérpretes, seguramente te estarás preguntando algunas cosas acerca del propio intérprete de Python. ¿En qué lenguaje está escrito? ¿Está escrito en un lenguaje compilado? Cuando escribimos "python", ¿qué ocurre exactamente?

El intérprete de Python está escrito en un lenguaje de alto nivel llamado "C". En realidad, puedes ver el código fuente del intérprete de Python acudiendo a www.python.org e incluso modificarlo a tu gusto. Quedamos, pues, en que Python es en sí mismo un programa y que está compilado en código máquina. Cuando instalaste Python en tu computadora (o el vendedor lo instaló), colocaste una copia del código máquina del programa Python traducido para tu sistema. En Windows, el código máquina ejecutable para el intérprete de Python es probablemente un fichero con un nombre como:

```
C:\Python35\python.exe
```

En realidad, eso ya es más de lo que necesitas saber para ser un programador en Python, pero algunas veces vale la pena contestar estas pequeñas dudas recurrentes justo al principio.

1.7 Escribiendo un programa

Escribir comandos en el intérprete de Python es un buen modo de experimentar con las capacidades de Python, pero no es lo más recomendado a la hora de resolver problemas más complejos.

Cuando queremos escribir un programa, usamos un editor de texto para escribir las instrucciones de Python en un fichero, que recibe el nombre de *script*. Por convención, los scripts de Python tienen nombres que terminan en `.py`.

Para ejecutar el script, hay que indicarle al intérprete de Python el nombre del fichero. En una ventana de comandos de Unix o Windows, escribirías `python hola.py`, de este modo:

```
csev$ cat hola.py
print('¡Hola, mundo!')
csev$ python hola.py
¡Hola, mundo!
csev$
```

"csev$" es el indicador (*prompt*) del sistema operativo, y el comando "cat hola.py" nos muestra que el archivo "hola.py" contiene un programa con una única línea de código que imprime en pantalla una cadena de texto.

Llamamos al intérprete de Python y le pedimos que lea el código fuente desde el archivo "hola.py", en vez de esperar a que vayamos introduciendo líneas de código Python de forma interactiva.

Habrás notado que cuando trabajamos con un fichero no necesitamos incluir el comando *quit()* al final del programa Python. Cuando Python va leyendo tu código fuente desde un archivo, sabe que debe parar cuando llega al final del fichero.

1.8 ¿Qué es un programa?

Podemos definir un *programa*, en su forma más básica, como una secuencia de declaraciones o sentencias que han sido diseñadas para hacer algo. Incluso nuestro sencillo script "hola.py" es un programa. Es un programa de una sola línea y no resulta particularmente útil, pero si nos ajustamos estrictamente a la definición, se trata de un programa en Python.

Tal vez resulte más fácil comprender qué es un programa pensando en un problema que pudiera ser resuelto a través de un programa, y luego estudiando cómo sería el programa que solucionaría ese problema.

Supongamos que estás haciendo una investigación de computación o informática social en mensajes de Facebook, y te interesa conocer cual es la palabra más utilizada en un conjunto de mensajes. Podrías imprimir el flujo de mensajes de Facebook y revisar con atención el texto, buscando la palabra más común, pero sería un proceso largo y muy propenso a errores. Sería más inteligente escribir un programa en Python para encargarse de la tarea con rapidez y precisión, y así poder emplear el fin de semana en hacer otras cosas más divertidas.

Por ejemplo, fíjate en el siguiente texto, que trata de un payaso y un coche. Estúdialo y trata de averiguar cual es la palabra más común y cuántas veces se repite.

```
el payaso corrió tras el coche y el coche se metió dentro de la tienda
y la tienda cayó sobre el payaso y el coche
```

Ahora imagina que haces lo mismo pero buscando a través de millones de líneas de texto. Francamente, tardarías menos aprendiendo Python y escribiendo un programa en ese lenguaje para contar las palabras que si tuvieras que ir revisando todas ellas una a una.

Pero hay una noticia aún mejor, y es que se me ha ocurrido un programa sencillo para encontrar cuál es la palabra más común dentro de un fichero de texto. Ya lo escribí, lo probé, y ahora te lo regalo para que lo puedas utilizar y ahorrarte mucho tiempo.

```
name = input('Enter file:')
handle = open(name, 'r')
counts = dict()

for line in handle:
    words = line.split()
    for word in words:
        counts[word] = counts.get(word, 0) + 1

bigcount = None
bigword = None
for word, count in list(counts.items()):
    if bigcount is None or count > bigcount:
        bigword = word
        bigcount = count

print(bigword, bigcount)

# Code: http://www.py4e.com/code3/words.py
```

No necesitas ni siquiera saber Python para usar este programa. Tendrás que llegar hasta el capítulo 10 de este libro para entender por completo las impresionantes técnicas de Python que se han utilizado para crearlo. Ahora eres el usuario final, sólo tienes que usar el programa y sorprenderte de sus habilidades y de cómo te permite ahorrar un montón de esfuerzo. Tan sólo tienes que escribir el código dentro de un fichero llamado *words.py* y ejecutarlo, o puedes descargar el código fuente directamente desde https://es.py4e.com/code3/ y ejecutarlo.

Este es un buen ejemplo de cómo Python y el lenguaje Python actúan como un intermediario entre tú (el usuario final) y yo (el programador). Python es un medio para que intercambiemos secuencias de instrucciones útiles (es decir, programas) en un lenguaje común que puede ser usado por cualquiera que instale Python en su computadora. Así que ninguno de nosotros está hablando *con Python*, sino que estamos comunicándonos uno con el otro *a través de* Python.

1.9 Los bloques de construcción de los programas

En los próximos capítulos, aprenderemos más sobre el vocabulario, la estructura de las frases y de los párrafos y la estructura de los relatos en Python. Aprenderemos cuáles son las poderosas capacidades de Python y cómo combinar esas capacidades entre sí para crear programas útiles.

Hay ciertos patrones conceptuales de bajo nivel que se usan para estructurar los programas. Esas estructuras no son exclusivas de Python, sino que forman parte de cualquier lenguaje de programación, desde el código máquina hasta los lenguajes de alto nivel.

entrada Obtener datos del "mundo exterior". Puede consistir en leer datos desde un fichero, o incluso desde algún tipo de sensor, como un micrófono o un GPS.

En nuestros primeros programas, las entradas van a provenir del usuario, que introducirá los datos a través del teclado.

salida Mostrar los resultados del programa en una pantalla, almacenarlos en un fichero o incluso es posible enviarlos a un dispositivo como un altavoz para reproducir música o leer un texto.

ejecución secuencial Ejecutar una sentencia tras otra en el mismo orden en que se van encontrando en el *script*.

ejecución condicional Comprobar ciertas condiciones y luego ejecutar u omitir una secuencia de sentencias.

ejecución repetida Ejecutar un conjunto de sentencias varias veces, normalmente con algún tipo de variación.

reutilización Escribir un conjunto de instrucciones una vez, darles un nombre y así poder reutilizarlas luego cuando se necesiten en cualquier punto de tu programa.

Parece demasiado simple para ser cierto, y por supuesto nunca es tan sencillo. Es como si dijéramos que andar es simplemente "poner un pie delante del otro". El "arte" de escribir un programa es componer y entrelazar juntos esos elementos básicos muchas veces hasta conseguir al final algo que resulte útil para sus usuarios.

El programa para contar palabras que vimos antes utiliza al mismo tiempo todos esos patrones excepto uno.

1.10 ¿Qué es posible que vaya mal?

Como vimos en nuestra anterior conversación con Python, debemos comunicarnos con mucha precisión cuando escribimos código Python. El menor error provocará que Python se niegue a hacer funcionar tu programa.

Los programadores novatos a menudo se toman el hecho de que Python no permita cometer errores como la prueba definitiva de que es perverso, odioso y cruel. A pesar de que a Python parece gustarle todos los demás, es capaz de identificar a los novatos en concreto, y les guarda un gran rencor. Debido a ello, toma sus programas perfectamente escritos, y los rechaza, considerándolos como "inservibles", sólo para atormentarlos.

```
>>> primt '¡Hola, mundo!'
  File "<stdin>", line 1
    primt '¡Hola, mundo!'
                ^
SyntaxError: invalid syntax
>>> primt ('Hola, mundo')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'primt' is not defined

>>> ¡Te odio, Python!
  File "<stdin>", line 1
    ¡Te odio, Python!
     ^
```

```
SyntaxError: invalid syntax
>>> si sales fuera, te daré una lección
  File "<stdin>", line 1
    si sales fuera, te daré una lección
               ^
SyntaxError: invalid syntax
>>>
```

No es mucho lo que se gana discutiendo con Python. Solo es una herramienta. No tiene emociones, es feliz y está preparado para servirte en el momento que lo necesites. Sus mensajes de error parecen crueles, pero simplemente se trata de una petición de ayuda del propio Python. Ha examinado lo que has tecleado, y sencillamente no es capaz de entender lo que has escrito.

Python se parece mucho a un perro, te quiere incondicionalmente, entiende algunas pocas palabras clave, te mira con una mirada dulce en su cara(>>>), y espera que le digas algo que él pueda comprender. Cuando Python dice "SyntaxError: invalid syntax" (Error de sintaxis: sintaxis inválida), tan solo está agitando su cola y diciendo: "Creo que has dicho algo, pero no te entiendo; de todos modos, por favor, sigue hablando conmigo (>>>)."

A medida que tus programas vayan aumentando su complejidad, te encontrarás con tres tipos de errores generales:

Errores de sintaxis (Syntax errors) Estos son los primeros errores que cometerás y también los más fáciles de solucionar. Un error de sintaxis significa que has violado las reglas "gramaticales" de Python. Python hace todo lo que puede para señalar el punto exacto, la línea y el carácter donde ha detectado el fallo. Lo único complicado de los errores de sintaxis es que a veces el error que debe corregirse está en realidad en una línea anterior a la cual Python *detectó* ese fallo. De modo que la línea y el carácter que Python indica en un error de sintaxis pueden ser tan sólo un punto de partida para tu investigación.

Errores lógicos Se produce un error lógico cuando un programa tiene una sintaxis correcta, pero existe un error en el orden de las sentencias o en la forma en que están relacionadas unas con otras. Un buen ejemplo de un error lógico sería: "toma un trago de tu botella de agua, ponla en tu mochila, camina hasta la biblioteca y luego vuelve a enroscar la tapa en la botella."

Errores semánticos Un error semántico ocurre cuando la descripción que has brindado de los pasos a seguir es sintácticamente perfecta y está en el orden correcto, pero sencillamente hay un error en el programa. El programa es correcto, pero no hace lo que tú *pretendías* que hiciera. Un ejemplo podría ser cuando le das indicaciones a alguien sobre cómo llegar a un restaurante, y le dices "...cuando llegues a la intersección con la gasolinera, gira a la izquierda, continúa durante otro kilómetro y el restaurante es el edificio rojo que encontrarás a tu izquierda.". Tu amigo se retrasa y te llama para decirte que está en una granja dando vueltas alrededor de un granero, sin rastro alguno de un restaurante. Entonces le preguntas "¿giraste a la izquierda o la derecha?", y te responde "Seguí tus indicaciones al pie de la letra, dijiste que girara a la izquierda y continuar un kilómetro desde la gasolinera.", entonces le respondes "Lo siento mucho, porque a pesar de que mis indica-

ciones fueron sintácticamente correctas, tristemente contenían un pequeño pero indetectado error semántico.".

Insisto en que, ante cualquiera de estos tres tipos de errores, Python únicamente hace lo que está a su alcance por seguir al pie de la letra lo que tú le has pedido que haga.

1.11 Depurando los programas

Cuando Python muestra un error, u obtiene un resultado diferente al que esperabas, empieza una intensa búsqueda de la causa del error. Depurar es el proceso de encontrar la causa o el origen de ese error en tu código. Cuando depuras un programa, y especialmente cuando tratas con un *bug* algo difícil de solucionar, existen cuatro cosas por hacer:

leer Revisar tu código, leerlo de nuevo, y asegurarte de que ahí está expresado de forma correcta lo que quieres decir.

ejecutar Prueba haciendo cambios y ejecutando diferentes versiones. Con frecuencia, si muestras en tu programa lo correcto en el lugar indicado, el problema se vuelve obvio, pero en ocasiones debes invertir algo de tiempo hasta conseguirlo.

pensar detenidamente ¡Toma tu tiempo para pensar!, ¿A qué tipo de error corresponde: sintaxis, en tiempo de ejecución, semántico?, ¿Qué información puedes obtener de los mensajes de error, o de la salida del programa?, ¿Qué tipo de errores podría generar el problema que estás abordando?, ¿Cuál fue el último cambio que hiciste, antes de que se presentara el problema?

retroceder En algún momento, lo mejor que podrás hacer es dar marcha atrás, deshacer los cambios recientes hasta obtener de nuevo un programa que funcione y puedas entender. Llegado a ese punto, podrás continuar con tu trabajo.

Algunas veces, los programadores novatos se quedan estancados en una de estas actividades y olvidan las otras. Encontrar un *bug* requiere leer, ejecutar, pensar detenidamente y algunas veces retroceder. Si te bloqueas en alguna de estas actividades, prueba las otras. Cada actividad tiene su procedimiento de análisis.

Por ejemplo, leer tu código podría ayudar si el problema es un error tipográfico, pero no si es uno conceptual. Si no comprendes lo que hace tu programa, puedes leerlo 100 veces y no encontrarás el error, puesto que dicho error está en tu mente.

Experimentar puede ayudar, especialmente si ejecutas pequeñas pruebas. Pero si experimentas sin pensar o leer tu código, podrías caer en un patrón que llamo "programación de paseo aleatorio", que es el proceso de realizar cambios al azar hasta que el programa logre hacer lo que debería. No hace falta mencionar que este tipo de programación puede tomar mucho tiempo.

Debes tomar el tiempo suficiente para pensar. Depurar es como una ciencia experimental. Debes plantear al menos una hipótesis sobre qué podría ser el problema. Si hay dos o más posibilidades, piensa en alguna prueba que pueda ayudarte a descartar una de ellas.

Descansar y conversar ayuda a estimular el pensamiento. Si le explicas el problema a alguien más (o incluso a tí mismo), a veces encontrarás la respuesta antes de terminar la pregunta.

Pero incluso las mejores técnicas de depuración fallarán si hay demasiados errores, o si el código que intentas mejorar es demasiado extenso y complicado. Algunas veces la mejor opción es retroceder, y simplificar el programa hasta que obtengas algo que funcione y puedas entender.

Por lo general, los programadores novatos son reacios a retroceder porque no soportan tener que borrar una línea de código (incluso si está mal). Si te hace sentir mejor, prueba a copiar tu programa en otro archivo antes de empezar a modificarlo. De esa manera, puedes recuperar poco a poco pequeñas piezas de código que necesites.

1.12 El camino del aprendizaje

Según vayas avanzando por el resto del libro, no te asustes si los conceptos no parecen encajar bien unos con otros al principio. Cuando estabas aprendiendo a hablar, no supuso un problema que durante los primeros años solo pudieras emitir lindos balbuceos. Y también fue normal que te llevara seis meses pasar de un vocabulario simple a frases simples, y que te llevara 5-6 años más pasar de frases a párrafos, y que todavía tuvieran que transcurrir unos cuantos años más hasta que fuiste capaz de escribir tu propia historia corta interesante.

Pretendemos que aprendas Python rápidamente, por lo que te enseñaremos todo al mismo tiempo durante los próximos capítulos. Aún así, ten en cuenta que el proceso es similar a aprender un idioma nuevo, que lleva un tiempo absorber y comprender antes de que te resulte familiar. Eso produce cierta confusión, puesto que revisaremos en distintas ocasiones determinados temas, y trataremos que de esa manera puedas visualizar los pequeños fragmentos que componen esta obra completa. A pesar de que el libro está escrito de forma lineal, no dudes en ser no lineal en la forma en que abordes las materias. Avanza y retrocede, y lee a veces por encima. Al ojear material más avanzado sin comprender del todo los detalles, tendrás una mejor comprensión del "¿por qué?" de la programación. Al revisar el material anterior e incluso al realizar nuevamente los ejercicios previos, te darás cuenta que ya has aprendido un montón de cosas, incluso si el tema que estás examinando en ese momento parece un poco difícil de abordar.

Normalmente, cuando uno aprende su primer lenguaje de programación, hay unos pocos momentos "¡A-já!" estupendos, en los cuales puedes levantar la vista de la roca que estás machacando con martillo y cincel, separarte unos pasos y comprobar que lo que estás intentando construir es una maravillosa escultura.

Si algo parece particularmente difícil, generalmente no vale la pena quedarse mirándolo toda la noche. Respira, toma una siesta, come algo, explícale a alguien (quizás a tu perro) con qué estás teniendo problemas, y después vuelve a observarlo con un perspectiva diferente. Te aseguro que una vez que aprendas los conceptos de la programación en el libro, volverás atrás y verás que en realidad todo era fácil, elegante y que simplemente te ha llevado un poco de tiempo llegar a absorberlo.

1.13 Glosario

bug Un error en un programa.

código fuente Un programa en un lenguaje de alto nivel.

código máquina El lenguaje de más bajo nivel para el software, es decir, el lenguaje que es directamente ejecutado por la unidad central de procesamiento (CPU).

compilar Traducir un programa completo escrito en un lenguaje de alto nivel a un lenguaje de bajo nivel, para dejarlo listo para una ejecución posterior.

error semántico Un error dentro de un programa que provoca que haga algo diferente de lo que pretendía el programador.

función print Una instrucción que hace que el intérprete de Python muestre un valor en la pantalla.

indicador de línea de comandos (*prompt*) Cuando un programa muestra un mensaje y se detiene para que el usuario teclee algún tipo de dato.

interpretar Ejecutar un programa escrito en un lenguaje de alto nivel traduciéndolo línea a línea.

lenguaje de alto nivel Un lenguaje de programación como Python, que ha sido diseñado para que sea fácil de leer y escribir por las personas.

lenguaje de bajo nivel Un lenguaje de programación que está diseñado para ser fácil de ejecutar para una computadora; también recibe el nombre de "código máquina", "lenguaje máquina" o "lenguaje ensamblador".

memoria principal Almacena los programas y datos. La memoria principal pierde su información cuando se desconecta la alimentación.

memoria secundaria Almacena los programas y datos y mantiene su información incluso cuando se interrumpe la alimentación. Es generalmente más lenta que la memoria principal. Algunos ejemplos de memoria secundaria son las unidades de disco y memorias flash que se encuentran dentro de los dispositivos USB.

modo interactivo Un modo de usar el intérprete de Python, escribiendo comandos y expresiones directamente en el indicador de la línea de comandos.

parsear Examinar un programa y analizar la estructura sintáctica.

portabilidad Es la propiedad que poseen los programas que pueden funcionar en más de un tipo de computadora.

programa Un conjunto de instrucciones que indican cómo realizar algún tipo de cálculo.

resolución de un problema El proceso de formular un problema, encontrar una solución, y mostrar esa solución.

semántica El significado de un programa.

unidad central de procesamiento El corazón de cualquier computadora. Es lo que ejecuta el software que escribimos. También recibe el nombre de "CPU" por sus siglas en inglés (*Central Processsing Unit*), o simplemente, "el procesador".

1.14 Ejercicios

Ejercicio 1: ¿Cuál es la función de la memoria secundaria en una computadora?

a) Ejecutar todos los cálculos y la lógica del programa

b) Descargar páginas web de Internet
c) Almacenar información durante mucho tiempo, incluso después de ciclos de apagado y encendido
d) Recolectar la entrada del usuario

Ejercicio 2: ¿Qué es un programa?

Ejercicio 3: ¿Cuál es la diferencia entre un compilador y un intérprete?

Ejercicio 4: ¿Cuál de los siguientes contiene "código máquina"?

a) El intérprete de Python
b) El teclado
c) El código fuente de Python
d) Un documento de un procesador de texto

Ejercicio 5: ¿Qué está mal en el siguiente código?:

```
>>> primt '¡Hola, mundo!'
  File "<stdin>", line 1
    primt '¡Hola, mundo!'
                ^
SyntaxError: invalid syntax
>>>
```

Ejercicio 6: ¿En qué lugar del computador queda almacenada una variable, como en este caso "X", después de ejecutar la siguiente línea de Python?:

```
x = 123
```

a) Unidad central de procesamiento
b) Memoria Principal
c) Memoria Secundaria
d) Dispositivos de Entrada
e) Dispositivos de Salida

Ejercicio 7: ¿Qué mostrará en pantalla el siguiente programa?:

```
x = 43
x = x + 1
print(x)
```

a) 43
b) 44
c) x + 1
d) Error, porque x = x + 1 no es posible matemáticamente.

Ejercicio 8: Explica cada uno de los siguientes conceptos usando un ejemplo de una capacidad humana: (1) Unidad central de procesamiento, (2) Memoria principal, (3) Memoria secundaria, (4) Dispositivos de entrada, y (5) Dispositivos de salida. Por ejemplo, "¿Cuál sería el equivalente humano de la Unidad central de procesamiento?".

Ejercicio 9: ¿Cómo puedes corregir un "Error de sintaxis"?.

Chapter 2

Variables, expresiones y sentencias

2.1 Valores y tipos

Un *valor* es una de las cosas básicas que utiliza un programa, como una letra o un número. Los valores que hemos visto hasta ahora han sido `1`, `2`, y "¡Hola, mundo!"

Esos valores pertenecen a *tipos* diferentes: `2` es un entero (int), y "¡Hola, mundo!" es una *cadena* (string), que recibe ese nombre porque contiene una "cadena" de letras. Tú (y el intérprete) podéis identificar las cadenas porque van encerradas entre comillas.

La sentencia `print` también funciona con enteros. Vamos a usar el comando `python` para iniciar el intérprete.

```
python
>>> print(4)
4
```

Si no estás seguro de qué tipo de valor estás manejando, el intérprete te lo puede decir.

```
>>> type(''¡Hola, mundo!)
<class 'str'>
>>> type(17)
<class 'int'>
```

Not surprisingly, strings belong to the type `str` and integers belong to the type `int`. Less obviously, numbers with a decimal point belong to a type called `float`, because these numbers are represented in a format called *floating point*.

```
>>> type(3.2)
<class 'float'>
```

¿Qué ocurre con valores como "17" y "3.2"? Parecen números, pero van entre comillas como las cadenas.

```
>>> type('17')
<class 'str'>
>>> type('3.2')
<class 'str'>
```

Son cadenas.

Cuando escribes un entero grande, puede que te sientas tentado a usar comas o puntos para separarlo en grupos de tres dígitos, como en `1,000,000` [1]. Eso no es un entero válido en Python, pero en cambio sí que resulta válido algo como:

```
>>> print(1,000,000)
1 0 0
```

Bien, ha funcionado. ¡Pero eso no era lo que esperábamos!. Python interpreta `1,000,000` como una secuencia de enteros separados por comas, así que lo imprime con espacios en medio.

Éste es el primer ejemplo que hemos visto de un error semántico: el código funciona sin producir ningún mensaje de error, pero no hace su trabajo "correctamente".

2.2 Variables

Una de las características más potentes de un lenguaje de programación es la capacidad de manipular *variables*. Una variable es un nombre que se refiere a un valor.

Una *sentencia de asignación* crea variables nuevas y las da valores:

```
>>> mensaje = 'Y ahora algo completamente diferente'
>>> n = 17
>>> pi = 3.1415926535897931
```

Este ejemplo hace tres asignaciones. La primera asigna una cadena a una variable nueva llamada `mensaje`; la segunda asigna el entero `17` a `n`; la tercera asigna el valor (aproximado) de π a `pi`.

Para mostrar el valor de una variable, se puede usar la sentencia print:

```
>>> print(n)
17
>>> print(pi)
3.141592653589793
```

El tipo de una variable es el tipo del valor al que se refiere.

[1] En el mundo anglosajón el "separador de millares" es la coma, y no el punto (Nota del trad.)

```
>>> type(mensaje)
<class 'str'>
>>> type(n)
<class 'int'>
>>> type(pi)
<class 'float'>
```

2.3 Nombres de variables y palabras claves

Los programadores generalmente eligen nombres para sus variables que tengan sentido y documenten para qué se usa esa variable.

Los nombres de las variables pueden ser arbitrariamente largos. Pueden contener tanto letras como números, pero no pueden comenzar con un número. Se pueden usar letras mayúsculas, pero es buena idea comenzar los nombres de las variables con una letras minúscula (veremos por qué más adelante).

El carácter guión-bajo (_) puede utilizarse en un nombre. A menudo se utiliza en nombres con múltiples palabras, como en `mi_nombre` o `velocidad_de_golondrina_sin_carg` Los nombres de las variables pueden comenzar con un carácter guión-bajo, pero generalmente se evita usarlo así a menos que se esté escribiendo código para librerías que luego utilizarán otros.

Si se le da a una variable un nombre no permitido, se obtiene un error de sintaxis:

```
>>> 76trombones = 'gran desfile'
SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Teorema avanzado de Zymurgy'
SyntaxError: invalid syntax
```

`76trombones` es incorrecto porque comienza por un número. `more@` es incorrecto porque contiene un carácter no premitido, `@`. Pero, ¿qué es lo que está mal en `class`?

Pues resulta que `class` es una de las *palabras clave* de Python. El intérprete usa palabras clave para reconocer la estructura del programa, y esas palabras no pueden ser utilizadas como nombres de variables.

Python reserva 33 palabras claves para su propio uso:

```
and       del       from      None      True
as        elif      global    nonlocal  try
assert    else      if        not       while
break     except    import    or        with
class     False     in        pass      yield
continue  finally   is        raise
def       for       lambda    return
```

Puede que quieras tener esta lista a mano. Si el intérprete se queja por el nombre de una de tus variables y no sabes por qué, comprueba si ese nombre está en esta lista.

2.4 Sentencias

Una *sentencia* es una unidad de código que el intérprete de Python puede ejecutar. Hemos visto hasta ahora dos tipos de sentencia: print y las asignaciones.

Cuando escribes una sentencia en modo interactivo, el intérprete la ejecuta y muestra el resultado, si es que lo hay.

Un script normalmente contiene una secuencia de sentencias. Si hay más de una sentencia, los resultados aparecen de uno en uno según se van ejecutando las sentencias.

Por ejemplo, el script

```
print(1)
x = 2
print(x)
```

produce la salida

```
1
2
```

La sentencia de asignación no produce ninguna salida.

2.5 Operadores y operandos

Los operadores son símbolos especiales que representan cálculos, como la suma o la multiplicación. Los valores a los cuales se aplican esos operadores reciben el nombre de *operandos*.

Los operadores `+`, `-`, , `/`, y `\*` realizan sumas, restas, multiplicaciones, divisiones y exponenciación (elevar un número a una potencia), como se muestra en los ejemplos siguientes:

```
20+32
hour-1
hour*60+minute
minute/60
5**2
(5+9)*(15-7)
```

Ha habido un cambio en el operador de división entre Python 2.x y Python 3.x. En Python 3.x, el resultado de esta división es un resultado de punto flotante:

```
>>> minute = 59
>>> minute/60
0.9833333333333333
```

El operador de división en Python 2.0 dividiría dos enteros y truncar el resultado a un entero:

```
>>> minute = 59
>>> minute/60
0
```

Para obtener la misma respuesta en Python 3.0 use división dividida (// integer).

```
>>> minute = 59
>>> minute//60
0
```

En Python 3, la división de enteros funciona mucho más como cabría esperar. Si ingresaste la expresión en una calculadora.

2.6 Expresiones

Una *expresión* es una combinación de valores, variables y operadores. Un valor por si mismo se considera una expresión, y también lo es una variable, así que las siguientes expresiones son todas válidas (asumiendo que la variable `x` tenga un valor asignado):

```
17
x
x + 17
```

Si escribes una expresión en modo interactivo, el intérprete la *evalúa* y muestra el resultado:

```
>>> 1 + 1
2
```

Sin embargo, en un script, ¡una expresión por si misma no hace nada! Esto a menudo puede producir confusión entre los principiantes.

Ejercicio 1: Escribe las siguientes sentencias en el intérprete de Python para comprobar qué hacen:

```
5
x = 5
x + 1
```

2.7 Orden de las operaciones

Cuando en una expresión aparece más de un operador, el orden de evaluación depende de las *reglas de precedencia*. Para los operadores matemáticos, Python sigue las convenciones matemáticas. El acrónimo *PEMDSR* resulta útil para recordar esas reglas:

- Los *P*aréntesis tienen el nivel superior de precedencia, y pueden usarse para forzar a que una expresión sea evaluada en el orden que se quiera. Dado que las expresiones entre paréntesis son evaluadas primero, `2 * (3-1)` es 4, y `(1+1)**(5-2)` es 8. Se pueden usar también paréntesis para hacer una expresión más sencilla de leer, incluso si el resultado de la misma no varía por ello, como en `(minuto * 100) / 60`.
- La *E*xponenciación (elevar un número a una potencia) tiene el siguiente nivel más alto de precedencia, de modo que `2**1+1` es 3, no 4, y `3*1**3` es 3, no 27.
- La *M*ultiplicación y la *D*ivisión tienen la misma precedencia, que es superior a la de la *S*uma y la *R*esta, que también tienen entre si el mismo nivel de precedencia. Así que `2*3-1` es 5, no 4, y `6+4/2` es 8, no 5.
- Los operadores con igual precedencia son evaluados de izquierda a derecha. Así que la expresión `5-3-1` es 1 y no 3, ya que `5-3` se evalúa antes, y después se resta `1` de `2`.

En caso de duda, añade siempre paréntesis a tus expresiones para asegurarte de que las operaciones se realizan en el orden que tú quieres.

2.8 Operador módulo

El *operador módulo* trabaja con enteros y obtiene el resto de la operación consistente en dividir el primer operando por el segundo. En Python, el operador módulo es un signo de porcentaje (`%`). La sintaxis es la misma que se usa para los demás operadores:

```
>>> quotient = 7 // 3
>>> print(quotient)
2
>>> remainder = 7 % 3
>>> print(remainder)
1
```

Así que 7 dividido por 3 es 2 y nos sobra 1.

El operador módulo resulta ser sorprendentemente útil. Por ejemplo, puedes comprobar si un número es divisible por otro—si `x % y` es cero, entonces `x` es divisible por `y`.

También se puede extraer el dígito más a la derecha de los que componen un número. Por ejemplo, `x % 10` obtiene el dígito que está más a la derecha de `x` (en base 10). De forma similar, `x % 100` obtiene los dos últimos dígitos.

2.9 Operaciones con cadenas

El operador + funciona con las cadenas, pero no realiza una suma en el sentido matemático. En vez de eso, realiza una *concatenación*, que quiere decir que une ambas cadenas, enlazando el final de la primera con el principio de la segunda. Por ejemplo:

```
>>> primero = 10
>>> segundo = 15
>>> print(primero+segundo)
25
>>> primero = '100'
>>> segundo = '150'
>>> print(primero + segundo)
100150
```

La salida de este programa es `100150`.

El operador * también trabaja con cadenas multiplicando el contenido de una cadena por un entero. Por ejemplo:

```
>>> primero = 'Test '
>>> second = 3
>>> print(primero * second)
Test Test Test
```

2.10 Petición de información al usuario

A veces necesitaremos que sea el usuario quien nos proporcione el valor para una variable, a través del teclado. Python proporciona una función interna llamada `input` que recibe la entrada desde el teclado. Cuando se llama a esa función, el programa se detiene y espera a que el usuario escriba algo. Cuando el usuario pulsa `Retorno` o `Intro`, el programa continúa y `input` devuelve como una cadena aquello que el usuario escribió.

```
>>> entrada = input()
Cualquier cosa ridícula
>>> print(entrada)
Cualquier cosa ridícula
```

Antes de recibir cualquier dato desde el usuario, es buena idea escribir un mensaje explicándole qué debe introducir. Se puede pasar una cadena a `input`, que será mostrada al usuario antes de que el programa se detenga para recibir su entrada:

```
>>> nombre = input('¿Cómo te llamas?\n')
¿Cómo te llamas?
Chuck
>>> print(nombre)
Chuck
```

La secuencia `\n` al final del mensaje representa un *newline*, que es un carácter especial que provoca un salto de línea. Por eso la entrada del usuario aparece debajo de nuestro mensaje.

Si esperas que el usuario escriba un entero, puedes intentar convertir el valor de retorno a `int` usando la función `int()`:

```
>>> prompt = '¿Cual.... es la velocidad de vuelo de una golondrina sin carga?\n'
>>> velocidad = input(prompt)
¿Cual.... es la velocidad de vuelo de una golondrina sin carga?
17
>>> int(velocidad)
17
>>> int(velocidad) + 5
22
```

Pero si el usuario escribe algo que no sea una cadena de dígitos, obtendrás un error:

```
>>> velocidad = input(prompt)
¿Cual.... es la velocidad de vuelo de una golondrina sin carga?
¿Te refieres a una golondrina africana o a una europea?
>>> int(velocidad)
ValueError: invalid literal for int()
```

Veremos cómo controlar este tipo de errores más adelante.

2.11 Comentarios

A medida que los programas se van volviendo más grandes y complicados, se vuelven más difíciles de leer. Los lenguajes formales son densos, y a menudo es complicado mirar un trozo de código e imaginarse qué es lo que hace, o por qué.

Por eso es buena idea añadir notas a tus programas, para explicar en un lenguaje normal qué es lo que el programa está haciendo. Estas notas reciben el nombre de *comentarios*, y en Python comienzan con el símbolo `#`:

```
# calcula el porcentaje de hora transcurrido
porcentaje = (minuto * 100) / 60
```

En este caso, el comentario aparece como una línea completa. Pero también puedes poner comentarios al final de una línea

```
porcentaje = (minuto * 100) / 60      # porcentaje de una hora
```

Todo lo que va desde `#` hasta el final de la línea es ignorado—no afecta para nada al programa.

Las comentarios son más útiles cuando documentan características del código que no resultan obvias. Es razonable asumir que el lector puede descifrar *qué* es lo que el código hace; es mucho más útil explicarle *por qué*.

Este comentario es redundante con el código e inútil:

```
v = 5      # asigna 5 a v
```

Este comentario contiene información útil que no está en el código:

```
v = 5      # velocidad en metros/segundo.
```

Elegir nombres adecuados para las variables puede reducir la necesidad de comentarios, pero los nombres largos también pueden ocasionar que las expresiones complejas sean difíciles de leer, así que hay que conseguir una solución de compromiso.

2.12 Elección de nombres de variables mnemónicos

Mientras sigas las sencillas reglas de nombrado de variables y evites las palabras reservadas, dispondrás de una gran variedad de opciones para poner nombres a tus variables. Al principio, esa diversidad puede llegar a resultarte confusa, tanto al leer un programa como al escribir el tuyo propio. Por ejemplo, los tres programas siguientes son idénticos en cuanto a la función que realizan, pero muy diferentes cuando los lees e intentas entenderlos.

```
a = 35.0
b = 12.50
c = a * b
print(c)
```

```
horas = 35.0
tarifa = 12.50
salario = horas * tarifa
print(salario)
```

```
x1q3z9ahd = 35.0
x1q3z9afd = 12.50
x1q3p9afd = x1q3z9ahd * x1q3z9afd
print(x1q3p9afd)
```

El intérprete de Python ve los tres programas como *exactamente idénticos*, pero los humanos ven y asimilan estos programas de forma bastante diferente. Los humanos entenderán más rápidamente el *objetivo* del segundo programa, ya que el programador ha elegido nombres de variables que reflejan lo que pretendía de acuerdo al contenido que iba almacenar en cada variable.

Esa sabia elección de nombres de variables se denomina utilizar "nombres de variables mnemónicos". La palabra *mnemónico*[2] significa "que ayuda a memorizar".

[2]Consulta https://es.wikipedia.org/wiki/Mnemonico para obtener una descripción detallada de la palabra "mnemónico".

Elegimos nombres de variables mnemónicos para ayudarnos a recordar por qué creamos las variables al principio.

A pesar de que todo esto parezca estupendo, y de que sea una idea muy buena usar nombres de variables mnemónicos, ese tipo de nombres pueden interponerse en el camino de los programadores novatos a la hora de analizar y comprender el código. Esto se debe a que los programadores principiantes no han memorizado aún las palabras reservadas (sólo hay 33), y a veces variables con nombres que son demasiado descriptivos pueden llegar a parecerles parte del lenguaje y no simplemente nombres de variable bien elegidos[3].

Echa un vistazo rápido al siguiente código de ejemplo en Python, que se mueve en bucle a través de un conjunto de datos. Trataremos los bucles pronto, pero por ahora tan sólo trata de entender su significado:

```
for word in words:
    print(word)
```

¿Qué ocurre aquí? ¿Cuáles de las piezas (for, word, in, etc.) son palabras reservadas y cuáles son simplemente nombres de variables? ¿Acaso Python comprende de un modo básico la noción de palabras (`words`)? Los programadores novatos tienen problemas separando qué parte del código *debe* mantenerse tal como está en este ejemplo y qué partes son simplemente elección del programador.

El código siguiente es equivalente al de arriba:

```
for slice in pizza:
    print(slice)
```

Para los principiantes es más fácil estudiar este código y saber qué partes son palabras reservadas definidas por Python y qué partes son simplemente nombres de variables elegidas por el programador. Está bastante claro que Python no entiende nada de pizza ni de porciones, ni del hecho de que una pizza consiste en un conjunto de una o más porciones.

Pero si nuestro programa lo que realmente va a hacer es leer datos y buscar palabras en ellos, `pizza` y `porción` son nombres muy poco mnemónicos. Elegirlos como nombres de variables distrae del propósito real del programa.

Dentro de muy poco tiempo, conocerás las palabras reservadas más comunes, y empezarás a ver cómo esas palabras reservadas resaltan sobre las demás:

Las partes del código que están definidas por Python (`for`, `in`, `print`, y `:`) están en negrita, mientras que las variables elegidas por el programador (`word` y `words`) no lo están. Muchos editores de texto son conscientes de la sintaxis de Python y colorearán las palabras reservadas de forma diferente para darte pistas que te permitan mantener tus variables y las palabras reservadas separados. Dentro de poco empezarás a leer Python y podrás determinar rápidamente qué es una variable y qué es una palabra reservada.

[3] El párrafo anterior se refiere más bien a quienes eligen nombres de variables en inglés, ya que todas las palabras reservadas de Python coinciden con palabras propias de ese idioma (Nota del trad.)

2.13 Depuración

En este punto, el error de sintaxis que es más probable que cometas será intentar utilizar nombres de variables no válidos, como `class` y `yield`, que son palabras clave, o `odd~job` y `US$`, que contienen caracteres no válidos.

Si pones un espacio en un nombre de variable, Python cree que se trata de dos operandos sin ningún operador:

```
>>> bad name = 5
SyntaxError: invalid syntax

>>> month = 09
  File "<stdin>", line 1
    month = 09
            ^
SyntaxError: invalid token
```

Para la mayoría de errores de sintaxis, los mensajes de error no ayudan mucho. Los mensajes más comunes son `SyntaxError: invalid syntax` y `SyntaxError: invalid token`, ninguno de los cuales resulta muy informativo.

El runtime error (error en tiempo de ejecución) que es más probable que obtengas es un "use before def" (uso antes de definir); que significa que estás intentando usar una variable antes de que le hayas asignado un valor. Eso puede ocurrir si escribes mal el nombre de la variable:

```
>>> principal = 327.68
>>> interest = principle * rate
NameError: name 'principle' is not defined
```

Los nombres de las variables son sensibles a mayúsculas, así que `LaTeX` no es lo mismo que `latex`.

En este punto, la causa más probable de un error semántico es el orden de las operaciones. Por ejemplo, para evaluar $\frac{1}{2\pi}$, puedes sentirte tentado a escribir

```
>>> 1.0 / 2.0 * pi
```

Pero la división se evalúa antes, ¡así que obtendrás $\pi/2$, que no es lo mismo! No hay forma de que Python sepa qué es lo que querías escribir exactamente, así que en este caso no obtienes un mensaje de error; simplemente obtienes una respuesta incorrecta.

2.14 Glosario

asignación Una sentencia que asigna un valor a una variable.

cadena Un tipo que representa secuencias de caracteres.

concatenar Unir dos operandos, uno a continuación del otro.

comentario Información en un programa que se pone para otros programadores (o para cualquiera que lea el código fuente), y no tiene efecto alguno en la ejecución del programa.

división entera La operación que divide dos números y trunca la parte fraccionaria.

entero Un tipo que representa números enteros.

evaluar Simplificar una expresión realizando las operaciones en orden para obtener un único valor.

expresión Una combinación de variables, operadores y valores que representan un único valor resultante.

mnemónico Una ayuda para memorizar. A menudo damos nombres mnemónicos a las variables para ayudarnos a recordar qué está almacenado en ellas.

palabra clave Una palabra reservada que es usada por el compilador para analizar un programa; no se pueden usar palabres clave como `if`, `def`, y `while` como nombres de variables.

punto flotante Un tipo que representa números con parte decimal.

operador Un símbolo especial que representa un cálculo simple, como suma, multiplicación o concatenación de cadenas.

operador módulo Un operador, representado por un signo de porcentaje (`%`), que funciona con enteros y obtiene el resto cuando un número es dividido por otro.

operando Uno de los valores con los cuales un operador opera.

reglas de precedencia El conjunto de reglas que gobierna el orden en el cual son evaluadas las expresiones que involucran a múltiples operadores.

sentencia Una sección del código que representa un comando o acción. Hasta ahora, las únicas sentencias que hemos visto son asignaciones y sentencias print.

tipo Una categoría de valores. Los tipos que hemos visto hasta ahora son enteros (tipo `int`), números en punto flotante (tipo `float`), y cadenas (tipo `str`).

valor Una de las unidades básicas de datos, como un número o una cadena, que un programa manipula.

variable Un nombre que hace referencia a un valor.

2.15 Ejercicios

Ejercicio 2: Escribe un programa que use `input` para pedirle al usuario su nombre y luego darle la bienvenida.

```
Introduzca tu nombre: Chuck
Hola, Chuck
```

Ejercicio 3: Escribe un programa para pedirle al usuario el número de horas y la tarifa por hora para calcular el salario bruto.

```
Introduzca Horas: 35
Introduzca Tarifa: 2.75
Salario: 96.25
```

Por ahora no es necesario preocuparse de que nuestro salario tenga exactamente dos dígitos después del punto decimal. Si quieres, puedes probar la función interna de Python `round` para redondear de forma adecuada el salario resultante a dos dígitos decimales.

Ejercicio 4: Asume que ejecutamos las siguientes sentencias de asignación:

```
ancho = 17
alto = 12.0
```

Para cada una de las expresiones siguientes, escribe el valor de la expresión y el tipo (del valor de la expresión).

1. `ancho/2`
2. `ancho/2.0`
3. `alto/3`
4. `1 + 2 * 5`

Usa el intérprete de Python para comprobar tus respuestas.

Ejercicio 5: Escribe un programa que le pida al usuario una temperatura en grados Celsius, la convierta a grados Fahrenheit e imprima por pantalla la temperatura convertida.

Chapter 3

Ejecución condicional

3.1 Expresiones booleanas

Una *expresión booleana* es aquella que puede ser verdadera (`True`) o falsa (`False`). Los ejemplos siguientes usan el operador `==`, que compara dos operandos y devuelve `True` si son iguales y `False` en caso contrario:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

`True` y `False` son valores especiales que pertenecen al tipo `bool (booleano)`; no son cadenas:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

El operador `==` es uno de los *operadores de comparación*; los demás son:

```
x != y               # x es distinto de y
x > y                # x es mayor que y
x < y                # x es menor que y
x >= y               # x es mayor o igual que y
x <= y               # x es menor o igual que y
x is y               # x es lo mismo que y
x is not y           # x no es lo mismo que y
```

A pesar de que estas operaciones probablemente te resulten familiares, los símbolos en Python son diferentes de los símbolos matemáticos que se usan para realizar las mismas operaciones. Un error muy común es usar sólo un símbolo igual (`=`) en vez del símbolo de doble igualdad (`==`). Recuerda que `=` es un operador de asignación, y `==` es un operador de comparación. No existe algo como `=<` o `=>`.

3.2 Operadores lógicos

Existen tres *operadores lógicos*: `and (y)`, `or (o)`, y `not (no)`. El significado semántico de estas operaciones es similar a su significado en inglés. Por ejemplo,

`x > 0 and x < 10`

es verdadero sólo cuando `x` es mayor que 0 *y* menor que 10.

`n%2 == 0 or n%3 == 0` es verdadero si *cualquiera* de las condiciones es verdadera, es decir, si el número es divisible por 2 *o* por 3.

Finalmente, el operador `not` niega una expresión booleana, de modo que `not (x > y)` es verdadero si `x > y` es falso; es decir, si `x` es menor o igual que `y`.

Estrictamente hablando, los operandos de los operadores lógicos deberían ser expresiones booleanas, pero Python no es muy estricto. Cualquier número distinto de cero se interpreta como "verdadero."

```
>>> 17 and True
True
```

Esta flexibilidad puede ser útil, pero existen ciertas sutilezas en ese tipo de uso que pueden resultar confusas. Es posible que prefieras evitar usarlo de este modo hasta que estés bien seguro de lo que estás haciendo.

3.3 Ejecución condicional

Para poder escribir programas útiles, casi siempre vamos a necesitar la capacidad de comprobar condiciones y cambiar el comportamiento del programa de acuerdo a ellas. Las `sentencias condicionales` nos proporcionan esa capacidad. La forma más sencilla es la sentencia `if`:

```
if x > 0 :
    print('x es positivo')
```

La expresión booleana después de la sentencia `if` recibe el nombre de *condición*. La sentencia `if` se finaliza con un carácter de dos-puntos (:) y la(s) línea(s) que van detrás de la sentencia if van indentadas[1] (es decir, llevan una tabulación o varios espacios en blanco al principio).

Si la condición lógica es verdadera, la sentencia indentada será ejecutada. Si la condición es falsa, la sentencia indentada será omitida.

La sentencia `if` tiene la misma estructura que la definición de funciones o los bucles `for`[2]. La sentencia consiste en una línea de encabezado que termina con el carácter dos-puntos (:) seguido por un bloque indentado. Las sentencias de este tipo reciben el nombre de *sentencias compuestas*, porque se extienden a lo largo de varias líneas.

[1] el término correcto en español sería "sangradas", pero en el mundillo de la programación se suele decir que las líneas van "indentadas" (Nota del trad.)

[2] Estudiaremos las funciones en el capítulo 4 y los bucles en el capítulo 5.

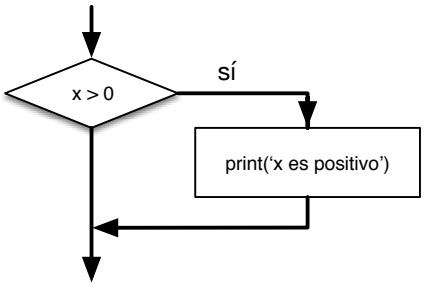

Figure 3.1: If Logic

No hay límite en el número de sentencias que pueden aparecer en el cuerpo, pero debe haber al menos una. Ocasionalmente, puede resultar útil tener un cuerpo sin sentencias (normalmente como emplazamiento reservado para código que no se ha escrito aún). En ese caso, se puede usar la sentencia `pass`, que no hace nada.

```
if x < 0 :
    pass          # ¡necesito gestionar los valores negativos!
```

Si introduces una sentencia `if` en el intérprete de Python, el prompt cambiará su aspecto habitual por puntos suspensivos, para indicar que estás en medio de un bloque de sentencias, como se muestra a continuación:

```
>>> x = 3
>>> if x < 10:
...     print('Pequeño')
...
Pequeño
>>>
```

Al usar el intérprete de Python, debe dejar una línea en blanco al final de un bloque, de lo contrario Python devolverá un error:

```
>>> x = 3
>>> if x < 10:
...     print('Pequeño')
... print('Hecho')
  File "<stdin>", line 3
    print('Hecho')
        ^
SyntaxError: invalid syntax
```

No es necesaria una línea en blanco al final de un bloque de instrucciones al escribir y ejecutar un script, pero puede mejorar la legibilidad de su código.

3.4 Ejecución alternativa

La segunda forma de la sentencia `if` es la *ejecución alternativa*, en la cual existen dos posibilidades y la condición determina cual de ellas será ejecutada. La sintaxis es similar a ésta:

```
if x%2 == 0 :
    print('x es par')
else :
    print('x es impar')
```

Si al dividir `x` por 2 obtenemos como resto 0, entonces sabemos que `x` es par, y el programa muestra un mensaje a tal efecto. Si esa condición es falsa, se ejecuta el segundo conjunto de sentencias.

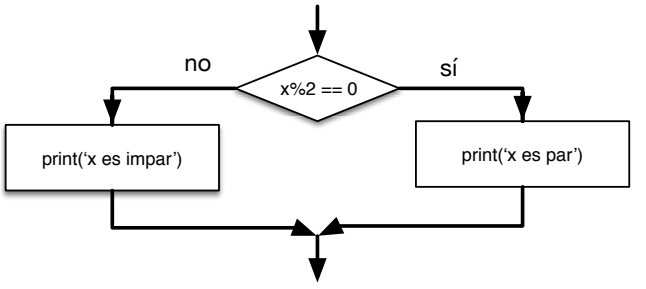

Figure 3.2: If-Then-Else Logic

Dado que la condición debe ser obligatoriamente verdadera o falsa, solamente una de las alternativas será ejecutada. Las alternativas reciben el nombre de *ramas*, dado que se trata de ramificaciones en el flujo de la ejecución.

3.5 Condicionales encadenados

Algunas veces hay más de dos posibilidades, de modo que necesitamos más de dos ramas. Una forma de expresar una operación como ésa es usar un *condicional encadenado*:

```
if x < y:
    print('x es menor que y')
elif x > y:
    print('x es mayor que y')
else:
    print('x e y son iguales')
```

`elif` es una abreviatura para "else if". En este caso también será ejecutada únicamente una de las ramas.

No hay un límite para el número de sentencias `elif`. Si hay una clausula `else`, debe ir al final, pero tampoco es obligatorio que ésta exista.

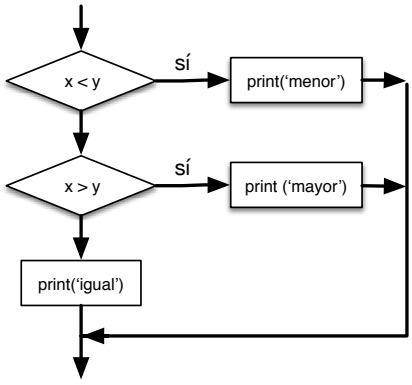

Figure 3.3: If-Then-ElseIf Logic

```
if choice == 'a':
    print('Respuesta incorrecta')
elif choice == 'b':
    print('Respuesta correcta')
elif choice == 'c':
    print('Casi, pero no es correcto')
```

Cada condición es comprobada en orden. Si la primera es falsa, se comprueba la siguiente y así con las demás. Si una de ellas es verdadera, se ejecuta la rama correspondiente, y la sentencia termina. Incluso si hay más de una condición que sea verdadera, sólo se ejecuta la primera que se encuentra.

3.6 Condicionales anidados

Un condicional puede también estar anidado dentro de otro. Podríamos haber escrito el ejemplo anterior de las tres ramas de este modo:

```
if x == y:
    print('x e y son iguales')
else:
    if x < y:
        print('x es menor que y')
    else:
        print('x es mayor que y')
```

El condicional exterior contiene dos ramas. La primera rama ejecuta una sentencia simple. La segunda contiene otra sentencia `if`, que tiene a su vez sus propias dos ramas. Esas dos ramas son ambas sentencias simples, pero podrían haber sido sentencias condicionales también.

A pesar de que el indentado de las sentencias hace que la estructura esté clara, los *condicionales anidados* pueden volverse difíciles de leer rápidamente. En general, es buena idea evitarlos si se puede.

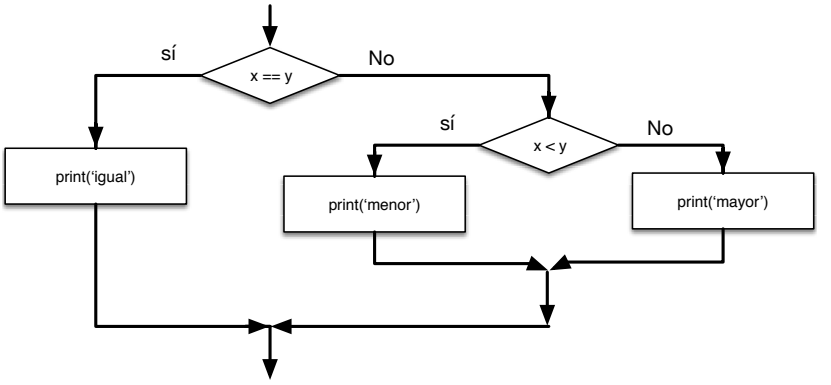

Figure 3.4: Nested If Statements

Los operadores lógicos a menudo proporcionan un modo de simplificar las sentencias condicionales anidadas. Por ejemplo, el código siguiente puede ser reescrito usando un único condicional:

```
if 0 < x:
    if x < 10:
        print('x es un número positivo con un sólo dígito.')
```

La sentencia `print` se ejecuta solamente si se cumplen las dos condiciones anteriores, así que en realidad podemos conseguir el mismo efecto con el operador `and`:

```
if 0 < x and x < 10:
    print('x es un número positivo con un sólo dígito.')
```

3.7 Captura de excepciones usando try y except

Anteriormente vimos un fragmento de código donde usábamos las funciones `input` e `int` para leer y analizar un número entero introducido por el usuario. También vimos lo poco seguro que podía llegar a resultar hacer algo así:

```
>>> velocidad = input(prompt)
¿Cual.... es la velocidad de vuelo de una golondrina sin carga?
¿Te refieres a una golondrina africana o a una europea?
>>> int(velocidad)
ValueError: invalid literal for int() with base 10:
>>>
```

Cuando estamos trabajando con el intérprete de Python, tras el error simplemente nos aparece de nuevo el prompt, así que pensamos "¡epa, me he equivocado!", y continuamos con la siguiente sentencia.

Sin embargo, si se escribe ese código en un script de Python y se produce el error, el script se detendrá inmediatamente, y mostrará un "traceback". No ejecutará la siguiente sentencia.

He aquí un programa de ejemplo para convertir una temperatura desde grados Fahrenheit a grados Celsius:

```
ent = input('Introduzca la Temperatura Fahrenheit:')
fahr = float(ent)
cel = (fahr - 32.0) * 5.0 / 9.0
print(cel)

# Code: http://www.py4e.com/code3/fahren.py
```

Si ejecutamos este código y le damos una entrada no válida, simplemente fallará con un mensaje de error bastante antipático:

```
python fahren.py
Introduzca la Temperatura Fahrenheit:72
22.2222222222


python fahren.py
Introduzca la Temperatura Fahrenheit:fred
Traceback (most recent call last):
  File "fahren.py", line 2, in <module>
    fahr = float(ent)
ValueError: invalid literal for float(): fred
```

Existen estructuras de ejecución condicional dentro de Python para manejar este tipo de errores esperados e inesperados, llamadas "try / except". La idea de `try` y `except` es que si se sabe que cierta secuencia de instrucciones puede generar un problema, sea posible añadir ciertas sentencias para que sean ejecutadas en caso de error. Estas sentencias extras (el bloque except) serán ignoradas si no se produce ningún error.

Puedes pensar en la característica `try` y `except` de Python como una "póliza de seguros" en una secuencia de sentencias.

Se puede reescribir nuestro conversor de temperaturas de esta forma:

```
ent = input('Introduzca la Temperatura Fahrenheit:')
try:
    fahr = float(ent)
    cel = (fahr - 32.0) * 5.0 / 9.0
    print(cel)
except:
    print('Por favor, introduzca un número')

# Code: http://www.py4e.com/code3/fahren2.py
```

Python comienza ejecutando la secuencia de sentencias del bloque `try`. Si todo va bien, se saltará todo el bloque `except` y terminará. Si ocurre una excepción dentro del bloque `try`, Python saltará fuera de ese bloque y ejecutará la secuencia de sentencias del bloque `except`.

```
python fahren2.py
Introduzca la Temperatura Fahrenheit:72
22.2222222222


python fahren2.py
Introduzca la Temperatura Fahrenheit:fred
Por favor, introduzca un número
```

Gestionar una excepción con una sentencia `try` recibe el nombre de *capturar* una excepción. En este ejemplo, la clausula `except` muestra un mensaje de error. En general, capturar una excepción te da la oportunidad de corregir el problema, volverlo a intentar o, al menos, terminar el programa con elegancia.

3.8 Evaluación en cortocircuito de expresiones lógicas

Cuando Python está procesando una expresión lógica, como `x >= 2 and (x/y) > 2`, evalúa la expresión de izquierda a derecha. Debido a la definición de `and`, si `x` es menor de 2, la expresión `x >= 2` resulta ser `falsa`, de modo que la expresión completa ya va a resultar `falsa`, independientemente de si `(x/y) > 2` se evalúa como `verdadera` o `falsa`.

Cuando Python detecta que no se gana nada evaluando el resto de una expresión lógica, detiene su evaluación y no realiza el cálculo del resto de la expresión. Cuando la evaluación de una expresión lógica se detiene debido a que ya se conoce el valor final, eso es conocido como *cortocircuitar* la evaluación.

A pesar de que esto pueda parecer hilar demasiado fino, el funcionamiento en cortocircuito nos descubre una ingeniosa técnica conocida como *patrón guardián*. Examina la siguiente secuencia de código en el intérprete de Python:

```
>>> x = 6
>>> y = 2
>>> x >= 2 and (x/y) > 2
True
>>> x = 1
>>> y = 0
>>> x >= 2 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and (x/y) > 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

La tercera operación ha fallado porque Python intentó evaluar `(x/y)` e `y` era cero, lo cual provoca un runtime error (error en tiempo de ejecución). Pero el segundo

ejemplo *no* falló, porque la primera parte de la expresión `x >= 2` fue evaluada como `falsa`, así que `(x/y)` no llegó a ejecutarse debido a la regla del *cortocircuito*, y no se produjo ningún error.

Es posible construir las expresiones lógicas colocando estratégicamente una evaluación como *guardián* justo antes de la evaluación que podría causar un error, como se muestra a continuación:

```
>>> x = 1
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x >= 2 and (x/y) > 2 and y != 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

En la primera expresión lógica, `x >= 2` es `falsa`, así que la evaluación se detiene en el `and`. En la segunda expresión lógica, `x >= 2` es `verdadera`, pero `y != 0` es `falsa`, de modo que nunca se alcanza `(x/y)`.

En la tercera expresión lógica, el `y != 0` va *después* del cálculo de `(x/y)`, de modo que la expresión falla con un error.

En la segunda expresión, se dice que `y != 0` actúa como *guardián* para garantizar que sólo se ejecute `(x/y)` en el caso de que `y` no sea cero.

3.9 Depuración

Los "traceback" que Python muestra cuando se produce un error contienen un montón de información, pero pueden resultar abrumadores. Las partes más útiles normalmente son:

- Qué tipo de error se ha producido, y
- Dónde ha ocurrido.

Los errores de sintaxis (syntax errors), normalmente son fáciles de localizar, pero a veces tienen trampa. Los errores debido a espacios en blanco pueden ser complicados, ya que los espacios y las tabulaciones son invisibles, y solemos ignorarlos.

```
>>> x = 5
>>>  y = 6
  File "<stdin>", line 1
    y = 6
    ^
IndentationError: unexpected indent
```

En este ejemplo, el problema es que la segunda línea está indentada por un espacio. Pero el mensaje de error apunta a `y`, lo cual resulta engañoso. En general, los mensajes de error indican dónde se ha descubierto el problema, pero el error real podría estar en el código previo, a veces en alguna línea anterior.

Ocurre lo mismo con los errores en tiempo de ejecución (runtime errors). Supón que estás tratando de calcular una relación señal-ruido en decibelios. La fórmula es $SNR_{db} = 10 \log_{10}(P_{senal}/P_{ruido})$. En Python, podrías escribir algo como esto:

```
import math
int_senal = 9
int_ruido = 10
relacion = int_senal / int_ruido
decibelios = 10 * math.log10(relacion)
print(decibelios)

# Code: http://www.py4e.com/code3/snr.py
```

Pero cuando lo haces funcionar, obtienes un mensaje de error[3]:

```
Traceback (most recent call last):
  File "snr.py", line 5, in ?
    decibelios = 10 * math.log10(relacion)
OverflowError: math range error
```

El mensaje de error apunta a la línea 5, pero no hay nada incorrecto en ese línea. Para encontrar el error real, puede resultar útil mostrar en pantalla el valor de `relacion`, que resulta ser 0. El problema está en la línea 4, ya que al dividir dos enteros se realiza una división entera. La solución es representar la intensidad de la señal y la intensidad del ruido con valores en punto flotante.

En general, los mensajes de error te dicen dónde se ha descubierto el problema, pero a menudo no es ahí exactamente donde se ha producido.

3.10 Glosario

condición La expresión booleana en una sentencia condicional que determina qué rama será ejecutada.

condicional anidado Una sentencia condicional que aparece en una de las ramas de otra sentencia condicional.

condicional encadenado Una sentencia condicional con una serie de ramas alternativas.

[3] En Python 3.0, ya no se produce el mensaje de error; el operador de división realiza división en punto flotante incluso con operandos enteros.

cortocircuito Cuando Python va evaluando una expresión lógica por tramos y detiene el proceso de evaluación debido a que ya conoce el valor final que va a tener el resultado sin necesidad de evaluar el resto de la expresión.

cuerpo La secuencia de sentencias en el interior de una sentencia compuesta.

expresión booleana Un expresión cuyo valor puede ser o bien `Verdadero` o bien `Falso`.

operadores de comparación Uno de los operadores que se utiliza para comparar dos operandos: `==`, `!=`, `>`, `<`, `>=`, y `<=`.

operador lógico Uno de los operadores que se combinan en las expresiones booleanas: `and`, `or`, y `not`.

patrón guardián Cuando construimos una expresión lógica con comparaciones adicionales para aprovecharnos del funcionamiento en cortocircuito.

rama Una de las secuencias alternativas de sentencias en una sentencia condicional.

sentencia compuesta Una sentencia que consiste en un encabezado y un cuerpo. El encabezado termina con dos-puntos (:). El cuerpo está indentado con relación al encabezado.

sentencia condicional Una sentencia que controla el flujo de ejecución, dependiendo de cierta condición.

traceback Una lista de las funciones que se están ejecutando, que se muestra en pantalla cuando se produce una excepción.

3.11 Ejercicios

Ejercicio 1: Reescribe el programa del cálculo del salario para darle al empleado 1.5 veces la tarifa horaria para todas las horas trabajadas que excedan de 40.

```
Introduzca las Horas: 45
Introduzca la Tarifa por hora: 10
Salario: 475.0
```

Ejercicio 2: Reescribe el programa del salario usando `try` y `except`, de modo que el programa sea capaz de gestionar entradas no numéricas con elegancia, mostrando un mensaje y saliendo del programa. A continuación se muestran dos ejecuciones del programa:

```
Introduzca las Horas: 20
Introduzca la Tarifa por hora: nueve
Error, por favor introduzca un número

Introduzca las Horas: cuarenta
Error, por favor introduzca un número
```

Ejercicio 3: Escribe un programa que solicite una puntuación entre 0.0 y 1.0. Si la puntuación está fuera de ese rango, muestra un mensaje de error. Si la puntuación está entre 0.0 y 1.0, muestra la calificación usando la tabla siguiente:

```
Puntuación Calificación
>= 0.9     Sobresaliente
>= 0.8     Notable
>= 0.7     Bien
>= 0.6     Suficiente
< 0.6      Insuficiente

Introduzca puntuación: 0.95
Sobresaliente

Introduzca puntuación: perfecto
Puntuación incorrecta

Introduzca puntuación: 10.0
Puntuación incorrecta

Introduzca puntuación: 0.75
Bien

Introduzca puntuación: 0.5
Insuficiente
```

Ejecuta el programa repetidamente, como se muestra arriba, para probar con varios valores de entrada diferentes.

Chapter 4

Funciones

4.1 Llamadas a funciones

En el contexto de la programación, una *función* es una secuencia de sentencias que realizan una operación y que reciben un nombre. Cuando se define una función, se especifica el nombre y la secuencia de sentencias. Más adelante, se puede "llamar" a la función por ese nombre. Ya hemos visto un ejemplo de una *llamada a una función*:

```
>>> type(32)
<class 'int'>
```

El nombre de la función es `type`. La expresión entre paréntesis recibe el nombre de *argumento* de la función. El argumento es un valor o variable que se pasa a la función como parámetro de entrada. El resultado de la función `type` es el tipo del argumento.

Es habitual decir que una función "toma" (o recibe) un argumento y "retorna" (o devuelve) un resultado. El resultado se llama *valor de retorno*.

4.2 Funciones internas

Python proporciona un número importante de funciones internas, que pueden ser usadas sin necesidad de tener que definirlas previamente. Los creadores de Python han escrito un conjunto de funciones para resolver problemas comunes y las han incluido en Python para que las podamos utilizar.

Las funciones `max` y `min` nos darán respectivamente el valor mayor y menor de una lista:

```
>>> max('¡Hola, mundo!')
'u'
>>> min('¡Hola, mundo!')
' '
>>>
```

La función `max` nos dice cuál es el "carácter más grande" de la cadena (que resulta ser la letra "u"), mientras que la función `min` nos muestra el carácter más pequeño (que en ese caso es un espacio).

Otra función interna muy común es `len`, que nos dice cuántos elementos hay en su argumento. Si el argumento de `len` es una cadena, nos devuelve el número de caracteres que hay en la cadena.

```
>>> len('Hola, mundo')
11
>>>
```

Estas funciones no se limitan a buscar en cadenas. Pueden operar con cualquier conjunto de valores, como veremos en los siguientes capítulos.

Se deben tratar los nombres de las funciones internas como si fueran palabras reservadas (es decir, evita usar "max" como nombre para una variable).

4.3 Funciones de conversión de tipos

Python también proporciona funciones internas que convierten valores de un tipo a otro. La función `int` toma cualquier valor y lo convierte en un entero, si puede, o se queja si no puede:

```
>>> int('32')
32
>>> int('Hola')
ValueError: invalid literal for int() with base 10: 'Hola'
```

`int` puede convertir valores en punto flotante a enteros, pero no los redondea; simplemente corta y descarta la parte decimal:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

`float` convierte enteros y cadenas en números de punto flotante:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

Finalmente, `str` convierte su argumento en una cadena:

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

4.4 Funciones matemáticas

Python tiene un módulo matemático (`math`), que proporciona la mayoría de las funciones matemáticas habituales. Antes de que podamos utilizar el módulo, deberemos importarlo:

```
>>> import math
```

Esta sentencia crea un *objeto módulo* llamado math. Si se imprime el objeto módulo, se obtiene cierta información sobre él:

```
>>> print(math)
<module 'math' (built-in)>
```

El objeto módulo contiene las funciones y variables definidas en el módulo. Para acceder a una de esas funciones, es necesario especificar el nombre del módulo y el nombre de la función, separados por un punto (también conocido en inglés como *peŕiod*). Este formato recibe el nombre de *notación punto*.

```
>>> relacion = int_senal / int_ruido
>>> decibelios = 10 * math.log10(relacion)

>>> radianes = 0.7
>>> altura = math.sin(radianes)
```

El primer ejemplo calcula el logaritmo en base 10 de la relación señal-ruido. El módulo math también proporciona una función llamada `log` que calcula logaritmos en base `e`.

El segundo ejemplo calcula el seno de la variable `radianes`. El nombre de la variable es una pista de que `sin` y las otras funciones trigonométricas (`cos`, `tan`, etc.) toman argumentos en radianes. Para convertir de grados a radianes, hay que dividir por 360 y multiplicar por 2π:

```
>>> grados = 45
>>> radianes = grados / 360.0 * 2 * math.pi
>>> math.sin(radianes)
0.7071067811865476
```

La expresión `math.pi` toma la variable `pi` del módulo math. El valor de esa variable es una aproximación de π, con una precisión de unos 15 dígitos.

Si sabes de trigonometría, puedes comprobar el resultado anterior, comparándolo con la raíz cuadrada de dos dividida por dos:

```
>>> math.sqrt(2) / 2.0
0.7071067811865476
```

4.5 Números aleatorios

A partir de las mismas entradas, la mayoría de los programas generarán las mismas salidas cada vez, que es lo que llamamos comportamiento *determinista*. El determinismo normalmente es algo bueno, ya que esperamos que la misma operación nos proporcione siempre el mismo resultado. Para ciertas aplicaciones, sin embargo, querremos que el resultado sea impredecible. Los juegos son el ejemplo obvio, pero hay más.

Conseguir que un programa sea realmente no-determinista no resulta tan fácil, pero hay modos de hacer que al menos lo parezca. Una de ellos es usar *algoritmos* que generen números *pseudoaleatorios*. Los números pseudoaleatorios no son verdaderamente aleatorios, ya que son generados por una operación determinista, pero si sólo nos fijamos en los números resulta casi imposible distinguirlos de los aleatorios de verdad.

El módulo `random` proporciona funciones que generan números pseudoaleatorios (a los que simplemente llamaremos "aleatorios" de ahora en adelante).

La función `random` devuelve un número flotante aleatorio entre 0.0 y 1.0 (incluyendo 0.0, pero no 1.0). Cada vez que se llama a `random`, se obtiene el número siguiente de una larga serie. Para ver un ejemplo, ejecuta este bucle:

```
import random

for i in range(10):
    x = random.random()
    print(x)
```

Este programa produce la siguiente lista de 10 números aleatorios entre 0.0 y hasta (pero no incluyendo) 1.0.

```
0.11132867921152356
0.5950949227890241
0.04820265884996877
0.841003109276478
0.997914947094958
0.04842330803368111
0.7416295948208405
0.510535245390327
0.27447040171978143
0.028511805472785867
```

Ejercicio 1: Ejecuta el programa en tu sistema y observa qué números obtienes.

La función `random` es solamente una de las muchas que trabajan con números aleatorios. La función `randint` toma los parámetros `inferior` y `superior`, y devuelve un entero entre `inferior` y `superior` (incluyendo ambos extremos).

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

Para elegir un elemento de una secuencia aleatoriamente, se puede usar `choice`:

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

El módulo `random` también proporciona funciones para generar valores aleatorios de varias distribuciones continuas, incluyendo gaussiana, exponencial, gamma, y unas cuantas más.

4.6 Añadiendo funciones nuevas

Hasta ahora, sólo hemos estado usando las funciones que vienen incorporadas en Python, pero es posible añadir también funciones nuevas. Una *definición de función* especifica el nombre de una función nueva y la secuencia de sentencias que se ejecutan cuando esa función es llamada. Una vez definida una función, se puede reutilizar una y otra vez a lo largo de todo el programa.

He aquí un ejemplo:

```
def muestra_estribillo():
    print('Soy un leñador, qué alegría.')
    print('Duermo toda la noche y trabajo todo el día.')
```

`def` es una palabra clave que indica que se trata de una definición de función. El nombre de la función es `muestra_estribillo`. Las reglas para los nombres de las funciones son los mismos que para las variables: se pueden usar letras, números y algunos signos de puntuación, pero el primer carácter no puede ser un número. No se puede usar una palabra clave como nombre de una función, y se debería evitar también tener una variable y una función con el mismo nombre.

Los paréntesis vacíos después del nombre indican que esta función no toma ningún argumento. Más tarde construiremos funciones que reciban argumentos de entrada.

La primera línea de la definición de la función es llamada la *cabecera*; el resto se llama el *cuerpo*. La cabecera debe terminar con dos-puntos (:), y el cuerpo debe ir indentado. Por convención, el indentado es siempre de cuatro espacios. El cuerpo puede contener cualquier número de sentencias.

Las cadenas en la sentencia print están encerradas entre comillas. Da igual utilizar comillas simples que dobles; la mayoría de la gente prefiere comillas simples, excepto en aquellos casos en los que una comilla simple (que también se usa como apostrofe) aparece en medio de la cadena.

Si escribes una definición de función en modo interactivo, el intérprete mostrará puntos suspensivos (. . .) para informarte de que la definición no está completa:

```
>>> def muestra_estribillo():
...     print 'Soy un leñador, qué alegría.'
...     print 'Duermo toda la noche y trabajo todo el día.'
...
```

Para finalizar la función, debes introducir una línea vacía (esto no es necesario en un script).

Al definir una función se crea una variable con el mismo nombre.

```
>>> print(muestra_estribillo)
<function muestra_estribillo at 0xb7e99e9c>
>>> print(type(muestra_estribillo))
<type 'function'>
```

El valor de `muestra_estribillo` es *function object* (objeto función), que tiene como tipo "function".

La sintaxis para llamar a nuestra nueva función es la misma que usamos para las funciones internas:

```
>>> muestra_estribillo()
Soy un leñador, qué alegría.
Duermo toda la noche y trabajo todo el día.
```

Una vez que se ha definido una función, puede usarse dentro de otra. Por ejemplo, para repetir el estribillo anterior, podríamos escribir una función llamada `repite_estribillo`:

```
def repite_estribillo():
    muestra_estribillo()
    muestra_estribillo()
```

Y después llamar a `repite_estribillo`:

```
>>> repite_estribillo()
Soy un leñador, qué alegría.
Duermo toda la noche y trabajo todo el día.
Soy un leñador, qué alegría.
Duermo toda la noche y trabajo todo el día.
```

Pero en realidad la canción no es así.

4.7 Definición y usos

Reuniendo los fragmentos de código de las secciones anteriores, el programa completo sería algo como esto:

```
def muestra_estribillo():
    print('Soy un leñador, que alegría.')
    print('Duermo toda la noche y trabajo todo el día.')

def repite_estribillo():
    muestra_estribillo()
    muestra_estribillo()

repite_estribillo()

# Code: http://www.py4e.com/code3/lyrics.py
```

Este programa contiene dos definiciones de funciones: `muestra_estribillo` y `repite_estribillo`. Las definiciones de funciones son ejecutadas exactamente igual que cualquier otra sentencia, pero su resultado consiste en crear objetos del tipo función. Las sentencias dentro de cada función son ejecutadas solamente cuando se llama a esa función, y la definición de una función no genera ninguna salida.

Como ya te imaginarás, es necesario crear una función antes de que se pueda ejecutar. En otras palabras, la definición de la función debe ser ejecutada antes de que la función se llame por primera vez.

Ejercicio 2: Desplaza la última línea del programa anterior hacia arriba, de modo que la llamada a la función aparezca antes que las definiciones. Ejecuta el programa y observa qué mensaje de error obtienes.

Ejercicio 3: Desplaza la llamada de la función de nuevo hacia el final, y coloca la definición de `muestra_estribillo` después de la definición de `repite_estribillo`. ¿Qué ocurre cuando haces funcionar ese programa?

4.8 Flujo de ejecución

Para asegurarnos de que una función está definida antes de usarla por primera vez, es necesario saber el orden en que las sentencias son ejecutadas, que es lo que llamamos el *flujo de ejecución*.

La ejecución siempre comienza en la primera sentencia del programa. Las sentencias son ejecutadas una por una, en orden de arriba hacia abajo.

Las *definiciones* de funciones no alteran el flujo de la ejecución del programa, pero recuerda que las sentencias dentro de una función no son ejecutadas hasta que se llama a esa función.

Una llamada a una función es como un desvío en el flujo de la ejecución. En vez de pasar a la siguiente sentencia, el flujo salta al cuerpo de la función, ejecuta todas las sentencias que hay allí, y después vuelve al punto donde lo dejó.

Todo esto parece bastante sencillo, hasta que uno recuerda que una función puede llamar a otra. Cuando está en mitad de una función, el programa puede tener que ejecutar las sentencias de otra función. Pero cuando está ejecutando esa nueva función, ¡tal vez haya que ejecutar todavía más funciones!

Afortunadamente, Python es capaz de llevar el seguimiento de dónde se encuentra en cada momento, de modo que cada vez que completa la ejecución de una función, el programa vuelve al punto donde lo dejó en la función que había llamado a esa. Cuando esto le lleva hasta el final del programa, simplemente termina.

¿Cuál es la moraleja de esta extraña historia? Cuando leas un programa, no siempre te convendrá hacerlo de arriba a abajo. A veces tiene más sentido seguir el flujo de la ejecución.

4.9 Parámetros y argumentos

Algunas de las funciones internas que hemos visto necesitan argumentos. Por ejemplo, cuando se llama a `math.sin`, se le pasa un número como argumento. Algunas funciones necesitan más de un argumento: `math.pow` toma dos, la base y el exponente.

Dentro de las funciones, los argumentos son asignados a variables llamadas *parámetros*. A continuación mostramos un ejemplo de una función definida por el usuario que recibe un argumento:

```
def muestra_dos_veces(bruce):
    print(bruce)
    print(bruce)
```

Esta función asigna el argumento a un parámetro llamado `bruce`. Cuando la función es llamada, imprime el valor del parámetro (sea éste lo que sea) dos veces.

Esta función funciona con cualquier valor que pueda ser mostrado en pantalla.

```
>>> muestra_dos_veces('Spam')
Spam
Spam
>>> muestra_dos_veces(17)
17
17
>>> muestra_dos_veces(math.pi)
3.14159265359
3.14159265359
```

Las mismas reglas de composición que se aplican a las funciones internas, también se aplican a las funciones definidas por el usuario, de modo que podemos usar cualquier tipo de expresión como argumento para `muestra_dos_veces`:

```
>>> muestra_dos_veces('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> muestra_dos_veces(math.cos(math.pi))
-1.0
-1.0
```

El argumento es evaluado antes de que la función sea llamada, así que en los ejemplos, la expresión `Spam *4` y `math.cos(math.pi)` son evaluadas sólo una vez.

También se puede usar una variable como argumento:

```
>>> michael = 'Eric, la medio-abeja.'
>>> muestra_dos_veces(michael)
Eric, la medio-abeja.
Eric, la medio-abeja.
```

El nombre de la variable que pasamos como argumento, (`michael`) no tiene nada que ver con el nombre del parámetro (`bruce`). No importa cómo se haya llamado al valor en origen (en la llamada); dentro de `muestra_dos_veces`, siempre se llamará `bruce`.

4.10 Funciones productivas y funciones estériles

Algunas de las funciones que estamos usando, como las matemáticas, producen resultados; a falta de un nombre mejor, las llamaremos *funciones productivas* (fruitful functions). Otras funciones, como `muestra_dos_veces`, realizan una acción, pero no devuelven un valor. A esas las llamaremos *funciones estériles* (void functions).

Cuando llamas a una función productiva, casi siempre querrás hacer luego algo con el resultado; por ejemplo, puede que quieras asignarlo a una variable o usarlo como parte de una expresión:

```
x = math.cos(radians)
aurea = (math.sqrt(5) + 1) / 2
```

Cuando llamas a una función en modo interactivo, Python muestra el resultado:

```
>>> math.sqrt(5)
2.23606797749979
```

Pero en un script, si llamas a una función productiva y no almacenas el resultado de la misma en una variable, ¡el valor de retorno se desvanece en la niebla!

```
math.sqrt(5)
```

Este script calcula la raíz cuadrada de 5, pero dado que no almacena el resultado en una variable ni lo muestra, no resulta en realidad muy útil.

Las funciones estériles pueden mostrar algo en la pantalla o tener cualquier otro efecto, pero no devuelven un valor. Si intentas asignar el resultado a una variable, obtendrás un valor especial llamado `None` (nada).

```
>>> resultado = muestra_dos_veces('Bing')
Bing
Bing
>>> print(resultado)
None
```

El valor `None` no es el mismo que la cadena "None". Es un valor especial que tiene su propio tipo:

```
>>> print(type(None))
<class 'NoneType'>
```

Para devolver un resultado desde una función, usamos la sentencia `return` dentro de ella. Por ejemplo, podemos crear una función muy simple llamada `sumados`, que suma dos números y devuelve el resultado.

```
def sumados(a, b):
    suma = a + b
    return suma

x = sumados(3, 5)
print(x)

# Code: http://www.py4e.com/code3/addtwo.py
```

Cuando se ejecuta este script, la sentencia `print` mostrará "8", ya que la función `sumados` ha sido llamada con 3 y 5 como argumentos. Dentro de la función, los parámetros `a` y `b` equivaldrán a 3 y a 5 respectivamente. La función calculó la suma de ambos número y la guardó en una variable local a la función llamada `suma`. Después usó la sentencia `return` para enviar el valor calculado de vuelta al código de llamada como resultado de la función, que fue asignado a la variable `x` y mostrado en pantalla.

4.11 ¿Por qué funciones?

Puede no estar muy claro por qué merece la pena molestarse en dividir un programa en funciones. Existen varias razones:

- El crear una función nueva te da la oportunidad de dar nombre a un grupo de sentencias, lo cual hace tu programa más fácil de leer, entender y depurar.
- Las funciones pueden hacer un programa más pequeño, al eliminar código repetido. Además, si quieres realizar cualquier cambio en el futuro, sólo tendrás que hacerlo en un único lugar.
- Dividir un programa largo en funciones te permite depurar las partes de una en una y luego ensamblarlas juntas en una sola pieza.
- Las funciones bien diseñadas a menudo resultan útiles para otros muchos programas. Una vez que has escrito y depurado una, puedes reutilizarla.

A lo largo del resto del libro, a menudo usaremos una definición de función para explicar un concepto. Parte de la habilidad de crear y usar funciones consiste en llegar a tener una función que represente correctamente una idea, como "encontrar el valor más pequeño en una lista de valores". Más adelante te mostraremos el código para encontrar el valor más pequeño de una lista de valores y te lo presentaremos como una función llamada `min`, que toma una lista de valores como argumento y devuelve el menor valor de esa lista.

4.12 Depuración

Si estás usando un editor de texto para escribir tus propios scripts, puede que tengas problemas con los espacios y tabulaciones. El mejor modo de evitar esos problemas es usar espacios exclusivamente (no tabulaciones). La mayoría de los editores de texto que reconocen Python lo hacen así por defecto, aunque hay algunos que no.

Las tabulaciones y los espacios normalmente son invisibles, lo cual hace que sea difícil depurar los errores que se pueden producir, así que mejor busca un editor que gestione el indentado por ti.

Tampoco te olvides de guardar tu programa antes de hacerlo funcionar. Algunos entornos de desarrollo lo hacen automáticamente, pero otros no. En ese caso, el programa que estás viendo en el editor de texto puede no ser el mismo que estás ejecutando en realidad.

¡La depuración puede llevar mucho tiempo si estás haciendo funcionar el mismo programa con errores una y otra vez!

Asegúrate de que el código que estás examinando es el mismo que estás ejecutando. Si no estás seguro, pon algo como `print("hola")` al principio del programa y hazlo funcionar de nuevo. Si no ves `hola` en la pantalla, ¡es que no estás ejecutando el programa correcto!

4.13 Glosario

algoritmo Un proceso general para resolver una categoría de problemas.

argumento Un valor proporcionado a una función cuando ésta es llamada. Ese valor se asigna al parámetro correspondiente en la función.

cabecera La primera línea de una definición de función.

cuerpo La secuencia de sentencias dentro de la definición de una función.

composición Uso de una expresión o sentencia como parte de otra más larga,

definición de función Una sentencia que crea una función nueva, especificando su nombre, parámetros, y las sentencias que ejecuta.

determinístico Perteneciente a un programa que hace lo mismo cada vez que se ejecuta, a partir de las mismas entradas.

función Una secuencia de sentencias con un nombre que realizan alguna operación útil. Las funciones pueden tomar argumentos o no, y pueden producir un resultado o no.

función productiva (fruitful function) Una función que devuelve un valor.

función estéril (void function) Una función que no devuelve ningún valor.

flujo de ejecución El orden en el cual se ejecutan las sentencias durante el funcionamiento de un programa.

llamada a función Una sentencia que ejecuta una función. Consiste en el nombre de la función seguido por una lista de argumentos.

notación punto La sintaxis para llamar a una función de otro módulo, especificando el nombre del módulo seguido por un punto y el nombre de la función.

objeto función Un valor creado por una definición de función. El nombre de la función es una variable que se refiere al objeto función.

objeto módulo Un valor creado por una sentencia `import`, que proporciona acceso a los datos y código definidos en un módulo.

parámetro Un nombre usado dentro de una función para referirse al valor pasado como argumento.

pseudoaleatorio Perteneciente a una secuencia de números que parecen ser aleatorios, pero son generados por un programa determinista.

sentencia import Una sentencia que lee un archivo módulo y crea un objeto módulo.

valor de retorno El resultado de una función. Si una llamada a una función es usada como una expresión, el valor de retorno es el valor de la expresión.

4.14 Ejercicios

Ejercicio 4: ¿Cuál es la utilidad de la palabra clave "def" en Python?

a) Es una jerga que significa "este código es realmente estupendo"
b) Indica el comienzo de una función
c) Indica que la siguiente sección de código indentado debe ser almacenada para usarla más tarde
d) b y c son correctas ambas
e) Ninguna de las anteriores

Ejercicio 5: ¿Qué mostrará en pantalla el siguiente programa Python?

```
def fred():
   print("Zap")

def jane():
   print("ABC")

jane()
fred()
jane()
```

a) Zap ABC jane fred jane
b) Zap ABC Zap
c) ABC Zap jane
d) ABC Zap ABC
e) Zap Zap Zap

Ejercicio 6: Reescribe el programa de cálculo del salario, con tarifa-y-media para las horas extras, y crea una función llamada `calculo_salario` que reciba dos parámetros (`horas` y `tarifa`).

```
Introduzca Horas: 45
Introduzca Tarifa: 10
Salario: 475.0
```

Ejercicio 7: Reescribe el programa de calificaciones del capítulo anterior usando una función llamada `calcula_calificacion`, que reciba una puntuación como parámetro y devuelva una calificación como cadena.

```
Puntuación Calificación
> 0.9      Sobresaliente
> 0.8      Notable
> 0.7      Bien
> 0.6      Suficiente
<= 0.6     Insuficiente


Introduzca puntuación: 0.95
Sobresaliente
```

```
Introduzca puntuación: perfecto
Puntuación incorrecta


Introduzca puntuación: 10.0
Puntuación incorrecta


Introduzca puntuación: 0.75
Bien


Introduzca puntuación: 0.5
Insuficiente
```

Ejecuta el programa repetidamente para probar con varios valores de entrada diferentes.

Chapter 5

Iteración

5.1 Actualización de variables

Uno de los usos habituales de las sentencia de asignación consiste en realizar una actualización sobre una variable – en la cual el valor nuevo de esa variable depende del antiguo.

```
x = x + 1
```

Esto quiere decir "'toma el valor actual de `x`, añádele 1, y luego actualiza `x` con el nuevo valor".

Si intentas actualizar una variable que no existe, obtendrás un error, ya que Python evalúa el lado derecho antes de asignar el valor a `x`:

```
>>> x = x + 1
NameError: name 'x' is not defined
```

Antes de que puedas actualizar una variable, debes *inicializarla*, normalmente mediante una simple asignación:

```
>>> x = 0
>>> x = x + 1
```

Actualizar una variable añadiéndole 1 se denomina *incrementar*; restarle 1 recibe el nombre de *decrementar* (o disminuir).

5.2 La sentencia `while`

Los PCs se suelen utilizar a menudo para automatizar tareas repetitivas. Repetir tareas idénticas o muy similares sin cometer errores es algo que a las máquinas se les da bien y en cambio a las personas no. Como las iteraciones resultan tan

habituales, Python proporciona varias características en su lenguaje para hacerlas más sencillas.

Una forma de iteración en Python es la sentencia `while`. He aquí un programa sencillo que cuenta hacia atrás desde cinco y luego dice "¡Despegue!".

```
n = 5
while n > 0:
    print(n)
    n = n - 1
print('¡Despegue!')
```

Casi se puede leer la sentencia `while` como si estuviera escrita en inglés. Significa, "Mientras `n` sea mayor que 0, muestra el valor de `n` y luego reduce el valor de `n` en 1 unidad. Cuando llegues a 0, sal de la sentencia `while` y muestra la palabra `¡Despegue!`"

Éste es el flujo de ejecución de la sentencia `while`, explicado de un modo más formal:

1. Se evalúa la condición, obteniendo `Verdadero` or `Falso`.
2. Si la condición es falsa, se sale de la sentencia `while` y se continúa la ejecución en la siguiente sentencia.
3. Si la condición es verdadera, se ejecuta el cuerpo del `while` y luego se vuelve al paso 1.

Este tipo de flujo recibe el nombre de *bucle*, ya que el tercer paso enlaza de nuevo con el primero. Cada vez que se ejecuta el cuerpo del bucle se dice que realizamos una *iteración*. Para el bucle anterior, podríamos decir que "ha tenido cinco iteraciones", lo que significa que el cuerpo del bucle se ha ejecutado cinco veces.

El cuerpo del bucle debe cambiar el valor de una o más variables, de modo que la condición pueda en algún momento evaluarse como falsa y el bucle termine. La variable que cambia cada vez que el bucle se ejecuta y controla cuándo termina éste, recibe el nombre de *variable de iteración*. Si no hay variable de iteración, el bucle se repetirá para siempre, resultando así un *bucle infinito*.

5.3 Bucles infinitos

Una fuente de diversión sin fin para los programadores es la constatación de que las instrucciones del champú: "Enjabone, aclare, repita", son un bucle infinito, ya que no hay una *variable de iteración* que diga cuántas veces debe ejecutarse el proceso.

En el caso de una `cuenta atrás`, podemos verificar que el bucle termina, ya que sabemos que el valor de `n` es finito, y podemos ver que ese valor se va haciendo más pequeño cada vez que se repite el bucle, de modo que en algún momento llegará a 0. Otras veces un bucle es obviamente infinito, porque no tiene ninguna variable de iteración.

5.4 "Bucles infinitos" y break

A veces no se sabe si hay que terminar un bucle hasta que se ha recorrido la mitad del cuerpo del mismo. En ese caso se puede crear un bucle infinito a propósito y usar la sentencia `break` para salir fuera de él cuando se desee.

El bucle siguiente es, obviamente, un *bucle infinito*, porque la expresión lógica de la sentencia `while` es simplemente la constante lógica `True (verdadero)`;

```
n = 10
while True:
    print(n, end=' ')
    n = n - 1
print('¡Terminado!')
```

Si cometes el error de ejecutar este código, aprenderás rápidamente cómo detener un proceso de Python bloqueado en el sistema, o tendrás que localizar dónde se encuentra el botón de apagado de tu equipo. Este programa funcionará para siempre, o hasta que la batería del equipo se termine, ya que la expresión lógica al principio del bucle es siempre cierta, en virtud del hecho de que esa expresión es precisamente el valor constante `True`.

A pesar de que en este caso se trata de un bucle infinito inútil, se puede usar ese diseño para construir bucles útiles, siempre que se tenga la precaución de añadir código en el cuerpo del bucle para salir explícitamente, usando `break` cuando se haya alcanzado la condición de salida.

Por ejemplo, supón que quieres recoger entradas de texto del usuario hasta que éste escriba `fin`. Podrías escribir:

```
while True:
    linea = input('> ')
    if linea == 'fin':
        break
    print(linea)
print('¡Terminado!')

# Code: http://www.py4e.com/code3/copytildone1.py
```

La condición del bucle es `True`, lo cual es verdadero siempre, así que el bucle se repetirá hasta que se ejecute la sentencia break.

Cada vez que se entre en el bucle, se pedirá una entrada al usuario. Si el usuario escribe `fin`, la sentencia `break` hará que se salga del bucle. En cualquier otro caso, el programa repetirá cualquier cosa que el usuario escriba y volverá al principio del bucle. Éste es un ejemplo de su funcionamiento:

```
> hola a todos
hola a todos
> he terminado
he terminado
> fin
¡Terminado!
```

Este modo de escribir bucles `while` es habitual, ya que así se puede comprobar la condición en cualquier punto del bucle (no sólo al principio), y se puede expresar la condición de parada afirmativamente ("detente cuando ocurra..."), en vez de tener que hacerlo con lógica negativa ("sigue haciéndolo hasta que ocurra...").

5.5 Finalizar iteraciones con `continue`

Algunas veces, estando dentro de un bucle se necesita terminar con la iteración actual y saltar a la siguiente de forma inmediata. En ese caso se puede utilizar la sentencia `continue` para pasar a la siguiente iteración sin terminar la ejecución del cuerpo del bucle para la actual.

A continuación se muestra un ejemplo de un bucle que repite lo que recibe como entrada hasta que el usuario escribe "fin", pero trata las líneas que empiezan por el carácter almohadilla como líneas que no deben mostrarse en pantalla (algo parecido a lo que hace Python con los comentarios).

```
while True:
    linea = input('> ')
    if linea[0] == '#' :
        continue
    if linea == 'fin':
        break
    print(linea)
print('¡Terminado!')

# Code: http://www.py4e.com/code3/copytildone2.py
```

He aquí una ejecución de ejemplo de ese nuevo programa con la sentencia `continue` añadida.

```
> hola a todos
hola a todos
> # no imprimas esto
> ¡imprime esto!
¡imprime esto!
> fin
¡Terminado!
```

Todas las líneas se imprimen en pantalla, excepto la que comienza con el símbolo de almohadilla, ya que en ese caso se ejecuta `continue`, finaliza la iteración actual y salta de vuelta a la sentencia `while` para comenzar la siguiente iteración, de modo que que se omite la sentencia `print`.

5.6 Bucles definidos usando `for`

A veces se desea repetir un bucle a través de un *conjunto* de cosas, como una lista de palabras, las líneas de un archivo, o una lista de números. Cuando se

tiene una lista de cosas para recorrer, se puede construir un bucle *definido* usando una sentencia `for`. A la sentencia `while` se la llama un bucle *indefinido*, porque simplemente se repite hasta que cierta condición se hace `Falsa`, mientras que el bucle `for` se repite a través de un conjunto conocido de elementos, de modo que ejecuta tantas iteraciones como elementos hay en el conjunto.

La sintaxis de un bucle `for` es similar a la del bucle `while`, en ella hay una sentencia `for` y un cuerpo que se repite:

```
amigos = ['Joseph', 'Glenn', 'Sally']
for amigo in amigos:
    print('Feliz año nuevo:', amigo)
print('¡Terminado!')
```

En términos de Python, la variable `amigos` es una lista[1] de tres cadenas y el bucle `for` se mueve recorriendo la lista y ejecuta su cuerpo una vez para cada una de las tres cadenas en la lista, produciendo esta salida:

```
Feliz año nuevo: Joseph
Feliz año nuevo: Glenn
Feliz año nuevo: Sally
¡Terminado!
```

La traducción de este bucle `for` al español no es tan directa como en el caso del `while`, pero si piensas en los amigos como un *conjunto*, sería algo así como: "Ejecuta las sentencias en el cuerpo del bucle una vez para cada amigo que esté *en (in)* el conjunto llamado amigos."

Revisando el bucle, `for`, *for* e *in* son palabras reservadas de Python, mientras que `amigo` y `amigos` son variables.

```
for amigo in amigos:
    print('Feliz año nuevo::', amigo)
```

En concreto, `amigo` es la *variable de iteración* para el bucle for. La variable `amigo` cambia para cada iteración del bucle y controla cuándo se termina el bucle `for`. La *variable de iteracion* se desplaza sucesivamente a través de las tres cadenas almacenadas en la variable `amigos`.

5.7 Diseños de bucles

A menudo se usa un bucle `for` o `while` para movernos a través de una lista de elementos o el contenido de un archivo y se busca algo, como el valor más grande o el más pequeño de los datos que estamos revisando.

Los bucles generalmente se construyen así:

- Se inicializan una o más variables antes de que el bucle comience

[1] Examinaremos las listas con más detalle en un capítulo posterior.

- Se realiza alguna operación con cada elemento en el cuerpo del bucle, posiblemente cambiando las variables dentro de ese cuerpo.
- Se revisan las variables resultantes cuando el bucle se completa

Usaremos ahora una lista de números para demostrar los conceptos y construcción de estos diseños de bucles.

5.7.1 Bucles de recuento y suma

Por ejemplo, para contar el número de elementos en una lista, podemos escribir el siguiente bucle `for`:

```
contador = 0
for valor in [3, 41, 12, 9, 74, 15]:
    contador = contador + 1
print('Num. elementos: ', contador)
```

Ajustamos la variable `contador` a cero antes de que el bucle comience, después escribimos un bucle `for` para movernos a través de la lista de números. Nuestra variable de *iteración* se llama `valor`, y dado que no usamos `valor` dentro del bucle, lo único que hace es controlar el bucle y hacer que el cuerpo del mismo sea ejecutado una vez para cada uno de los valores de la lista.

En el cuerpo del bucle, añadimos 1 al valor actual de `contador` para cada uno de los valores de la lista. Mientras el bucle se está ejecutando, el valor de `contador` es la cantidad de valores que se hayan visto "hasta ese momento".

Una vez el bucle se completa, el valor de `contador` es el número total de elementos. El número total "cae en nuestro poder" al final del bucle. Se construye el bucle de modo que obtengamos lo que queremos cuando éste termina.

Otro bucle similar, que calcula el total de un conjunto de números, se muestra a continuación:

```
total = 0
for valor in [3, 41, 12, 9, 74, 15]:
    total = total + valor
print('Total: ', total)
```

En este bucle, *sí* utilizamos la *variable de iteración*. En vez de añadir simplemente uno a `contador` como en el bucle previo, ahora durante cada iteración del bucle añadimos el número actual (3, 41, 12, etc.) al total en ese momento. Si piensas en la variable `total`, ésta contiene la "suma parcial de valores hasta ese momento". Así que antes de que el bucle comience, `total` es cero, porque aún no se ha examinado ningún valor. Durante el bucle, `total` es la suma parcial, y al final del bucle, `total` es la suma total definitiva de todos los valores de la lista.

Cuando el bucle se ejecuta, `total` acumula la suma de los elementos; una variable que se usa de este modo recibe a veces el nombre de *acumulador*.

Ni el bucle que cuenta los elementos ni el que los suma resultan particularmente útiles en la práctica, dado que existen las funciones internas `len()` y `sum()` que cuentan el número de elementos de una lista y el total de elementos en la misma respectivamente.

5.7.2 Bucles de máximos y mínimos

[maximumloop] Para encontrar el valor mayor de una lista o secuencia, construimos el bucle siguiente:

```
mayor = None
print('Antes:', mayor)
for valor in [3, 41, 12, 9, 74, 15]:
    if mayor is None or valor > mayor :
        mayor = valor
    print('Bucle:', valor, mayor)
print('Mayor:', mayor)
```

Cuando se ejecuta el programa, se obtiene la siguiente salida:

```
Antes: None
Bucle: 3 3
Bucle: 41 41
Bucle: 12 41
Bucle: 9 41
Bucle: 74 74
Bucle: 15 74
Mayor: 74
```

Debemos pensar en la variable `mayor` como el "mayor valor visto hasta ese momento". Antes del bucle, asignamos a `mayor` el valor `None`. `None` es un valor constante especial que se puede almacenar en una variable para indicar que la variable está "vacía".

Antes de que el bucle comience, el mayor valor visto hasta entonces es `None`, dado que no se ha visto aún ningún valor. Durante la ejecución del bucle, si `mayor` es `None`, entonces tomamos el primer valor que tenemos como el mayor hasta entonces. Se puede ver en la primera iteración, cuando el valor de `valor` es 3, mientras que `mayor` es `None`, inmediatamente hacemos que `mayor` pase a ser 3.

Tras la primera iteración, `mayor` ya no es `None`, así que la segunda parte de la expresión lógica compuesta que comprueba si `valor > mayor` se activará sólo cuando encontremos un valor que sea mayor que el "mayor hasta ese momento". Cuando encontramos un nuevo valor "mayor aún", tomamos ese nuevo valor para `mayor`. Se puede ver en la salida del programa que `mayor` pasa desde 3 a 41 y luego a 74.

Al final del bucle, se habrán revisado todos los valores y la variable `mayor` contendrá entonces el valor más grande de la lista.

Para calcular el número más pequeño, el código es muy similar con un pequeño cambio:

```
print('Antes:', menor)
for valor in [3, 41, 12, 9, 74, 15]:
    if menor is None or valor < menor:
        menor = valor
    print('Bucle:', valor, menor)
print('Menor:', menor)
```

De nuevo, `menor` es el "menor hasta ese momento" antes, durante y después de que el bucle se ejecute. Cuando el bucle se ha completado, `menor` contendrá el valor mínimo de la lista

También como en el caso del número de elementos y de la suma, las funciones internas `max()` y `min()` convierten la escritura de este tipo de bucles en innecesaria.

Lo siguiente es una versión simple de la función interna de Python `min()`:

```
def min(valores):
    menor = None
    for valor in valores:
        if menor is None or valor < menor:
            menor = valor
    return menor
```

En esta versión de la función para calcular el mínimo, hemos eliminado las sentencias `print`, de modo que sea equivalente a la función `min`, que ya está incorporada dentro de Python.

5.8 Depuración

A medida que vayas escribiendo programas más grandes, puede que notes que vas necesitando emplear cada vez más tiempo en depurarlos. Más código significa más oportunidades de cometer un error y más lugares donde los bugs pueden esconderse.

Un método para acortar el tiempo de depuración es "depurar por bisección". Por ejemplo, si hay 100 líneas en tu programa y las compruebas de una en una, te llevará 100 pasos.

En lugar de eso, intenta partir el problema por la mitad. Busca en medio del programa, o cerca de ahí, un valor intermedio que puedas comprobar. Añade una sentencia `print` (o alguna otra cosa que tenga un efecto verificable), y haz funcionar el programa.

Si en el punto medio la verificación es incorrecta, el problema debería estar en la primera mitad del programa. Si ésta es correcta, el problema estará en la segunda mitad.

Cada vez que realices una comprobación como esta, reduces a la mitad el número de líneas en las que buscar. Después de seis pasos (que son muchos menos de 100), lo habrás reducido a una o dos líneas de código, al menos en teoría.

En la práctica no siempre está claro qué es "el medio del programa", y no siempre es posible colocar ahí una verificación. No tiene sentido contar las líneas y encontrar el

punto medio exacto. En lugar de eso, piensa en lugares del programa en los cuales pueda haber errores y en lugares donde resulte fácil colocar una comprobación. Luego elige un sitio donde estimes que las oportunidades de que el bug esté por delante y las de que esté por detrás de esa comprobación son más o menos las mismas.

5.9 Glosario

acumulador Una variable usada en un bucle para sumar o acumular un resultado.

bucle infinito Un bucle en el cual la condición de terminación no se satisface nunca o para el cual no existe dicha condición de terminación.

contador Una variable usada en un bucle para contar el número de veces que algo sucede. Inicializamos el contador a cero y luego lo vamos incrementando cada vez que queramos que "cuente" algo.

decremento Una actualización que disminuye el valor de una variable.

inicializar Una asignación que da un valor inicial a una variable que va a ser después actualizada.

incremento Una actualización que aumenta el valor de una variable (a menudo en una unidad).

iteración Ejecución repetida de una serie de sentencias usando bien una función que se llama a si misma o bien un bucle.

5.10 Ejercicios

Exercise 1: Escribe un programa que lea repetidamente números hasta que el usuario introduzca "fin". Una vez se haya introducido "fin", muestra por pantalla el total, la cantidad de números y la media de esos números. Si el usuario introduce cualquier otra cosa que no sea un número, detecta su fallo usando `try` y `except`, muestra un mensaje de error y pasa al número siguiente.

```
Introduzca un número: 4
Introduzca un número: 5
Introduzca un número: dato erróneo
Entrada inválida
Introduzca un número: 7
Introduzca un número: fin
16 3 5.33333333333
```

Exercise 2: Escribe otro programa que pida una lista de números como la anterior y al final muestre por pantalla el máximo y mínimo de los números, en vez de la media.

Chapter 6

Cadenas

6.1 Una cadena es una secuencia

Una cadena es una *secuencia* de caracteres. Puedes acceder los caracteres uno a la vez con el operador corchete:

```
>>> fruta = 'banana'
>>> letra = fruta[1]
```

La segunda sentencia extrae el caracter en la posición del indice 1 de la variable `fruta` y la asigna a la variable `letra`.

La expresión en los corchetes es llamada un *índice*. El índice indica cuál caracter de la secuencia quieres (de aquí el nombre).

Pero podrías no obtener lo que esperas:

```
>>> print(letra)
a
```

Para la mayoría de las personas, la primer letra de "banana" es "b", no "a". Pero en Python, el índice es un desfase desde el inicio de la cadena, y el desfase de la primer letra es cero.

```
>>> letra = fruta[0]
>>> print(letra)
b
```

Así que "b" es la letra 0 ("cero") de "banana", "a" es la 1er letra ("primer"), y "n" es la 2da ("segunda") letra.

Puedes usar cualquier expresión, incluyendo variables y operadores, como un índice, pero el valor del índice tiene que ser un entero. De otro modo obtendrás:

```
>>> letra = fruta[1.5]
TypeError: string indices must be integers
```

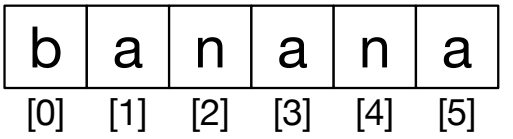

Figure 6.1: Indices de Cadenas

6.2 Obtener el tamaño de una cadena usando `len`

`len` es una función nativa que regresa el número de caracteres en una cadena:

```
>>> fruta = 'banana'
>>> len(fruta)
6
```

Para obtener la última letra de una cadena, podrías estar tentado a probar algo como esto:

```
>>> tamaño = len(fruta)
>>> ultima = fruta[tamaño]
IndexError: string index out of range
```

La razón de que haya un `IndexError` es que ahí no hay ninguna letra en "banana" con el índice 6. Puesto que empezamos a contar desde cero, las seis letras están enumeradas desde 0 hasta 5. Para obtener el último caracter, tienes que restar 1 del `length`:

```
>>> ultima = fruta[tamaño-1]
>>> print(ultima)
a
```

Alternativamente, puedes usar índices negativos, los cuales cuentan hacia atrás desde el final de la cadena. La expresión `fruta[-1]` regresa la última letra, `fruta[-2]` regresa la penúltima letra, y así sucesivamente.

6.3 Recorriendo una cadena mediante un bucle

Muchos de los cálculos requieren procesar una cadena con un caracter a la vez. Frecuentemente empiezan desde el inicio, seleccionando cada caracter presente, hacer algo con él, y continuar hasta el final. Este patrón de procesamiento es llamado un *recorrido*. Una manera de escribir un recorrido es con un bucle `while`:

```
indice = 0
while indice < len(fruta):
    letra = fruta[indice]
    print(letra)
    indice = indice + 1
```

Este bucle recorre la cadena e imprime cada letra en una línea cada una. La condición del bucle es `indice < len(fruta)`, así que cuando `indice` es igual al tamaño de la cadena, la condición es falsa, y el código del bucle no se ejecuta. El último caracter accedido es el que tiene el índice `len(fruta)-1`, el cual es el último caracter en la cadena.

Ejercicio 1: Escribe un bucle `while` que comienza con el último caracter en la cadena y trabaja hacia atrás hasta el primer caracter en la cadena, imprimiendo cada letra en una línea independiente, excepto al revés.

Otra forma de escribir un recorrido es con un bucle `for`:

```
for caracter in fruta:
    print(caracter)
```

Cada vez que iteramos el bucle, el siguiente caracter en la cadena es asignado a la variable `caracter`. El ciclo continua hasta que no quedan caracteres.

6.4 Parte de una cadena

Un segmento de una cadena es llamado *parte*. Seleccionar una parte es similar a seleccionar un caracter:

```
>>> s = 'Monty Python'
>>> print(s[0:5])
Monty
>>> print(s[6:12])
Python
```

El operador retorna la parte de la cadena desde el "n-avo" caracter hasta el "m-avo" caracter, incluyendo el primero pero excluyendo el último.

Si omites el primer índice (antes de los dos puntos), la parte comienza desde el inicio de la cadena. Si omites el segundo índice, la parte va hasta el final de la cadena:

```
>>> fruta = 'banana'
>>> fruta[:3]
'ban'
>>> fruta[3:]
'ana'
```

Si el primer índice es mayor que o igual que el segundo, el resultado es una *cadena vacía*, representado por dos comillas:

```
>>> fruta = 'banana'
>>> fruta[3:3]
''
```

Una cadena vacía no contiene caracteres y tiene un tamaño de 0, pero fuera de esto, es lo mismo que cualquier otra cadena.

Ejercicio 2: Dado que `fruta` es una cadena, ¿que significa `fruta[:]`?